UNIVERSAL SYLLABUS · ARTIFICIAL INTELLIGENCE

Artificial Intelligence

From depth-first to A* search

The notes for the lesson “From depth-first to A* search”: chapters 2.3 to 2.5, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

3 chapters | Chapters 2.3 to 2.5

About these notes

These are the notes for the University free n easy video lesson **From depth-first to A* search** on Artificial Intelligence: chapters 2.3 to 2.5, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

2.3 Depth-first, depth-limited and iterative deepening search

Depth-first search (DFS) always expands the deepest node in the frontier: it dives down one branch until it reaches a dead end, then backs up and tries the next branch. It needs very little memory, but it can wander down an endless branch and miss a goal close to the root. **Depth-limited search** (DLS) stops the dive at a fixed depth, and **iterative deepening search** (IDS) runs DLS with limits 0, 1, 2, ... until a goal turns up. IDS gets the guarantees of BFS with the memory of DFS, and it is the usual choice of uninformed search when the space is large and the solution depth is unknown.

You need: the search loop and the symbols b, d, m from 2.1, and BFS from 2.2 for comparison.

Depth-first search

DFS keeps the frontier in a LIFO stack, or uses recursion, which is the same thing: the newest node is expanded first. Its properties are mixed:

- **Not complete** in general. The tree-search version can loop (A → B → A → ...), and in an infinite space any version can go down an endless path. The graph-search version is complete on a finite space.
- **Not optimal.** On the delivery map of Figure 2.2.1, with neighbours in alphabetical order, DFS visits S, A, C, D, B, G and returns S–A–C–D–G, 11 minutes: neither the fewest roads (BFS, 3) nor

the quickest (UCS, 9 minutes).

- **Time $O(b^m)$**, where m is the deepest path. This can be far worse than b^d when m is much larger than d.
- **Space O(bm)**. DFS stores only the current path and the unexpanded siblings of each node on it. With b = 10 and m = 12 that is about 120 nodes, against about 10^{12} for BFS at depth 12. A variant called **backtracking search** generates one child at a time and needs only O(m).

Depth-limited search

DLS is DFS with a depth limit ℓ: a node at depth ℓ is treated as if it had no children. It can end in three ways: a **solution**; **failure**, meaning no solution exists at all; or **cutoff**, meaning none was found within the limit. It is incomplete if $\ell < d$ and not optimal if $\ell > d$. Time is $O(b^\ell)$ and space $O(b\ell)$. Knowledge of the problem sets ℓ: on a map of 20 towns no route needs more than 19 roads, so $\ell = 19$ is safe, and if any town reaches any other in at most 9 roads (the map's *diameter*), $\ell = 9$ is better. The recursive form, DLS(n, ℓ):

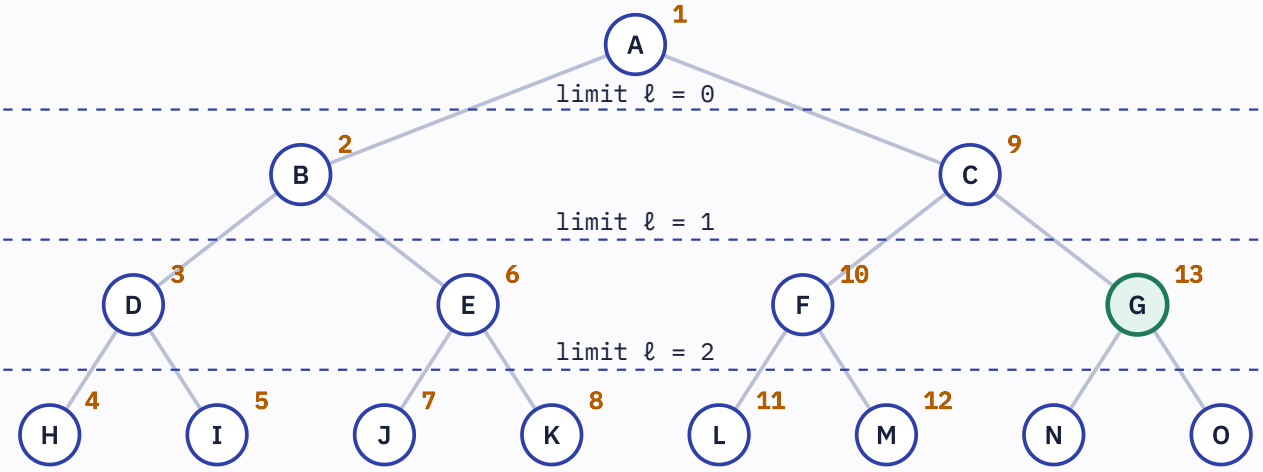
1. If n is a goal, return the solution.
2. If $\ell = 0$, return cutoff.
3. For each child c of n: let r = DLS(c, $\ell - 1$). If r is a solution, return it; if r is cutoff, remember that a cutoff happened.
4. Return cutoff if any child was cut off; otherwise return failure.

