

Why Search?

- **Successful**
 - Success in game playing programs based on search.
 - Many other AI problems can be successfully solved by search.
- **Practical**
 - Many problems don't have a simple algorithmic solution. Casting these problems as search problems is often the easiest way of solving them.
 - Often specialized algorithms cannot be easily modified to take advantage of extra knowledge. Heuristics provide search provides a natural way of utilizing extra knowledge.
- Some critical aspects of intelligent behaviour, e.g., planning, can be naturally cast as search.

→ Components of Search

- Statespace
- Successor function, cost
- Start state, goal test
- Solution

Representing a problem: The Formalism

To formulate a problem as a search problem we need the following components.

1. Formulate a state space over which to search.
2. Formulate actions that allow one to move between different states.
3. Identify the initial state that best represents your current state and the desired condition one wants to achieve.

Formulate various heuristics to help guide the search process (initially we do without). → For informed search

A search problem consists of:

• A state space

• A successor function (with actions, costs)

• A start state and a goal test

• A **solution** is a sequence of actions (a plan) which transforms the start state to a goal state

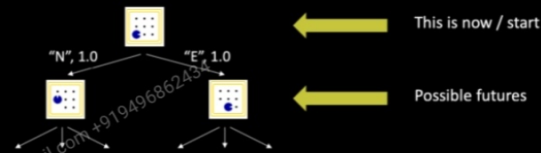
State Space Graphs

- State space graph: A mathematical representation of a search problem
 - Nodes are (abstracted) world configurations
 - Arcs represent successors (action results)
 - The goal test is a set of goal nodes (maybe only one)

In a state space graph, each state occurs only once!



Search Trees



- A search tree:
 - A "what if" tree of plans and their outcomes
 - The start state is the root node
 - Children correspond to successors
 - Nodes show states
 - For most problems, we can never actually build the whole tree

The set of all leaf nodes available for expansion at any given point is called the **frontier** or **open list**.

→ AKA Fringe

Fringe → A data structure used to store all possible states that can be explored.

Defines the order of selecting which node to expand
 Search algorithm differs in this queuing function

Generic Search Algorithm : ☆

1. Initialize search algorithm with initial state of the problem
2. **Repeat**
 1. If no candidate nodes can be expanded, **return failure**
 2. Choose leaf node for expansion, according to search strategy
 3. If node contains a goal state, **return solution**
 4. Otherwise, expand the node, by applying legal operators to the state within the node. Add resulting nodes to the tree

☆ expand does not goal check children

Uninformed techniques :

- Systematically search complete graph
- Unguided
- Also known as brute force, naïve, or blind

Fixed rule

Informed methods :

- Use problem specific information to guide search in promising directions

- Breadth first search
- Uniform Cost "
- Depth first "
- Depth Limited "
- Iterative Deepening "

→ A*

Properties of Search :

- **Completeness:** Will the search always find a solution if a solution exists?
- **Optimality:** Will the search always find the least cost solution? (when actions have costs)
- **Time complexity:** What is the maximum number of nodes than can be expanded or generated?
- **Space complexity:** What is the maximum number of nodes that have to be stored in memory?

In a search algorithm like Depth First Search, Depth Limited Search, or Breadth First Search, a node is selected from the fringe and then expanded, and then without checking whether the children of these nodes are the goal node, regardless these nodes are added to the fringe, and then the next node in the fringe is selected based on some strategy. Before expanding, we check whether this is a goal node using the goal test, and if it is not a goal node, then we expand. Again, after expansion, the children are not checked for whether they are goal node, they are just added to the fringe. Is this how it works? It seems a little inefficient. Isn't it better to also check for the children of the node that is under expansion?



Great question – this is a subtle but important part of how search algorithms are defined in AI textbooks (like Russell & Norvig).

Let me clarify exactly how it works, why it may seem inefficient, and why the standard design is actually intentional.

Short Answer

Yes, your understanding is correct:

In DFS, DLS, BFS, UCS, A*, etc., we test for the goal only when a node is *selected from the fringe for expansion*, not when it is *generated*.

So the typical loop is:

1. Pick a node from fringe (according to strategy)
2. Goal-test this node
3. If not goal → expand
4. Add children to fringe **without goal-testing them**

This is the standard algorithm.

Why do we not goal-test newly generated children?

Reason 1 – Correctness for “tree search” vs “graph search”

In *tree search*, goal testing children is technically safe.

But in *graph search*, it causes a subtle correctness problem:

- You might find a goal in the “wrong path”
- You may prematurely return a suboptimal solution

Example:

In UCS or A*, you **MUST** test the goal only when it is **popped** from the priority queue, not when generated.

This ensures optimality.

Reason 2 – Separation of concerns in the algorithm

The AI search framework defines:

- **Expand** = generate successors
- **Goal-test** = check selected node

This clean separation ensures the algorithm behaves consistently regardless of:

- search strategy
- priority functions
- pruning
- repeated-state checks

Reason 3 – Optimality (very important)

Consider **Uniform Cost Search** or **A***.

If you test children when generated, then the *first time you generate a goal*, you may accept a **non-optimal path**.

But these algorithms rely on the property:



The first time the goal node is *removed from the fringe*, it is the optimal solution.

This is mathematically guaranteed due to ordering of the priority queue.

Goal-testing at generation breaks this guarantee.

Reason 4 – Efficiency: not as wasteful as it seems

It appears inefficient to postpone goal testing, but:

- You will test a child very soon anyway – as soon as it reaches top of fringe.
- In BFS/DFS, the delay is just one loop.
- In UCS/A*, delaying is necessary for correctness.

Also, goal-tests are usually extremely fast compared to:

- state generation
- transitions
- maintaining fringe (heap operations)
- repeated-state checks

So the benefit is negligible.

But is it ever useful to test children immediately?

Yes – in **uninformed search** where optimality does not matter:

- DFS
- Depth-Limited Search
- Iterative Deepening (tree version)

Here, testing children early (“on generation”) is safe, and some implementations do this for a minor constant-factor speed-up.

BUT AI textbooks still teach the general framework where:

Goal test happens only when removing a node from the fringe.

They do this for consistency across all algorithms.

Summary Table

Search	Should goal-test generated children?	Why
DFS	Allowed, but not standard	Doesn't affect correctness
DLS	Allowed	Same as DFS
BFS	Allowed	Still gets optimal solution for unit-cost
UCS	✗ Not allowed	Breaks optimality
A*	✗ Not allowed	Breaks optimality guarantees
Graph search	✗ Often problematic	May miss better paths

Final Conclusion

Yes, the standard approach is exactly how you described:

- Check goal only when the node is **popped** for expansion
- Not when it is **generated**

It may look slightly inefficient in DFS/BFS, but it is required for:

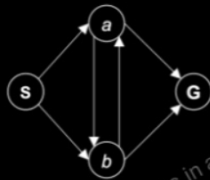
- correctness of UCS, A*, IDA*
- consistency of generic search framework
- avoiding premature termination
- simpler uniform algorithm structure

Stating a Problem as a Search Problem



- State space S
- Successor function: $x \in S \rightarrow \text{SUCCESSORS}(x) \in 2^S$
- Initial state s_0
- Goal test: $x \in S \rightarrow \text{GOAL?}(x) = T \text{ or } F$
- Arc cost

Consider this 4-state graph:



How big is its search tree (from S)?



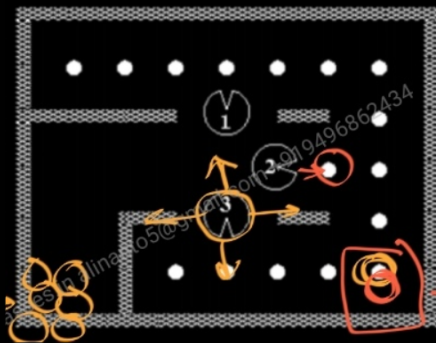
Cycle in the graph \Rightarrow Infinitely big tree
 (if states are allowed to be revisited) \Rightarrow even though statespace is finite

Depth = # edges from root to a node
 Depth of root = 0

Branching factor = Max no. of child nodes possible from any state node

Q

Pacman is trying eat all the dots, but he now has the help of his family! There are initially k dots, at positions (f_1, \dots, f_k) . There are also n Pac-People, at positions (p_1, \dots, p_n) ; initially, all the Pac-People start in the bottom left corner of the maze. Consider a search problem in which all Pac-People move simultaneously; that is, in each step each Pac-Person moves into some adjacent position (N, S, E, or W, no STOP). Note that any number of Pac-People may occupy the same position.



The state space consists of the following: a k -tuple of boolean variables E , where $E_i = 1$ if the i th food has been eaten and 0 otherwise, and the n -tuple of Pac-People positions $P = (p_1, \dots, p_n)$. We assign a cost of 2 for a movement of Pac-People which does not result in a food being eaten, and a cost of 1 for a movement which does.

a) Upper Bound for 8×8 grid \rightarrow size of statespace

$$2^k \rightarrow \text{dot configurations}$$

$$\text{each pacman can be in } 8 \times 8 \text{ different position}$$

$$= (8 \times 8)^n \text{ — pacman configurations}$$

$$\text{Total} = 2^k (8 \times 8)^n = \text{size of statespace}$$

b) Goal test? \rightarrow k tuple for dots all 1.

c) Maximum branching factor of the successor function in a general grid?