Iterative deepening search

IDS calls DLS with $\ell = 0$, then 1, then 2, and so on, and stops at the first result that is not cutoff. It looks wasteful, since the top of the tree is generated again in every round, but the top is small. With the goal at depth d, nodes at depth d are generated once, at depth d – 1 twice, and at depth 1 d times:

$$N(IDS) = d \cdot b + (d - 1) \cdot b^2 + \dots + 2 \cdot b^{d-1} + 1 \cdot b^d$$

Most nodes sit on the bottom level, so the total is still $O(b^d)$. IDS is **complete** when b is finite, **optimal** when all steps cost the same (it finds the shallowest goal first, like BFS), and needs only **O(bd)** space, like DFS. The repeated work matters most when b is small: as d grows, the ratio N(IDS)/N(BFS) approaches b/(b – 1), which is 1.11 for b = 10 but 2 for b = 2.



Orange numbers: the order in which DFS visits nodes. It reaches the goal G 13th.
Dashed lines: IDS with limit ℓ never goes below the line marked ℓ.

Figure 2.3.1 A binary tree (b = 2) of depth 3 with the goal G at depth 2 on the right. DFS goes through all of B's subtree (visits 2 to 8) and F's (10 to 12) before it reaches G on visit 13. IDS with ℓ = 2 stays above the line marked ℓ = 2 and reaches G on the 7th visit of that round.

Comparing the uninformed strategies

CRITERION	BFS	UCS	DFS	DLS	IDS
Complete?	yes[1]	yes[1] [2]	no	only if ℓ ≥ d	yes[1]
Optimal?	yes[3]	yes	no	no	yes[3]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$

[1] if b is finite; [2] if every step costs at least ε > 0; [3] if all steps cost the same. Bidirectional search is added in 2.4.

WORKED EXAMPLE

Question 1. In the tree of Figure 2.3.1 the goal is G. Children are taken left to right. List the nodes goal-tested by (a) DFS, (b) DLS with $\ell = 1$ and (c) IDS, and compare the counts with BFS.

1. (a) DFS: A B D H I E J K C F L M G. Goal found on visit **13**.
2. (b) DLS, $\ell = 1$: A B C. B and C are at the limit, so they are not expanded. Result: **cutoff** (G is at depth 2, below the limit).
3. (c) IDS. $\ell = 0$: A, cutoff. $\ell = 1$: A B C, cutoff. $\ell = 2$: A B D E C F G, goal. Total $1 + 3 + 7 =$ **11** visits.
4. BFS: A B C D E F G, **7** visits, but with the whole level stored at once.

IDS beats DFS here because the goal is shallow and on the right. If the goal were H, DFS would find it on visit 4.

Question 2. For $b = 10$ and a goal at depth $d = 5$, compare the nodes generated by IDS and BFS (goal tested at generation), and the memory each needs.

1. IDS: $5 \cdot 10 + 4 \cdot 100 + 3 \cdot 1{,}000 + 2 \cdot 10{,}000 + 1 \cdot 100{,}000 = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 =$ **123,450**.
2. BFS: $10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 =$ **111,110**.
3. Extra work: $123{,}450 - 111{,}110 = 12{,}340$ nodes, about **11%**.
4. Memory: IDS holds about $b \cdot d = 50$ nodes; BFS holds up to 100,000 nodes of the last level.

EXAM TRAPS

- DFS is neither optimal nor, as tree search, complete. Say why: a loop or an infinite branch.
- DLS has two kinds of "no": cutoff (maybe a solution deeper down) and failure (none anywhere). IDS keeps going after cutoff and stops on failure.
- Write $O(bm)$ for DFS space but $O(bd)$ for IDS space; IDS never goes deeper than d.
- "IDS is wasteful" is the wrong conclusion: the overhead is about 11% for $b = 10$. It does approach 100% when $b = 2$.
- Favourite question: "Explain iterative deepening and show that it combines the advantages of BFS and DFS", often with a tree to trace.