\rightarrow each pacman can turn left right up or down
 \rightarrow 4 different action for each pacman
 \rightarrow $4 \times 4 \times \dots \times 4$ — n times for n pacoman = 4^n branches possible

The maximum branching factor of the successor function in a general grid will be 4^n , since there are n Pac-People and each of them can go in one of four directions, N, S, E, or W. (Note that the minimum may be lower, since different combinations of directions may result in the same state, for example if a Pac-Person is in the upper-left corner then going left and going up results in the same outcome for that Pac-Person).

Q Pacman and Ms. Pacman are lost in an $N \times N$ maze and would like to meet; they don't care where. In each time step, both simultaneously move in one of the following directions: {NORTH, SOUTH, EAST, WEST, STOP}. They do not alternate turns. You must devise a plan which positions them together, somewhere, in as few time steps as possible. Passing each other does not count as meeting; they must occupy the same square at the same time.

a) Define the state space of the search problem. \rightarrow
 $P_1 \text{ — } \langle x, y \rangle$
 $P_2 \text{ — } \langle x, y \rangle$

b) size of state space = $N \times N \times N \times N = N^4$

c) Branching factor = $5^2 = 25$

d) Goal test $P_1 = P_2$

What is the main difference between a state space graph and a search tree?

- A) A search tree has cycles, while a state space graph does not
- B) A state space graph represents all unique states, while a search tree may repeat states
- C) A search tree is used only for BFS, while a state space graph is used only for DFS
- D) A search tree represents states without edges, while a state space graph contains edges

state space graph can have cycles even though each state node does not repeat
 search tree has repeating states — but no cycles
 Both has edges for action

Which of the following is NOT a component of a state space?

- A) Set of initial states
- B) Set of goal states
- C) Operators or actions
- D) Decision tree nodes

False

In a state space graph, nodes represent:

- A) Actions or operators
- B) States of the problem
- C) Cost of transitions
- D) Heuristics for searching

The branching factor in a search problem is defined as:

- A) The maximum depth of the search tree.
- B) The number of child nodes generated from a single state.
- C) The total number of nodes in the search tree.
- D) The cost associated with moving from one state to another.

If a search tree has depth d and a branching factor b , what is the maximum number of nodes in the tree?

- A) b^d
- B) $b^{d+1} - 1$
- C) $(b^d - 1)/(b - 1)$
- D) $b \cdot d$

Answer: C) $(b^{d+1} - 1)/(b - 1)$

$$1 + b + b^2 + b^3 \dots + b^d = \frac{1 \cdot (b^{d+1} - 1)}{b - 1}$$

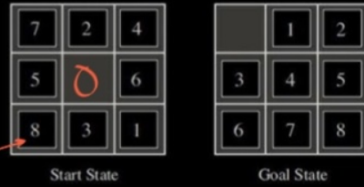
☆☆☆☆

Summary Table

Concept	Definition	Root Value	Leaf Value
Depth	Edges from root to node	0	Varies
Level	Depth + 1	1	Varies
Height (of node)	Longest downward path to a leaf	Height of tree	0
Height (of tree)	Height of root / max depth	Tree height	—

In a puzzle-solving problem with n tiles and one blank space, how many states are there in the state space?

- A) n^n
- B) $(n + 1)!$
- C) $(n - 1)!$
- D) $n!$



9 distinct objects
 9 distinct slots = 9! ways
 $(n+1)!$

If a search tree has a branching factor of 3 and a depth of 4, how many nodes are there at the fourth level?

- A) 3^4
- B) 4^3
- C) $3 \cdot 4$
- D) $4 \cdot 3$

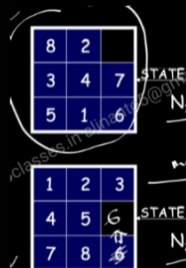
depth
 0 — 1
 1 — 3
 2 — 3^2
 3
 :
 n — 3^n

Depth First Search

Fringe — set of all search nodes that haven't been expanded yet

Fringe is implemented as priority Datastructure
 → Ordering defines search strategy.

- Blind (or un-informed) strategies do not exploit state descriptions to order FRINGE. They only exploit the positions of the nodes in the search tree
- Heuristic (or informed) strategies exploit state descriptions to order FRINGE (the most "promising" nodes are placed at the beginning of FRINGE)



For a heuristic strategy counting the number of misplaced tiles, N_1 is more promising than N_2

for blind strategies both N_1 & N_2 are equal

The set of all leaf nodes available for expansion at any given point is called the frontier or open list.

≡ Fringe

Some search problems, such as the (n^2-1) - puzzle, are NP-hard

One can't expect to solve all instances of such problems in less than exponential time (in n)

One may still strive to solve each instance as efficiently as possible

This is the purpose of the search strategy

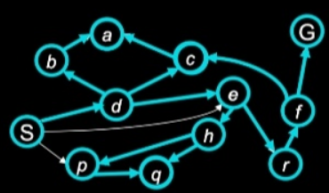


- Uninformed Search Methods**
- Depth-First Search
 - Breadth-First Search
 - Uniform-Cost Search
- Informed Search Methods**
- Best First / Greedy Search
 - A* Search
- Handwritten notes: Depth limited Search, Iterative Deepening, Bidirectional search, ID.

Depth first search

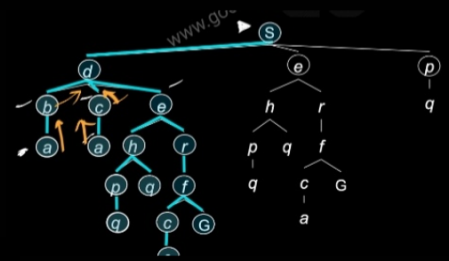
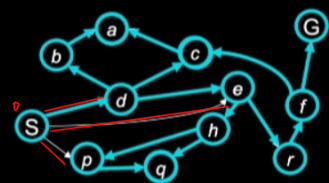
Expand deepest node first
Fringe is a LIFO stack

Strategy: expand a deepest node first
Implementation: Fringe is a LIFO stack



General idea:
- Expand the most recently expanded node if it has successors
- Otherwise backup to the previous node on the current path

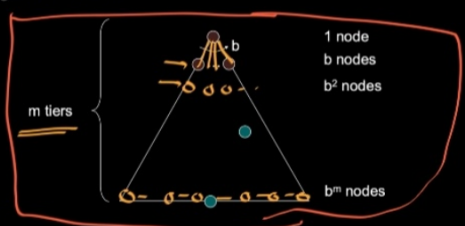
Backtracking



Fringe
S
p
e
d
h
r
q
f
c
G

Search Algorithm Properties

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?
- Cartoon of search tree:
 - b is the branching factor
 - m is the maximum depth
 - solutions at various depths



Number of nodes in entire tree?
 $1 + b + b^2 + \dots + b^m = O(b^{m+1})$

Branching Factor: no. of childs a Node can have

Time Complexity

Nodes for max depth M & branching factor B
 $= 1 + b + b^2 + \dots + b^m = \frac{b^{m+1} - 1}{b - 1} = O(b^m)$

- if finite m then time complexity $= O(b^m)$
- Since worst case time \propto # max nodes visited

Space Complexity

Worst case space \propto Max length of fringe



If a node is queued its parent and siblings have been queued, and its parent dequeued.

$S_{dfs} = O(b \cdot m)$

$(b-1)$ nodes in stack for any depth
for m total depth $= b + (m-1)(b-1)$
 $= O(m \cdot b)$

Complete: Only if m finite (ie prevent cycles)

\rightarrow cycle in search graph \Rightarrow infinite search tree

Finite depth \Rightarrow even if goal is not in tree we will visit all nodes

\Rightarrow If goal in tree, DFS will find it. (given finite m)

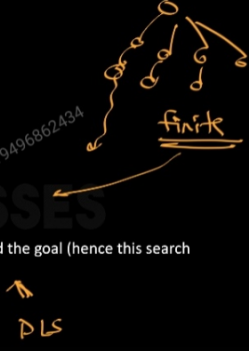
Optimal: NO \rightarrow leftmost solution

Search Method	Worst Time	Worst Space	Shortest Path?	Guaranteed to find path?
Depth-first	$\sim b^m$	b^*m	No	Yes for finite graph

Depth-Limited Search

Depth-Limited Search (DLS)

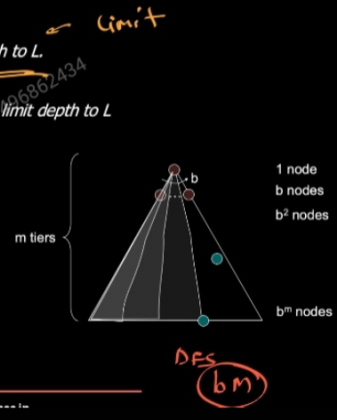
It is simply DFS with a depth bound.
 - Searching is not permitted beyond the depth bound.
 - Works well if we know what the depth of the solution is.
 - Termination is guaranteed.
 - If the solution is beneath the depth bound, the search cannot find the goal (hence this search algorithm is incomplete).



Main idea: Expand node at the deepest level, but limit depth to L.

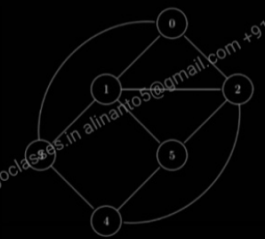
Implementation: Enqueue nodes in LIFO (last-in, first-out) order. But limit depth to L.

- Complete? Yes if there is a goal state at a depth less than L.
- Optimal? No
- Time Complexity: $O(b^L)$, where L is the cutoff.
- Space Complexity: $O(bL)$, where L is the cutoff.



1. Which sequence corresponds to that of depth first search for the graph given below. The search starts at vertex 0 and lexicographic ordering is assumed for the edges emanating from each vertex.

left to right

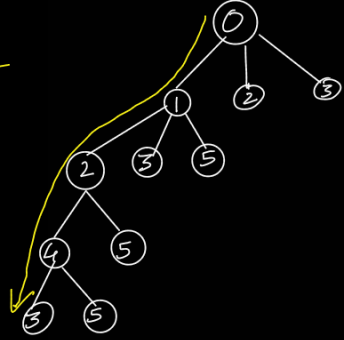


- A. 0 1 2 4 3 5
- B. 0 1 2 5 4 3
- C. 0 1 2 3 4 5
- D. 0 1 3 4 2 5

Answer: A. 0 1 2 4 3 5

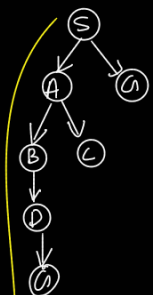
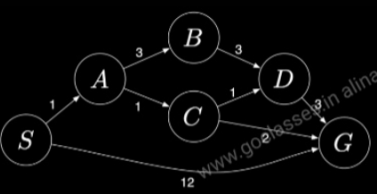
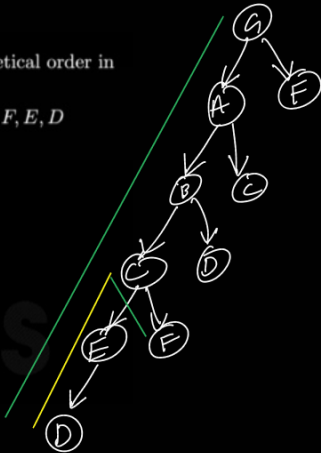
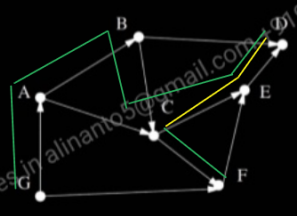
Explanation:
 A correct sequence of DFS traversal is 0 1 2 4 3 5
 There is no edge between 2 and 3 and also 5 which is connected to 2 is unvisited.

Fringe
 0
 3
 2
 1
 5
 4
 3
 2
 5
 4



For the graph shown in Figure 1, suppose vertices A, B, C, D, E, F, G are stored in alphabetical order in this graph. What is DFS order of traversing this graph starting from vertex G?

- A. G, A, B, C, D, E, F
- B. G, A, B, D, C, E, F
- C. G, A, B, C, E, D, F**
- D. G, A, B, C, F, E, D



S-A-B-D-G

Answer the following questions about the search problem shown above. Break any ties alphabetically. For the questions that ask for a path, please give your answers in the form 'S - A - D - G.'
 What path would depth-first graph search return for this search problem? *S is start state*

But note the question:

How many times will a state at level $d - 4$ have its successors generated?



A node generates its successors only when the depth limit is strictly greater than its depth

- At limit $d - 4$: node is reached but **no successor generation** (limit reached).
- At limits $d - 3, d - 2, d - 1, d$: successors **are generated**.

Depth-first search uses $O(bd)$ space, where b is the branching factor of the search tree and d is the depth of the shallowest goal node.
T/F?

if shallowest is replaced by deepest - still false

False. Not necessarily. DFS uses $O(bm)$, where m is the depth of the search space—not $O(bd)$, where d is the depth of the shallowest goal node. DFS can "miss" a shallow goal node and continue searching deep into the search space. If the depth is infinite, DFS might not even finish! In fact, it's not a complete search.



[1 pt] What is the time complexity (maximum number of nodes explored in the worst case) for depth-first tree search?

- $O(b^m)$
 $O(bm)$
 $O(m)$
 $O(b)$

In the worst case, we may need to explore the entire tree before finding a solution. The tree contains roughly $1 + b + b^2 + \dots + b^m = O(b^{m+1})$ search nodes.

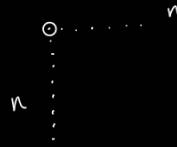
A Rubik's cube has about 4.3×10^{19} possible configurations, but any configuration can be solved in 20 moves or less. We pose the problem of solving a Rubik's cube as a search problem, where the states are the possible configurations, and there is an edge between two states if we can get from one state to another in a single move. Thus, we have 4.3×10^{19} states. Each edge has cost 1. Note that the state space graph does contain cycles. Since we can make 27 moves from each state, the branching factor is 27. Since any configuration can be solved in 20 moves or less, we have $h^*(n) \leq 20$.

For each of the following searches, estimate the approximate number of states expanded. Mark the option that is closest to the number of states expanded by the search. Assume that the shortest solution for our start state takes exactly 20 moves. Note that 27^{20} is much larger than 4.3×10^{19} .

- (i) [2 pts] DFS Tree Search
- Best Case: 20 4.3×10^{19} 27^{20} ∞ (never finishes)
- Worst Case: 20 4.3×10^{19} 27^{20} ∞ (never finishes)

Side Quest

Cost of Gauss Elimination



for 0 zero under first pivot
 n multiplication with a cost \rightarrow Rest as
 n addition with another rows \rightarrow same operation
 $\approx O(n)$
 for $(n-1)$ rows $\Rightarrow (n-1)n$ operations
 $(n-1)n + (n-2)(n-1) + (n-3)(n-2) + \dots + 2 \times 1$
 $= O(n^2 + n + n^2 + \dots + n^2)$
 $O(n(n^2)) = O(n^3)$

DFS is complete when the graph being searched is finite. True/False

True/False

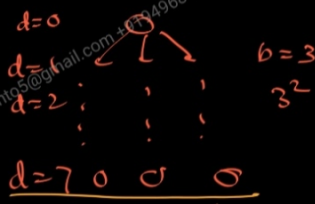
finite cyclic graph \equiv infinite tree

If you know for sure that the single goal state for a problem is at depth 19, and the search space has a large branching factor, then the best search algorithm is
 A) Depth-first search
 B) Depth-limited search (limit set to 19)

If a problem's solution lies at depth 9 in a search tree with branching factor $b = 3$, what would be the number of nodes expanded by Depth-Limited Search with $l = 7$?

- A. 3^7
 B. $\frac{3^8-1}{2}$
 C. $\frac{3^7-1}{2}$
 D. $\frac{3^9-1}{2}$

$1 + 3 + 3^2 + \dots + 3^7 = \frac{3^8 - 1}{2}$



Which of the following statements about Depth-Limited Search is FALSE?

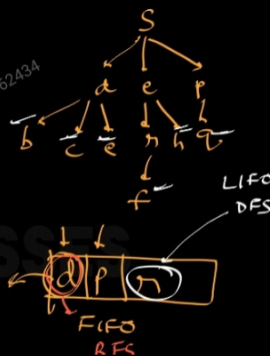
- A. It uses less memory compared to depth-first search.
 B. It is guaranteed to find a solution if the depth limit is set correctly.
 C. It can always guarantee optimality if the depth limit is sufficient.
 D. It can be incomplete if the depth limit is too low.

Breadth First Search

\rightarrow shallowest Node
 \rightarrow FIFO



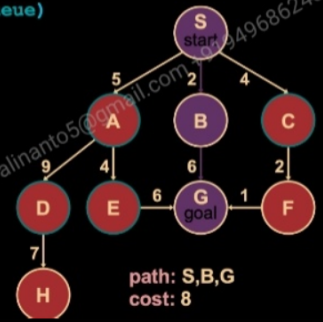
$s \rightarrow c \rightarrow h \rightarrow f$
 $s \rightarrow e \rightarrow h$
 $s \rightarrow d \rightarrow c$
 $s \rightarrow d \rightarrow c$
 $s \rightarrow d \rightarrow b$
 $s \rightarrow a \rightarrow b$
 $s \rightarrow a \rightarrow c$
 $s \rightarrow a \rightarrow d$
 $s \rightarrow a \rightarrow e$
 $s \rightarrow a \rightarrow f$
 $s \rightarrow a \rightarrow g$
 $s \rightarrow a \rightarrow h$
 $s \rightarrow a \rightarrow i$
 $s \rightarrow a \rightarrow j$
 $s \rightarrow a \rightarrow k$
 $s \rightarrow a \rightarrow l$
 $s \rightarrow a \rightarrow m$
 $s \rightarrow a \rightarrow n$
 $s \rightarrow a \rightarrow o$
 $s \rightarrow a \rightarrow p$
 $s \rightarrow a \rightarrow q$
 $s \rightarrow a \rightarrow r$
 $s \rightarrow a \rightarrow s$
 $s \rightarrow a \rightarrow t$
 $s \rightarrow a \rightarrow u$
 $s \rightarrow a \rightarrow v$
 $s \rightarrow a \rightarrow w$
 $s \rightarrow a \rightarrow x$
 $s \rightarrow a \rightarrow y$
 $s \rightarrow a \rightarrow z$



generalSearch(problem, queue)

of nodes tested: 7, expanded: 6

expnd. node	Frontier list
S	{S}
A	{A, B, C}
A	{B, C, D, E}
B	{C, D, E, G}
C	{D, E, G, F}
D	{E, G, F, H}
E	{G, F, H, G}
G	{F, H, G}