CHECK YOURSELF

– REMEMBER **What data structure holds the DFS frontier?** 1 mark

A LIFO stack, or the call stack when DFS is written recursively.

– REMEMBER **Name the three results depth-limited search can return.** 2 marks

A solution, failure (no solution exists) and cutoff (no solution within the limit ℓ).

– UNDERSTAND **Why does DFS need only O(bm) memory?** 2 marks

Once a subtree is fully explored it is dropped. DFS keeps only the current path, at most m nodes, plus up to b unexpanded siblings of each.

– UNDERSTAND **Why is IDS optimal for equal step costs when DFS is not?** 2 marks

IDS finishes every depth ℓ before trying $\ell + 1$, so the first goal it meets is a shallowest one. DFS returns the first goal on whichever branch it dives into, however deep.

– APPLY **For b = 3 and d = 4, find the nodes generated by IDS and by BFS.** 3 marks

IDS: $4 \cdot 3 + 3 \cdot 9 + 2 \cdot 27 + 1 \cdot 81 = 12 + 27 + 54 + 81 =$ **174**. BFS: $3 + 9 + 27 + 81 =$ **120**. IDS does 45% more work.

– APPLY **In Figure 2.3.1, with goal L instead of G, list the nodes IDS visits in its $\ell = 3$ round.** 2 marks

A B D H I E J K C F L: the goal is found on the 11th visit of that round.

– ANALYZE **A puzzle has b = 2, solutions near depth 30, infinite paths, and 1 GB of memory for the search. Choose an uninformed strategy and justify it with numbers.** 4 marks

BFS would store about $2^{30} \approx 1.07 \times 10^9$ nodes, about 1 TB at 1 KB per node: too much. DFS may follow an infinite path. IDS stores about $b \cdot d = 60$ nodes and finds the shallowest solution; it takes roughly twice BFS's time at b = 2, which is acceptable. Choose IDS.

BEYOND THE SYLLABUS

Not for the exam. Richard Korf's 1985 analysis made iterative deepening standard. Chess engines use it at every move: each deeper round orders moves using the best move of the previous round, and the engine can stop at any time and play the best move found so far (alpha-beta pruning, Unit 3). The same idea with cost limits instead of depth limits gives iterative lengthening and IDA* (Chapter 2.5). Search next: "iterative deepening in chess engines", "anytime algorithms", "iterative lengthening search".

2.4 Bidirectional search

Bidirectional search runs two searches at once, one forward from the start and one backward from the goal, and stops when the two frontiers meet. Each search goes only half the distance, and because a search tree grows exponentially, two half-depth trees are far smaller than one full-depth tree: about $2b^{d/2}$ nodes instead of b^d. Road-routing software uses the idea widely.

You need: BFS and UCS from 2.2, and goal-driven search from 2.1.

How it works

1. Put the start state in the forward frontier and the goal state in the backward frontier.
2. Alternate: expand one level (or one node) forward, using the actions; then one backward, using **predecessors**, the states from which one action leads to the current state.
3. After each expansion, check every new state against the other search's frontier and explored set. On a match the frontiers have met: join the forward path to that state with the reversed backward path.

Two conditions must hold. The goal must be one explicit state, not a test such as "any position where the king is checkmated". And predecessors must be computable, which is easy when actions can be undone (roads, sliding-tile moves) and hard when they cannot.

Why it saves time and memory

Each side reaches depth d/2, so time is $O(b^{d/2})$. With b = 10 and the goal at d = 6, one BFS generates $10 + 100 + \ldots + 10^6 = 1{,}111{,}110$ nodes, while two BFS runs to depth 3 generate 2 ×

(10 + 100 + 1,000) = 2,220: about 500 times fewer. Space is also $O(b^{d/2})$, since one frontier must stay in memory for the meeting test. With BFS on both sides the method is complete, and optimal when all steps cost the same. With unequal costs (UCS on both sides), the first meeting is not always the cheapest path: keep going until the smallest g in the forward frontier plus the smallest g in the backward frontier is at least the cost of the best path found so far.