\Rightarrow remember: not goal testing when expanding
 \Rightarrow goal test only when de-fringe

Time Complexity $\rightarrow (b^s)$ b - branching factor
 s - depth of shallowest goal

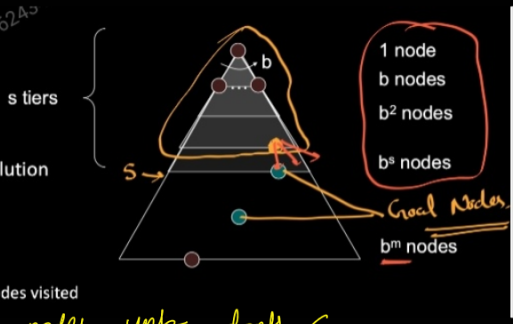
BFS : Time complexity

What nodes does BFS expand?

- Processes all nodes above shallowest solution
- Let depth of shallowest solution be s
 Search takes time $O(b^s)$

Worst case time T is proportional to number of nodes visited

\rightarrow Since we expand all nodes upto depth s
 $1 + b + b^2 + \dots + b^s = \frac{b^{s+1} - 1}{(b-1)} = O(b^s)$

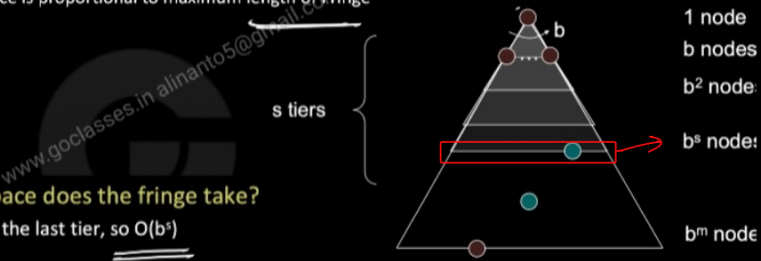


Space Complexity (b^s)

Worst case space is proportional to maximum length of Fringe

How much space does the fringe take?

- Has roughly the last tier, so $O(b^s)$



Optimal \rightarrow if all edge cost is 1 \rightarrow Yes \rightarrow No if edge cost $\neq 1$
 Complete \rightarrow if goal is in finite depth \rightarrow Yes

Uniform Cost Search

\rightarrow Expand cheapest node
 \rightarrow Priority queue by cheapest cost

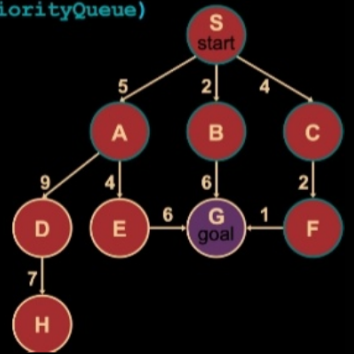
- Use a **Priority Queue** to order nodes in Frontier, sorted by path cost
- Let $g(n)$ = cost of path from start node s to current node n
- Sort nodes by increasing value of g

- Equivalent to breadth-first if step costs all equal
- Equivalent to Dijkstra's algorithm in general

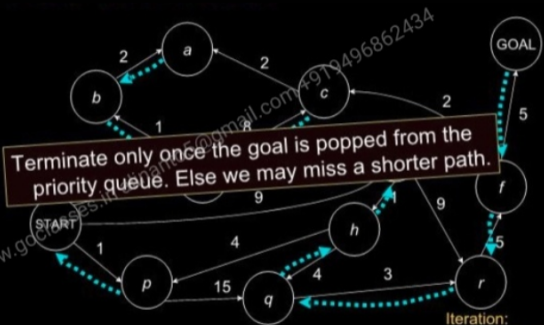
generalSearch (problem, priorityQueue)

of nodes tested: 6, expanded: 5

expnd. node	Frontier list
S	{S}
B	{B:2, C:4, A:5}
C	{A:5, F:6, G:8}
A	{F:6, G:8, E:9, D:14}
F	{G:7, G:8, E:9, D:14}
G goal	{G:8, E:9, D:14}
	no expand



☆ \rightarrow Update Node Cost if found cheaper path at any point in Algorithm



- Iteration:
1. Pop least-cost state from PQ
 2. Add successors

PQ = {}

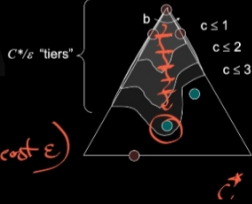
- \rightarrow Do not goal test when expanding
- \rightarrow Only goal test when de-fringing
- \rightarrow Program ends only when goal is removed from fringe for the first time not when goal is added for the first time

Time Complexity $\rightarrow b^{C^*/\epsilon}$

UCS : Time complexity

- What nodes does UCS expand?
 - Processes all nodes with cost less than cheapest solution!
 - If that solution costs C^* and arcs cost at least ϵ , then the "effective depth" is roughly C^*/ϵ
 - Takes time $O(b^{C^*/\epsilon})$ (exponential in effective depth)
- Worst case time T is proportional to number of nodes visited

Cost to reach our goal = C^*
 arc cost / edge cost $\approx \epsilon$ (at least ϵ)

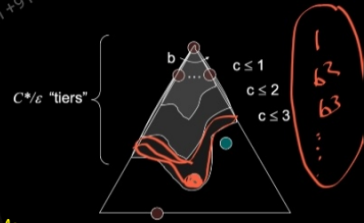


\rightarrow #nodes expanded up to level $(\frac{C^*}{\epsilon})$
 $= 1 + b + b^2 + \dots + b^{C^*/\epsilon} = O[b^{C^*/\epsilon}]$

Space Complexity $- b^{C^*/\epsilon}$

UCS : Space complexity

- Worst case space is proportional to maximum length of fringe
- How much space does the fringe take?
 - Has roughly the last tier, so $O(b^{C^*/\epsilon})$



\rightarrow We need to keep the entire C^*/ϵ depth in fringe $\Rightarrow b^{C^*/\epsilon}$ nodes

Complete \rightarrow

- solution has finite cost
- Min edge cost positive

 \Rightarrow Yes

Optimal \rightarrow Yes

Performance of search algorithms on trees

b : branching factor (assume finite) d : goal depth m : graph deg

Fringe Data Structure Strategy

FIFO	Shallowest Node
Priority Que	Cheapest Node
LIFO	Deepest Node

	Complete	optimal	time	space
Breadth-first search	Y	Y, if $\epsilon > 0$	$O(b^d)$	$O(b^d)$
Uniform-cost search ²	Y	Y	$O(b^{C^*/\epsilon})$	$O(b^{C^*/\epsilon})$
Depth-first search	N	N	$O(b^m)$	$O(bm)$

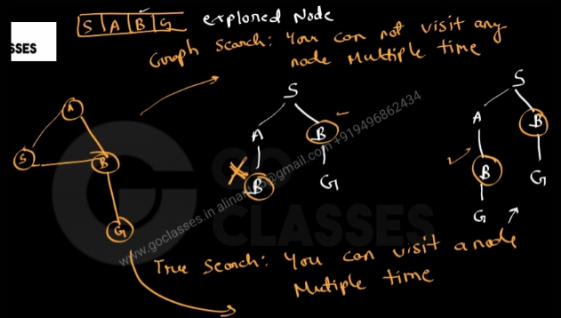
Disadvantage

- \rightarrow Not complete for < 0 edge cost
- \rightarrow Not optimal for cost of edge $\neq 1$
- \rightarrow Incomplete for infinite tree
- \rightarrow Not optimal

- edge cost constant, or positive non-decreasing in depth
- edge costs $\geq \epsilon > 0$. C^* is the best goal path cost.

Graph search v/s tree search :

- The Problem Is Always a Graph.
- It is always assumed you're dealing with a **graph**.
- The distinction lies in the **traversal pattern** that is used to search through the graph, which can be graph-shaped or tree-shaped.
- In **Graph Search** you hold a list of explored nodes, while in **Tree Search** you don't.
- Tree search can visit a state multiple times.



The set of all leaf nodes available for expansion at any given point is called the **frontier** or **open list**.

Fringe

```

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
  
```

Search generic algorithm
 → choosing defines search strategy
 → No mechanism in the algorithm to prevent visiting same nodes multiple times

```

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
  
```

graph search mechanism to prevent multiple visit
 → List of explored nodes
 → No duplicates allowed in frontier

Why is graph-search DFS complete (for problems with a finite set of states), but tree-search DFS is not?