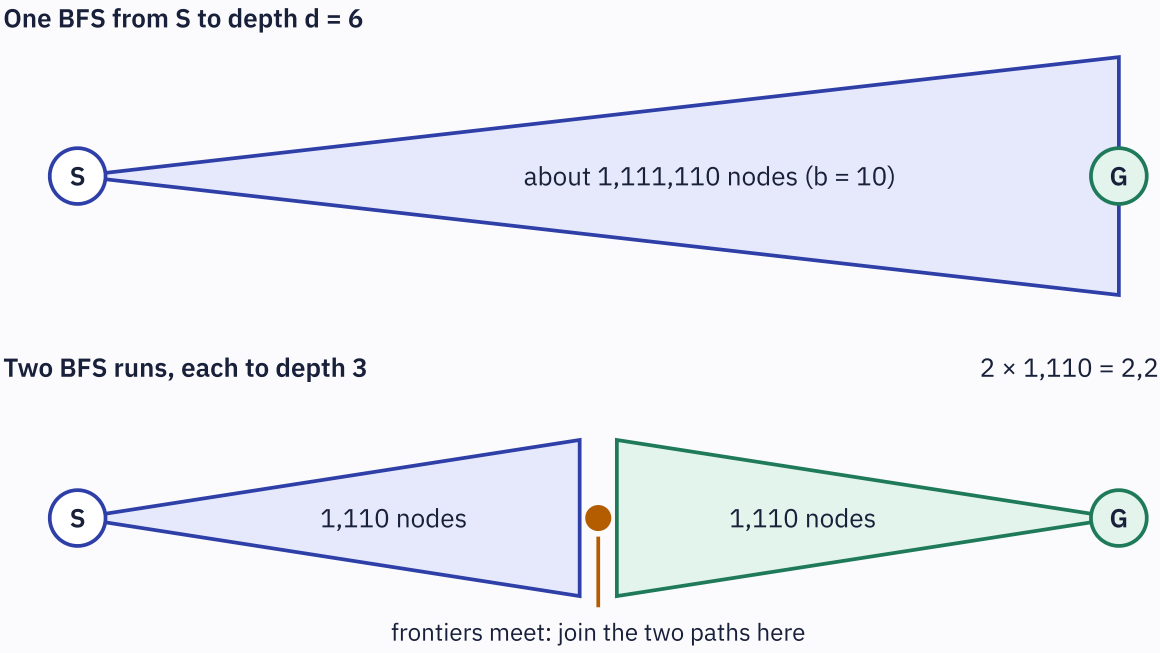


Figure 2.4.1 Top: one breadth-first tree must widen to depth 6 before it touches G. Bottom: a forward tree from S (blue) and a backward tree from G (green) each stop at depth 3, where they meet. Not to scale: the real ratio of node counts is about 500 to 1.

WORKED EXAMPLE

Question. On the delivery map of Figure 2.2.1, run bidirectional BFS between S and G, alternating one whole level at a time and taking neighbours alphabetically. Where do the searches meet, and what route results?

1. Forward, level 1: expand S, giving A, B, F. Backward set is {G}: no meeting.
2. Backward, level 1: expand G (its predecessors are its neighbours, since roads run both ways), giving D, E. Neither is in {S, A, B, F}: no meeting.
3. Forward, level 2: expand A, giving C and D. D is in the backward frontier: **the searches meet at D**.
4. Join S → A → D with D → G: route **S–A–D–G**, 3 roads, 2 + 9 + 6 = **17 minutes**, found after expanding 3 nodes (S, G, A) instead of the 6 that one-way BFS needed.

Fewest roads, but not the quickest (UCS found 9 minutes): bidirectional BFS is still blind to cost.

EXAM TRAPS

- The saving is in the exponent: $2b^{d/2}$ is not "half of b^d". For b = 10, d = 6 it is 2,000 against 1,000,000.
- It needs an explicit goal state and a way to compute predecessors. Say this when asked about its limitations.
- With step costs, the first meeting point need not give the cheapest path.

CHECK YOURSELF

– REMEMBER **State the two conditions a problem must meet for bidirectional search.** 2 marks

The goal is a single explicit state, and the predecessors of a state can be computed.

– UNDERSTAND **Why must at least one frontier be kept in memory?** 2 marks

Every new state of one search is checked against the other search's states. That check needs the other frontier to be stored, so space is $O(b^{d/2})$.

– APPLY **With b = 6 and d = 8, compare the dominant terms for one-way and two-way BFS.** 2 marks

One-way: 6^8 = 1,679,616. Two-way: 2×6^4 = 2,592. Ratio about **648**.

– APPLY **In the worked example, why did the backward search use G's neighbours as predecessors?** 1 mark

The roads are two-way, so every road into G can be driven out of G: predecessors equal neighbours.