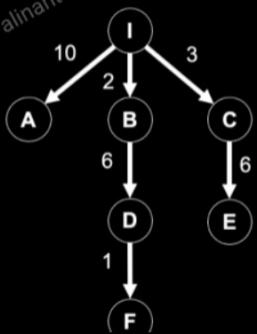
Graph Search: You can not visit a Node Multiple times.

Graph Search DFS: No chance that you visit any node multiple times ⇒ there will NOT be ∞ tree.

Graph DFS
 ⇒ Since #Nodes is finite ⇒ Tree is finite
 ⇒ Graph DFS is complete.

a) You are running UCS in the state space graph below. You just called the successor function on node D. What is the cost of node F?

b) You just expanded node C, What node will be expanded Next? → D



1. 2

2. 7

3. 8

4. 9 ✓



Q2-1: You are running DFS in the state space graph below. DFS expands nodes left to right. G is the goal state. The state space graph is infinite (the path after D does not terminate). What is the behavior of DFS?

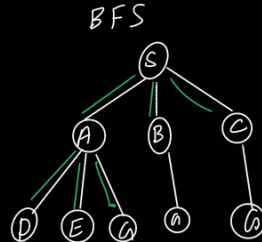
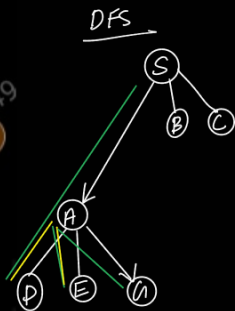
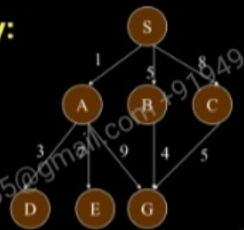


1. Get stuck in an infinite loop ✓
2. Return A
3. Return G
4. Return "failure"

S starts in goal

Nodes expanded by:

- Breadth-First Search: S A B C D E G
Solution found: S A G
- Uniform-Cost Search: S A D B C E G
Solution found: S B G (This is the only uninformed search that worries about costs.)
- Depth-First Search: S A D E G
Solution found: S A G



UCS

Exp	Fringe
S	{A-1, B-5, C-8}
A	{D-4, B-5, C-8, E-8, H-10}
D	{B-5, C-8, E-8, H-10}
B	{C-8, E-8, H-9, H-10}
C	{E-8, H-9, H-10, H-13}
E	{H-9, H-10, H-13}
H	

Expand left to right smallest to largest when tie

Q3-1: Consider the state space graph below. Goal states have bold borders. Nodes are expanded left to right when there are ties. What solution path is returned by BFS?



1. IADFH
2. IADFG
3. IAG
4. ICEG

EX	Fringe
I	A B C
A	B C D H
B	C D H
C	D H E
D	H E F
H	

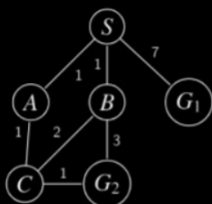
b) Same Question above in UCS →

path I-C-E-H (UCS)

EX	Fringe
I	{C-3, B-6, A-10}
C	{B-6, E-9, A-10}
B	{E-9, A-10}
E	{A-10, H-13}
A	{D-12, H-13, G-15}
D	{G-13, H-15, F-16}
H	

c) Same question in DCS →
path → I-A-D-F-H

[4] Which goal is reached and what is the total cost of the solution found for the following state-space graph when using Breadth-First Search and Uniform-Cost Search (S is the start state, G1 and G2 are the goal states, arcs are bidirectional, no repeated state checking, break any ties alphabetically)?



- A. BFS: G1 (Cost: 7), UCS: G2 (Cost: 4)
- B. BFS: G2 (Cost: 4), UCS: G1 (Cost: 7)
- C. BFS: G2 (Cost: 4), UCS: G2 (Cost: 4)
- D. BFS: G1 (Cost: 7), UCS: G2 (Cost: 3)
- E. BFS: G1 (Cost: 7), UCS: G1 (Cost: 7)

BFS UCS

EX	Fringe
S	{A B G1}
A	{B G1 C}
B	{G1 C G2}
G1	

path = S-G1
path cost = 7

EX	Fringe
S	{A-1, B-1, G1-7}
A	{B-1, C-2, G1-7}
B	{C-2, G2-4, G1-7}
C	{G2-3, G1-7}
G1	

path = S-A-C-G1
path cost = 3

(true or false) For a search problem, the path returned by uniform cost search may change if we add a positive constant C to every step cost.

True. Consider that there are two paths from the start state (S) to the goal (G), $S \rightarrow A \rightarrow G$ and $S \rightarrow G$. $cost(S,A) = 1$, $cost(A,G) = 1$, and $cost(S,G) = 3$. So the optimal path is through A . Now, if we add 2 to each of the costs, the optimal path is directly from S to G . Since uniform cost search finds the optimal path, its path will change.

} ☆ \rightarrow Order of Traversal & optimum path changes

- (a) Depth first search will find an optimal path with respect to the cost of the path.
False
- (b) Depth first search will find an optimal path with respect to the number of steps in the path.
False
- (c) Breadth first search will find an optimal path with respect to the cost of the path.
False
- (d) Breadth first search will find an optimal path with respect to the number of steps in the path.
True

Consider a graph search problem where for every action, the cost is at least ϵ , with $\epsilon > 0$.

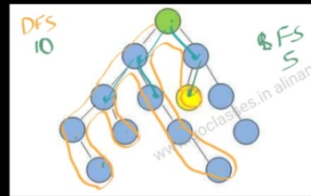
- (i) [true or false] Depth-first graph search is guaranteed to return an optimal solution.
False. Depth first search has no guarantees of optimality. Further, it measures paths in length and not cost.
- (ii) [true or false] Breadth-first graph search is guaranteed to return an optimal solution.
False. Breadth first search has no guarantees of optimality unless the actions all have the same cost which is not the case here.
- (iii) [true or false] Uniform-cost graph search is guaranteed to return an optimal solution.
True. UCS expands paths in order of least total cost so that the optimal solution is found.

- Consider the following graph. If there is ever a decision between multiple neighbor nodes in the BFS or DFS algorithms, assume we always choose the letter closest to the beginning of the alphabet first.



In what order will the nodes be visited using a Breadth First Search? **The answer is: ABDCEGHF**
In what order will the nodes be visited using a Depth First Search? **The answer is: ABCEHFGD**

- In the following graphs, assume that if there is ever a choice amongst multiple nodes, both the BFS and DFS algorithms will choose the left-most node first.



- A: BFS
- B: DFS
- C: Neither BFS nor DFS will ever encounter the goal node in this graph.
- D: BFS and DFS encounter same number of nodes before encounter the goal node

Starting from the green node at the top, which algorithm will visit the least number of nodes before visiting the yellow goal node?

f. (3 pts) Depth-first search uses $O(bd)$ space, where b is the branching factor of the search tree and d is the depth of the shallowest goal node. T/F? Explain.

False. Not necessarily. DFS uses $O(bm)$, where m is the depth of the search space—not $O(bd)$, where d is the depth of the shallowest goal node. DFS can “miss” a shallow goal node and continue searching deep into the search space. If the depth is infinite, DFS might not even finish! In fact, it’s not a complete search.

For this problem, assume that all of our search algorithms use tree search, unless specified otherwise.

- (a) For each algorithm below, indicate whether the path returned after the modification to the search tree is guaranteed to be identical to the unmodified algorithm. Assume all edge weights are non-negative before modifications.

(i) Adding additional cost $c > 0$ to every edge weight.

	Yes	No
BFS	<input checked="" type="radio"/>	<input type="radio"/>
DFS	<input checked="" type="radio"/>	<input type="radio"/>
UCS	<input type="radio"/>	<input checked="" type="radio"/>

(ii) Multiplying a constant $w > 0$ to every edge weight.

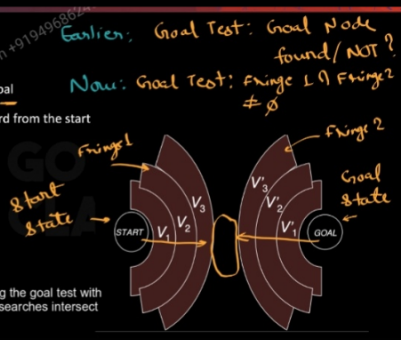
	Yes	No
BFS	<input checked="" type="radio"/>	<input type="radio"/>
DFS	<input checked="" type="radio"/>	<input type="radio"/>
UCS	<input checked="" type="radio"/>	<input type="radio"/>

Monicity is maintained for all possible path lengths if $p_1 \neq p_2$ are two paths before modification & $p_1' \neq p_2'$ are two paths after modification then
 $p_1 > p_2 \Rightarrow p_1' > p_2'$
 $p_1 = p_2 \Rightarrow p_1' = p_2'$
 $p_1 < p_2 \Rightarrow p_1' < p_2'$

Bidirectional Search

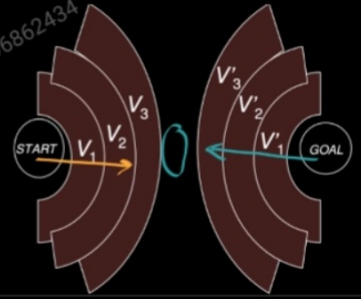
Unidirectional Search

- Breadth-first search from both start and goal
- Search backward from the goal and forward from the start simultaneously

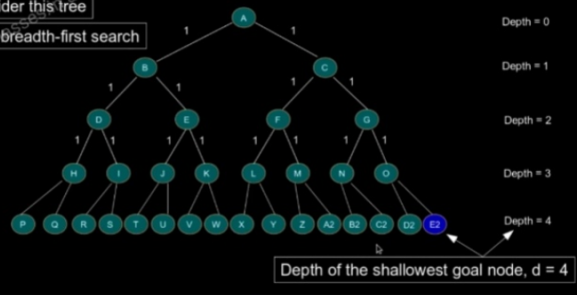


Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect

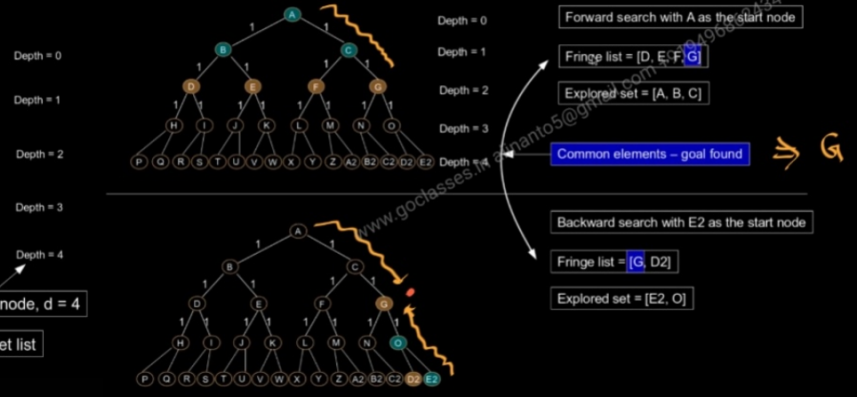
- Alternate searching from
 - start state \rightarrow goal
 - goal state \rightarrow start
- Stop when the frontiers intersect.
- Works well only when there are unique start and goal states
- Requires ability to generate "predecessor" states.



Let us consider this tree
Let us perform breadth-first search



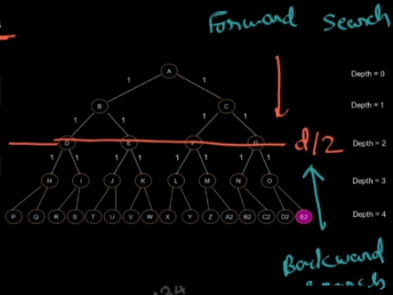
By the time we reach the goal node E2, all these nodes will be in the explored set list



Time & Space Complexity

Bidirectional Search : Time and Space complexity

In general, if d is the depth of the shallowest goal node
Both forward and backward search will meet after each search expands $d/2$ levels
Which means each search in the worst case generates $O(b^{d/2})$ nodes
Forward search generates $O(b^{d/2})$ nodes
Backward search generates $O(b^{d/2})$ nodes
Total nodes generated thus is $= O(b^{d/2}) + O(b^{d/2})$
 $= O(b^{d/2})$ nodes
Space and time complexity of bidirectional search $= O(b^{d/2})$



Optimum - Yes (if edges have uniform cost)

Completeness \rightarrow Yes if BFS is used in both direction

Let $b=10$ and $d=6$

Both forward and backward search will expand nodes upto depth $d/2 = 3$

Let breadth first search be used for both forward and backward search

How many nodes Forward search will then generate?

$10 + 100 + 1000 = 1110$ nodes

How many nodes Backward search will then generate?

$10 + 100 + 1000 = 1110$ nodes

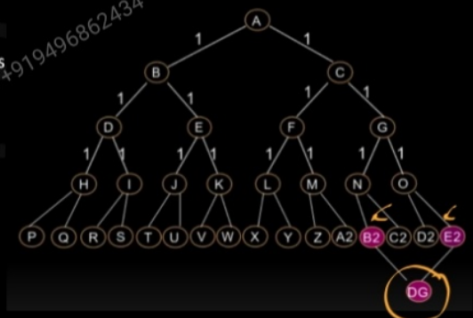
Total nodes generated by bidirectional search?

$1110 + 1110 = 2220$ nodes

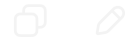
Limitations of Bidirectional search

- How can we search backward? It would require generating the predecessor of a node.
- This may not be easy always.
- What to do if there are more than one goal nodes

In this case, create a dummy goal node whose immediate predecessors are the immediate goal nodes.



In search algorithm that return the path. How does the algorithm keep track of path. In fringe this info is not stored. For example in UCS the search stops when we pop goal node from fringe. But how the algorithm returns the shortest path



Great question – this is a very common confusion when learning search algorithms like BFS, UCS, A*, and DFS.

Short answer:

Search algorithms *do* store path information – not explicitly inside the fringe priority queue or stack, but **within the nodes** that are placed in the fringe.

✅ How search algorithms keep track of paths

Every time you generate a node during search, the node data structure typically includes:

- `state`
- `parent` pointer (the node from which this one was generated)
- `action` (optional: which action led here)
- `path_cost` (for UCS/A*/etc.)

This is true even though books may say "the fringe stores nodes" but they rarely show the full structure.

✓ Node representation

A typical "node" in search is:

yaml

Copy code

```
node = {  
  state: S,  
  parent: parent_node,  
  action: a,  
  path_cost: g  
}
```

When you push something into the fringe (queue/stack/priority queue), you store this whole node.

The fringe **does** store path indirectly – through the parent links.

What happens in UCS?

UCS pseudocode (core idea)

sql

 Copy code

```
Insert start node into PQ with (state = start, cost = 0, parent = None)
```

```
While PQ not empty:
```

```
    node = pop lowest-cost node
```

```
    If node.state == GOAL:
```

```
        return solution by backtracking parents
```

```
    For each successor (s, step_cost):
```

```
        create new_node:
```

```
            state = s
```

```
            parent = node
```

```
            path_cost = node.path_cost + step_cost
```

```
        push new_node into PQ
```

When the goal is popped

UCS guarantees that when we *pop* the goal node from the priority queue, its `path_cost` is minimal.

Now we reconstruct the path:

Backtracking path

Starting from goal node:

lua

 Copy code

```
path = []
```

```
cur = goal_node
```

```
while cur != None:
```

```
    path.append(cur.state)
```

```
    cur = cur.parent
```

→ for this parent node has to be in memory and cannot be deallocated if not in fringe → All parents of fringe nodes + the predecessor chain to root has to be retained in memory

```
reverse(path)
return path
```

→ But this predecessor chain is a tuple per node
and space complexity — $O(bd) < O(b^d)$
→ This is why literature does not mention this.

This gives the optimal path.

Example

Suppose you reached the goal G via:

```
rust
```

 Copy code

```
Start -> A -> B -> G
```

The stored nodes will look like:

```
ini
```

 Copy code

```
node_G.parent = node_B
node_B.parent = node_A
node_A.parent = node_Start
node_Start.parent = None
```

So reconstructing the path is trivial.

Why textbooks sometimes hide this

To keep theoretical discussion simpler, they talk about:

- *states* in the fringe
- *operators*
- *costs*

But actual implementation **always** attaches parent pointers to nodes.

Summary

Concept

Answer

Does UCS store path in fringe?

Not explicitly, but **nodes inserted in fringe contain parent pointers.**

Concept

Answer

How do we get the path?

Backtrack from the goal node by following parent links.

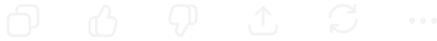
Why goal popped implies optimal path?

Because UCS (uniform-cost search) expands nodes in increasing order of path cost.

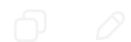
If you want, I can also show:

- Python implementation of UCS tracking parents
- How A* and BFS do the same
- Visual examples with diagrams

Just tell me!



Does that mean that all the nodes that are explored at that particular point of time has to be kept in memory and cannot be deallocated? That implies that for all the algorithms like DFS, BFS, and UCS, the explored nodes has to be kept in memory and not just the fringe. In literature, it says that the cost of memory or the space cost or space complexity is just the maximum size of the fringe. It does not talk about that we have to keep all the explored nodes in memory.



Great question — and this is exactly where most textbooks seem confusing.

You are **right** to notice the contradiction:

If nodes need parent pointers, shouldn't **all expanded nodes** stay in memory?

But textbooks say the **space complexity is the size of the fringe**.

Let's resolve this cleanly.

Key idea

Search algorithms do **NOT** always need to store all expanded/explored nodes.

It depends on:

1. **What the algorithm must guarantee** (optimality? completeness?)
2. **Whether you need to reconstruct the path**
3. **Whether revisiting states must be prevented** (closed list?)

Memory usage differs across BFS, DFS, UCS, A*, etc.

★ Important distinction: Node vs. State

Term	Meaning
Node	Data structure used internally (state, parent pointer, path_cost)
State	The abstract condition of the world (position on graph/gameboard/etc.)

You can **throw away nodes** while still remembering a state in the *closed list*.