– ANALYZE **Is bidirectional search a good way to find a checkmate in chess? Compare it with the 8-puzzle.** 4 marks

No for chess: "checkmate" is a test met by a huge number of positions, not one state, and generating the legal positions that lead to a given position is hard. The 8-puzzle suits it: one goal state, and every move can be undone, so predecessors are the same as successors.

BEYOND THE SYLLABUS

Not for the exam. Bidirectional Dijkstra is standard in road routing and is combined with preprocessing such as contraction hierarchies. Cryptography uses the same idea in the meet-in-the-middle attack, which is why double DES is much weaker than its key length suggests. Search next: "bidirectional Dijkstra", "MM bidirectional heuristic search", "meet-in-the-middle attack".

2.5 Heuristic functions; greedy best-first and A* search

A **heuristic function** h(n) estimates the cost of the cheapest path from node n to a goal, such as the straight-line distance to the destination. **Informed search** uses it to expand the most promising node first. **Greedy best-first search** trusts h alone; **A*** adds the cost already paid, f(n) = g(n) + h(n), and finds the cheapest path whenever h never overestimates. A* is the standard path finder in games, robots and maps.

You need: UCS and the delivery map from 2.2, and the strategy comparison of 2.3.

Heuristics and best-first search

A heuristic is problem knowledge written as a number: h(n) ≥ 0, and h = 0 at a goal. **Best-first search** is graph search with a priority queue ordered by an evaluation function f(n): f = g gives UCS, f = h greedy best-first, and f = g + h A*. In Figure 2.5.1, h is the rider's time along a straight line at top speed, never more than the real time.

Greedy best-first search

Greedy search expands the node that looks closest to the goal. With a good h it is fast, but it ignores the cost already paid, so it is **not optimal**. As tree search it is not complete (it can swing between two states that both look close); as graph search it is complete on a finite space. Worst-case time and space are $O(b^m)$.

A* search

A* orders the frontier by f(n) = g(n) + h(n), the estimated cost of the cheapest solution through n. Two properties of h decide whether A* is optimal. Let h*(n) be the true cheapest cost from n to a goal.

- **Admissible**: h(n) ≤ h*(n) for every n; h never overestimates.
- **Consistent** (monotone): h(n) ≤ c(n, n′) + h(n′) for every successor n′, a triangle inequality. Every consistent h is admissible, and with a consistent h the value of f never decreases along a path: f(n′) = g(n) + c(n, n′) + h(n′) ≥ g(n) + h(n) = f(n).

Optimality. Tree-search A* is optimal when h is admissible; graph-search A* is optimal when h is consistent. The argument: let C* be the optimal cost and G_2 a worse goal in the frontier, so $f(G_2) = g(G_2) > C^*$. Some node n on the optimal path is also in the frontier, and admissibility gives $f(n) \le C^* < f(G_2)$, so n, and in turn the whole optimal path, is expanded before G_2. A* expands every node with f(n) < C* and none with f(n) > C*; no other optimal algorithm with the same h is guaranteed to expand fewer. Its weakness is memory: it keeps every generated node.

In code, A* is the Python UCS program of Chapter 2.2 with priority `g[m] + H[m]`, reading path costs from `g` instead of the popped number.

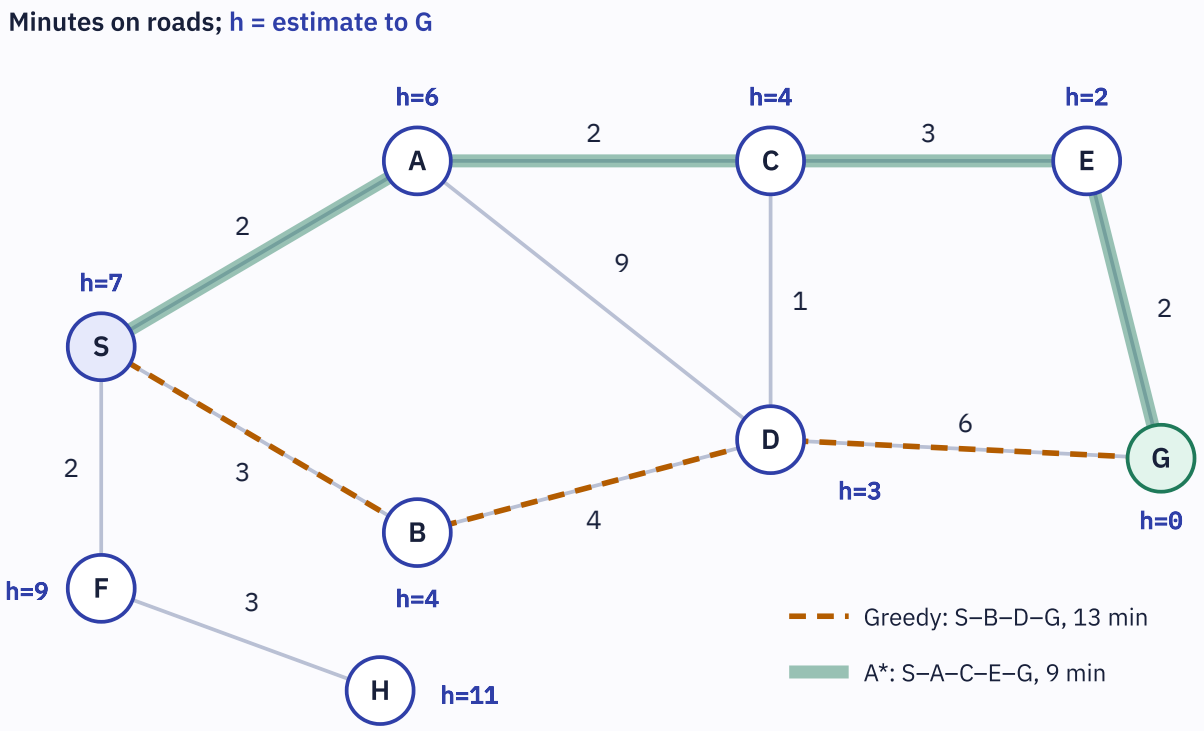


Figure 2.5.1 The map of Figure 2.2.1 with a heuristic h (blue), the straight-line time to G. Greedy search (orange dashes) follows the smallest h through B and D and pays 13 minutes. A* (green) weighs h against the time already spent and finds the 9-minute route. Notice F: h = 9 tells A* the side lane is hopeless, so F and H are never expanded.

Designing heuristics

For the 8-puzzle, two classic admissible heuristics (blank not counted):

- h_1 = the number of misplaced tiles. Each misplaced tile needs at least one move.
- h_2 = the sum of the **Manhattan distances** of the tiles from their goal squares (rows apart plus columns apart). Each tile needs at least that many moves.

Since $h_2(n) \geq h_1(n)$ for every state, h_2 **dominates** h_1: A* with h_2 never expands more nodes, apart from ties. Both come from **relaxed problems**: h_1 is the exact cost if a tile could jump anywhere, h_2 if it could slide onto an occupied square, and the exact cost of a relaxed problem is always admissible. The maximum of several admissible heuristics is admissible too. Quality is measured by the **effective branching factor** b*: if A* generates N nodes for a solution at depth d, b* solves $N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$. The closer b* is to 1, the better the heuristic.

Memory-bounded heuristic search

A* keeps every node it generates, so memory runs out first. Three versions trade time for memory:

- **IDA*** (iterative-deepening A*): IDS with an f-limit instead of a depth limit. Each round is a depth-first search that prunes nodes with f above the limit; the next limit is the smallest f pruned. Space O(bd).
- **RBFS** (recursive best-first search): a depth-first recursion that remembers the f of the best alternative path and unwinds when the current f exceeds it, storing the best f of the dropped subtree in its parent. Linear space.
- **SMA*** (simplified memory-bounded A*): A* until memory is full, then it drops the leaf with the highest f and backs that value up to the parent, so the subtree can be regrown later. Optimal if the optimal solution fits in memory.

WORKED EXAMPLE

Question 1. On Figure 2.5.1, trace greedy best-first search and A* from S to G (graph search, ties alphabetical) and compare them.

STEP	GREEDY: EXPAND (H)	FRONTIER (H)	A*: EXPAND (G + H)	FRONTIER (F)
1	S (7)	`B4 A6 F9`	S (0 + 7)	`B7 A8 F11`
2	B (4)	`D3 A6 F9`	B (3 + 4)	`A8 D10 F11`
3	D (3)	`G0 C4 A6 F9`	A (2 + 6)	`C8 D10 F11`
4	G (0): goal		C (4 + 4)	`D8 E9 F11`
5			D (5 + 3)	`E9 F11 G11`
6			E (7 + 2)	`G9 F11`
7			G (9 + 0): goal	

1. Greedy: **S–B–D–G**, 3 + 4 + 6 = **13 minutes**, 4 expansions.
2. A*: at step 4, C reaches D with g = 5 (7 via B), so f(D) drops from 10 to 8; at step 6, E reaches G with g = 9 (11 via D). Path **S–A–C–E–G**, **9 minutes**, 7 expansions.
3. UCS needed 9 expansions for this answer; A* skipped F and H because f(F) = 2 + 9 = 11 > 9.