🔍 When do we really need to keep expanded nodes?

Case 1 — Path reconstruction needed

If the algorithm must return the path (as BFS/UCS/A* do), then you must keep a **parent pointer chain** from:

```
powershell
```

 Copy code

```
Start → ... → Goal
```

But you do **NOT** need to keep *all* explored nodes, only:

- nodes in the fringe
- parents of nodes in the fringe
- parents of parents of fringe nodes
- back to root

This chain is **O(bd)** in worst case but often far smaller than the total explored set.

Nodes not used in these chains can be deallocated.

🔍 Closed list vs. Parent pointers

BFS

- Needs parent pointers only for one shortest path tree.
- Needs a closed set to avoid revisits.
- Space complexity: $\Theta(b^d)$ (fringe dominates).
- Explored nodes can be partially released except the BFS tree.

→ sub trees that have no further children to explore can be deallocated

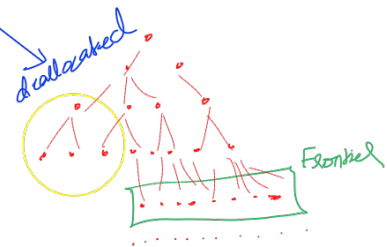
DFS

- Needs only the current path from root to the node.
- Can deallocate everything else.
- Space complexity: $\Theta(\text{depth})$
→ very small, no need to store explored states unless to avoid cycles (graph search).

UCS / A*

- Must keep the entire *frontier* and their parent pointers.
- Closed list needed to avoid re-expanding states.
- But explored nodes whose children have all been processed and whose state is safely in the closed list can be deallocated except the parent chain if it leads to a frontier node.

sub tree with no further children



🎯 Why literature says “space = fringe size”

Because:

1. Fringe dominates storage

In BFS/UCS/A*, the frontier is exponentially large ($\approx b^d$).

Parent pointers add only +1 reference per node, so $O(\text{fringe})$.

2. You don't need to keep all explored nodes

Most explored nodes can be thrown away as soon as their useful data has been extracted (except closed-list state markers, which are small).

3. Path reconstruction uses only one chain

After the goal is found, you reconstruct the path using the parent pointers **already attached to the fringe nodes and their ancestors**, not all explored nodes.

✓ Concrete Example (UCS)

Expand node A

→ generate B (parent = A) and C (parent = A)

→ put B and C into fringe

You can safely delete node A except for the tiny entry `closed[A] = true`.

Parent pointers for B and C still point to A's node *or a compact representation of it*.

Textbooks assume you store parent pointers in a compressed form (just a tuple: parent state, cost).

This keeps memory equal to the size of the frontier.

✓ Key conclusion

No, you do NOT need to store all explored nodes.

Yes, space complexity is dominated by the fringe.

Only nodes needed to reconstruct the final path stay alive.

This is why:

- DFS space = $O(\text{depth})$
- BFS space = $O(b^d)$
- UCS/A* space = $O(b^d)$

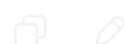
Because in all of these, the **frontier** is the memory bottleneck.

Explored nodes are tiny entries in a closed list or eliminated entirely.

If you want, I can also draw diagrams showing **exactly which nodes must be kept** during BFS, DFS, and UCS, and which ones can be safely deleted.



Yes



Here are **clear diagrams + explanations** showing exactly **which nodes must be stored** in memory at each moment for BFS, DFS, and UCS/A*.

This will make the space-complexity reasoning completely transparent.

★ 1. DFS (Depth-First Search) – What must be kept?

DFS **does NOT** need to store explored nodes (unless you want to avoid cycles).

It only needs:

- the current **path from root to the active node**
- the **unexplored children** of nodes on that path

DFS memory picture (example tree)

mathematica

 Copy code



Suppose DFS explores:

$S \rightarrow A \rightarrow D$, then backtracks.

Memory at deepest point:

mathematica

 Copy code

Call stack / path:

$S \rightarrow A \rightarrow D$

Fringe (children yet to explore):

E (child of A)

B, C (children of S)

✓ **Memory needed = $O(\text{depth})$**

When DFS backtracks, it **deallocates** nodes D , then A 's finished branches.

★ 2. BFS – What must be kept?

BFS needs:

- The **frontier (queue)** containing the next layer
- A **visited set** (only stores states, not nodes)

BFS does *not* keep the entire explored tree.

BFS example

```
mathematica
```

 Copy code

```
Layer 0: S
```

```
Layer 1: A B C
```

```
Layer 2: D E F G H I
```

Memory during BFS:

```
css
```

 Copy code

```
Fringe: entire Layer 2 (dominates memory)
```

```
Visited: tiny set {S,A,B,C,D,E,F,G,H,I}
```

! Parent pointers are still needed

But only for **nodes currently in the fringe**.

Nodes already expanded (like S, A, B, C) can be stored only as:

```
python-repl
```

 Copy code

```
visited[S] = true
```

```
parent_of[A] = S
```

```
parent_of[B] = S
```

```
...
```

This is *compact information*, not full nodes in memory.

✓ **Space = size of frontier = $O(b^d)$**

Not size of explored set.

★ 3. UCS / A* — What must be kept?

UCS and A* store:

- **Frontier** (priority queue) → large
- **Visited/closed set** → small (only states + best-cost-so-far)
- **Parent pointer per node** → small overhead

Example frontier (cost-sorted):

mathematica

 Copy code

```
PQ (frontier):
  (D, cost 2) parent = B
  (E, cost 3) parent = B
  (C, cost 5) parent = A
  ...
```

Nodes expanded earlier (like A or B) are not needed except for:

- `closed[A] = cost`
- `parent[B] = A`
- That's it. Their full node data is not stored.

✓ Space = $O(\text{frontier})$ = exponential

The closed list is tiny compared to the frontier.

★ 4. Why we don't need to store all explored nodes

For **path reconstruction**, you only need a chain:

powershell

 Copy code

```
Goal → parent → parent → parent → ... → Start
```

This chain comes from the **parent table**, not from storing full nodes.

You do NOT need the whole explored tree — only parent pointers for states.

This is usually stored like:

perl

 Copy code

```
parent[state] = previous_state
```

which is extremely small.

★ Summary Table (very important)

Algorithm	Stores full nodes?	Stores explored nodes?	Why?	Space
DFS	Only depth stack	No	Backtracking frees nodes	$O(\text{depth})$
BFS	Only nodes in frontier	Only states (visited)	To avoid revisits	$O(b^d)$
UCS/A*	Only nodes in frontier	Only states + costs	For optimality	$O(b^d)$
IDDFS	No	No	DFS-style	$O(\text{depth})$

Full explored nodes are never required, only parent pointers and visited flags.

This is why textbooks are correct: space complexity = *fringe size*, not total explored.

Want diagrams for memory at each step for BFS / DFS / UCS on a small graph like:

mathematica

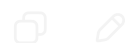
 Copy code

```
S → A → D
  ↓   ↘ E
  B → C → F
```

I can generate them with each algorithm step-by-step.



Can you summarise the entire thing in a pdf



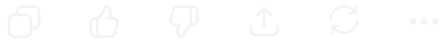
Your PDF summary is ready!

You can download it here:

 [Download search_memory_summary.pdf](#)

If you'd like a more detailed version, diagrams, or a multi-page formatted style, just tell me!

</>



Memory Usage in Search Algorithms — Summary 1. Key Concept Search algorithms do not store all explored nodes. They only store: • The frontier (fringe) nodes. • A small visited/closed set (state markers, not full nodes). • Parent pointers for path reconstruction. The fringe dominates space complexity. **2. DFS** • Stores only the current path + unexpanded siblings. • Does NOT store explored nodes. • Space = $O(\text{depth})$. • Backtracking frees memory. **3. BFS** • Must store the entire frontier of the current layer. • Explored nodes stored only as visited flags (very small). • Parent pointers stored compactly as $\text{parent}[\text{state}] = \text{previous_state}$. • Space = $O(b^d)$, dominated by frontier size. **4. UCS / A*** • Store frontier in a priority queue (largest part). • Closed list stores only the best cost and state (small). • Parent pointers allow reconstructing the path. • Space = $O(b^d)$, dominated by frontier. **5. Why Full Explored Tree Is Not Needed** Path reconstruction requires only the parent chain: Goal \rightarrow parent \rightarrow parent \rightarrow ... \rightarrow Start This is maintained in a compact parent table, not by keeping full nodes. **6. Final Summary** • DFS: stores minimal memory ($O(\text{depth})$). • BFS, UCS, A*: store mainly the frontier ($O(b^d)$). • Explored nodes are NOT stored as full node objects; only tiny state metadata. • Space complexity equals frontier size because it dominates all other storage.

parent table = python \rightarrow dictionary } cheaper than frontiers which is
in C \rightarrow hashtable } stack / queue / priority queue

Look at open source code for UCS in python pandas library for clarity & to refresh.

Iterative Deepening Search

Iterative Deepening Search

Depth-Limited Search

Idea: get DFS's space advantage with BFS's time / shallow-solution advantages

- Run a DFS with depth limit 1. If no solution...
- Run a DFS with depth limit 2. If no solution...
- Run a DFS with depth limit 3.