Question 2. The 8-puzzle state is `1 2 _ / 8 5 3 / 4 7 6` and the goal is `1 2 3 / 4 5 6 / 7 8 _`. Find h_1 and h_2, choose A*'s first move with h_2, and find b* if A* generated N = 52 nodes for a depth-5 solution.

1. h_1: tiles 3, 4, 6, 7 and 8 are misplaced, so h_1 = **5**.
2. h_2: tile 3 is 1 row away, 4 is 1, 6 is 1, 7 is 1 column, and 8 is 1 row and 1 column = 2. h_2 = 1 + 1 + 1 + 1 + 2 = **6**. The true cost is 8 moves, so both are admissible and h_2 is closer.
3. Moves: blank down (tile 3 slides up) gives h_2 = 5, f = 1 + 5 = 6; blank left gives h_2 = 7, f = 8. A* expands **blank down** first, which is on an optimal path (7 moves remain).
4. b*: try 1.9: 1 + 1.9 + 3.61 + 6.86 + 13.03 + 24.76 = 51.2 < 53. Try 1.92: 53.4 > 53. So **b* ≈ 1.92**.

EXAM TRAPS

- Greedy best-first is not A*: it ignores g and is not optimal (13 minutes against 9 here).
- A* tests the goal when it is chosen, like UCS. Stopping when G first enters the frontier (f = 11 at step 5) gives the wrong route.
- Admissible alone is enough for tree-search A*; graph-search A* needs consistency. h = 0 is admissible and turns A* into UCS.
- Do not count the blank in h_1 or h_2.
- Favourite question: "Explain A*, show it is optimal with an admissible heuristic, and trace it on the given graph."

CHECK YOURSELF

– REMEMBER **Define an admissible heuristic.** 1 mark

One that never overestimates: h(n) ≤ h*(n) for every node n, where h*(n) is the true cheapest cost to a goal.

– REMEMBER **Write the evaluation function of UCS, greedy best-first and A*.** 2 marks

UCS f = g(n); greedy f = h(n); A* f = g(n) + h(n).

– UNDERSTAND **Why does h_2 dominate h_1 for the 8-puzzle, and why does it matter?** 2 marks

A misplaced tile adds 1 to h_1 but at least 1 to h_2, so $h_2 \geq h_1$ everywhere. Both are admissible, so the larger one is closer to the truth and A* with h_2 expands no more nodes.

– APPLY **Find h_1 and h_2 for the 8-puzzle state `1 2 3 / 4 _ 8 / 7 6 5` with the goal above.** 2 marks

Misplaced: 8, 6, 5, so h_1 = **3**. Distances: 8 is 1 row and 1 column away (2), 6 is 2, 5 is 2, so h_2 = **6**, which here equals the true cost.

– APPLY **In Figure 2.5.1, set h(E) = 6 instead of 2. Is h admissible? What does A* return?** 3 marks

The true cost from E is 2, so h(E) = 6 overestimates: **not admissible**. Now f(E) = 7 + 6 = 13, so G via D (f = 11) is chosen first: A* returns **S–A–C–D–G, 11 minutes**, not the optimal 9.

– ANALYZE **A robot moves on a grid, 4 directions, cost 1 per step. Compare straight-line and Manhattan distance as heuristics. What changes if diagonal moves of cost √2 are allowed?** 4 marks

With 4 directions both are admissible and Manhattan ≥ straight-line, so Manhattan dominates (it is exact on an empty grid): use it. With diagonals, one step costing √2 ≈ 1.41 can cut Manhattan distance by 2, so Manhattan overestimates and is not admissible; use straight-line or "octile" distance.

BEYOND THE SYLLABUS

Not for the exam. A* was published in 1968 by Hart, Nilsson and Raphael for the Shakey robot project. Game engines run it on grids and navigation meshes with speed-ups such as jump point search; robots that replan as they find obstacles use D* Lite. Pattern databases store exact costs of sub-problems as strong admissible heuristics, and current research trains neural networks to learn heuristics. Search next: "pattern databases", "D* Lite", "jump point search".

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
DFS / DLS	LIFO frontier; time $O(b^m)$, space $O(bm)$; DLS limit ℓ returns solution, failure or cutoff	2.3
IDS	$N = d \cdot b + (d-1) \cdot b^2 + \ldots + 1 \cdot b^d$; time $O(b^d)$, space $O(bd)$	2.3
Bidirectional	two searches meet in the middle: $2b^{d/2}$ instead of b^d	2.4
Greedy best-first / A*	$f(n) = h(n)$ / $f(n) = g(n) + h(n)$	2.5
Admissible / consistent	$h(n) \le h^*(n)$ / $h(n) \le c(n, n') + h(n')$	2.5
Effective branching factor	$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$	2.5

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 iterative deepening search with an example tree. Derive the number of nodes it generates and show that it is complete, optimal for equal costs and needs O(bd) memory.** 8 marks

DLS with $\ell = 0, 1, 2, \ldots$; trace on Figure 2.3.1 (visits 1 + 3 + 7 = 11); $N(IDS) = d \cdot b + (d - 1) \cdot b^2 + \ldots + b^d$; b = 10, d = 5 gives 123,450 against 111,110 for BFS; properties and reasons.

– REMEMBER **Define admissible and consistent heuristics. Show that a consistent heuristic makes f non-decreasing along a path, and state when A* is optimal.** 8 marks

$h \le h^*$; $h(n) \le c(n, n') + h(n')$; $f(n') = g(n) + c + h(n') \ge g(n) + h(n) = f(n)$; consistent implies admissible; tree-search A* optimal with admissible h, graph-search A* with consistent h; the G_2 argument.

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 **Draw the A* search tree with h_2 (Manhattan distance) for the 8-puzzle start `1 2 3 / _ 4 6 / 7 5 8` and goal `1 2 3 / 4 5 6 / 7 8 _`, writing g + h at every node.** 10 marks

Root f = 0 + 3. Children: blank right f = 1 + 2 = 3, blank up 1 + 4 = 5, blank down 1 + 4 = 5. Expand blank right: blank down 2 + 1 = 3; blank up 2 + 3 = 5; blank right 2 + 3 = 5. Expand: blank right gives the goal, f = 3 + 0 = 3. Solution 3 moves (4 left, 5 up, 8 left).

Analyze and evaluate

– EVALUATE **"A* is the best search algorithm." Evaluate this claim.** 10 marks

For: optimal with an admissible or consistent h, optimally efficient, beats UCS (7 against 9 expansions on the map). Against: exponential memory (use IDA*, RBFS, SMA*), needs a good h (a bad one wastes work; an overestimate loses optimality, as with h(E) = 6), and for optimisation problems where the path does not matter, local search or GAs are the right tools.

Glossary (terms used in this lesson)

A* search	Best-first search on f(n) = g(n) + h(n); optimal with an admissible (tree) or consistent (graph) heuristic.
Admissible heuristic	A heuristic that never overestimates: h(n) ≤ h*(n).
Bidirectional search	Forward search from the start and backward search from the goal, stopping when they meet.
Depth-first search	Expands the deepest node first, using a LIFO stack or recursion.
Depth-limited search	DFS that treats nodes at depth ℓ as leaves; returns a solution, failure or cutoff.
Explored set	The states already expanded (the closed list).
Frontier	Nodes generated but not yet expanded (the open list or fringe).
Greedy best-first search	Best-first search on f(n) = h(n); fast but not optimal.
Heuristic function	h(n), an estimate of the cheapest cost from n to a goal.
Iterative deepening search	DLS run with limits 0, 1, 2, ... until a solution is found.
Node	A search-tree record: state, parent, action, path cost and depth.
Relaxed problem	The problem with some rules removed; its exact cost is an admissible heuristic.

Sources and further reading

Standard textbooks (all from memory; check edition and chapter against your library copy):

- 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. (from memory; check)

- 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). (from memory; check)
- Saroj Kaushik, *Artificial Intelligence*, Cengage Learning: the chapter on state-space search and control strategies, including iterative-deepening A*. (from memory; check)
- George F. Luger, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Pearson: state-space search, data-driven and goal-driven search, heuristic search. (from memory; check)
- Nils J. Nilsson, *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann: heuristic search and A*. (from memory; check)
- David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley; Melanie Mitchell, *An Introduction to Genetic Algorithms*. (from memory; check)
- 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). (from memory; check)

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 (from memory; not opened)
- Red Blob Games, "Introduction to the A* algorithm", interactive BFS, Dijkstra and A* on grids: redblobgames.com (from memory; not opened)

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.