IDS (Iterative Deepening Search):

- Run DFS by searching only path of length 1 (DFS stops if length of path is greater than 1)
- If that doesn't find a solution, try again by running DFS on paths of length 2 or less
- If that doesn't find a solution, try again by running DFS on paths of length 3 or less
-
- Continue until a solution is found

Sounds horrible: We need to run DFS many times

Search proceeds like BFS but fringe in like DFS

- Take the idea of depth limited search one step further
- Starting at depth limit $L = 0$, we iteratively increase the depth limit, performing a depth limited search for each depth limit.

- Memory usage same as DFS
- Computation cost comparable to BFS even with repeated searches, especially for large B .

Space Complexity = $O(b \cdot m)$ = same as DFS

Time Complexity = $O(b^s)$ = same as BFS — why. see below.

Normal DFS for $B=10, S=5$ (depth of shallowest goal node)

$$\begin{aligned} \text{No. of Nodes expanded} &= 1 + 10 + 10^2 + 10^3 + 10^4 + 10^5 \\ &= \underline{\underline{11111}} \end{aligned}$$

Iterative deepening

Iteration	# Expansion	Graph
1 depth=0	1	
2 depth=1	1+b	
3 d=2	1+b+b^2	
4 d=3	1+b+b^2+b^3	
5 d=h	1+b+b^2+b^3+b^4	

$$\begin{aligned} \text{Total cost} &= 1 + 1b + 1b + b^2 + 1b + b^3 + \dots + b^2 + b^5 + 1 \\ &= \left. \begin{aligned} &1 \\ &1+b \\ &1+b+b^2 \\ &1+b+b^2+b^3 \\ &1+b+b^2+b^3+b^4 \end{aligned} \right\} = 1 \cdot b + b \cdot 5 + b^2 \cdot 4 + b^3 \cdot 3 + b^4 \cdot 2 + b^5 \cdot 1 \\ &= \sum_{i=0}^s (s+1-i)b^i = O(b^s) \\ \text{DFS: } &\sum_{i=0}^s b^i = O(b^s) \end{aligned}$$

Additional coefficient does not increase complexity

example with $b=10, S=5 \Rightarrow 1 \cdot 6 + (0.5) + 100 \cdot 4 + 1000 \cdot 3 + 10000 \cdot 2 + 100000 \cdot 1$

$= \underline{123456} \quad \left. \vphantom{\underline{123456}} \right\} O(b^S)$
 $= \underline{111111}$

Comparable to DFS

Completeness \rightarrow Yes - if finite cost goal exists
 Optimal \rightarrow if all costs are same - Yes \rightarrow otherwise no
 \rightarrow finds path of shortest #actions

- Trades a little time for a huge reduction in space
 - lets you do breadth-first search with (more space efficient) depth-first search
- Preferred uninformed search method

Performance of search algorithms on trees

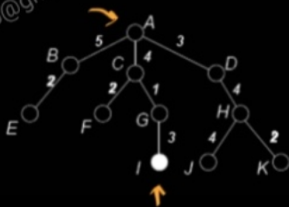
b: branching factor (assume finite) d: goal depth m: graph depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if 1	$O(b^d)$	$O(b^d)$
Uniform-cost search ²	Y	Y	$O(b^{C^*/\epsilon})$	$O(b^{C^*/\epsilon})$
Depth-first search	N	N	$O(b^m)$	$O(bm)$
Iterative deepening	Y	Y, if 1	$O(b^d)$	$O(bd)$

1. edge cost constant, or positive non-decreasing in depth
2. edge costs $\geq \epsilon > 0$. C^* is the best goal path cost.

Practice Questions & P40s

For the following tree, show the order of nodes visited for breadth-first search, depth-first search, uniform cost search, and iterative deepening search. The goal node is I and the numbers next to the edges indicate the associated cost. Break the ties from left to right.



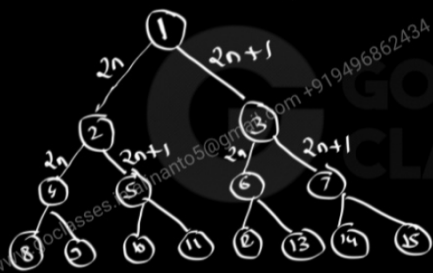
BFS \rightarrow ABCDEFHAI
 DFS \rightarrow ABECFHAI
 UCS \rightarrow ADCBHFEHI
 IDS \rightarrow A
 ABCD
 ABECFHDH
 ABECFHI

Does a finite state space always lead to a finite search tree?

Yes if the algorithm remembers states already visited and thus avoids visiting each state an infinite number of times.

No if the algorithm does not keep track of states already visited

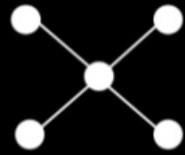
Consider a state space where the start state is number 1 and the successor function for state n returns two states, numbers $2n$ and $2n+1$. Draw the portion of the state space for states 1 to 15.



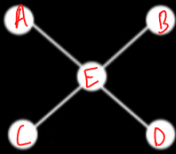
Suppose the goal state is 11. List the order in which nodes will be visited for breadth-first search, depth-limited search with limit 3 (depth starts with 1), and iterative deepening search.

BFS \rightarrow 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
 DLS₃ \rightarrow 1, 2, 4, 8, 9, 5, 10, 11
 IDS \rightarrow 1
 1 2 3
 1, 2, 4, 5, 3, 6, 7
 1, 2, 4, 8, 9, 5, 10, 11

Consider the following undirected graph on 5 nodes



	Assume you are performing breadth first search on this graph using a queue data structure. How many unique breadth first orderings are possible on this graph?
(A)	2
(B)	24
(C)	48
(D)	120

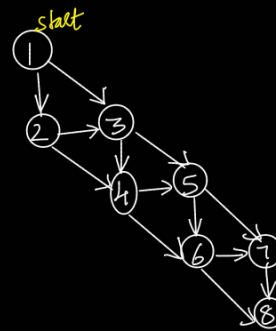


BFS starting @ A → A E C D B → Any order possible = 3! = 6
 // @ D → D E A C B → 6
 Similarly for C, B → 6 + 6 = 12
 Total 6 × 4 → when starting with A, D, B, C = 24
 When starting with E → E A B C D → 4! ordering possible
 ⇒ 24

starting with E
 24 + 24 = 48
 starting with A or D or C or B

GATE 2024

Q.44 Consider a state space where the start state is number 1. The successor function for the state numbered n returns two states numbered $n+1$ and $n+2$. Assume that the states in the unexpanded state list are expanded in the ascending order of numbers and the previously expanded states are not added to the unexpanded state list.



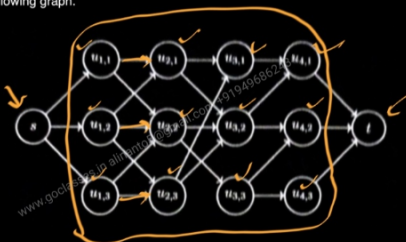
BFS
 expanded order = 1, 2, 3, 4, 5, 6
 returned path = 1, 2, 4, 6
 DFS
 Ex Order = 1, 2, 3, 4, 5, 6
 search path = 1, 2, 3, 4, 5, 6

Which ONE of the following statements about breadth-first search (BFS) and depth-first search (DFS) is true, when reaching the goal state number 6?

- (A) BFS expands more states than DFS.
- (B) DFS expands more states than BFS.
- (C) Both BFS and DFS expand equal number of states. ✓
- (D) Both BFS and DFS do not reach the goal state number 6.

DRDO CSE 2022 Paper 1 | Question: 33 (a)

Consider the following graph.



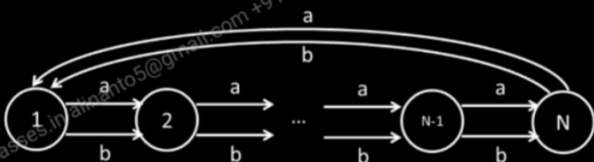
a) Order - $s \rightarrow v_{1,1}, v_{1,2}, v_{1,3}$
 $v_{2,1}, v_{2,2}, v_{2,3}$
 $v_{3,1}, v_{3,2}, v_{3,3}$
 $v_{4,1}, v_{4,2}, v_{4,3}$ } 12 nodes

a) How many nodes (apart from s) does the Breadth First Search algorithm discover before discovering t when starting from s .

b) $s \rightarrow v_{1,1} \rightarrow v_{2,1} \rightarrow v_{3,1} \rightarrow v_{4,1} \rightarrow t$ } 4 nodes

b) in case of DFS?

Consider a state space with N states, numbered 1 to N depicted in the following figure. There are 2 actions, a and b , that apply to each state. Both actions have the same effect: If you are in state N you return to state 1. Otherwise, if you are in state i it takes you to state $i+1$. The last state, N , is the only goal state.



Because this state space has a cycle depth-first search can fall into an infinite loop when solving this problem.

Consider a problem whose search space is a tree, with a single goal state located at level d . How many times will a state at level $d-4$ have its successors generated by iterative deepening search?

4. The states at level d have no successors generated, the states at level $d-1$ have successors generated just once, the states at level $d-2$ have successors generated twice, and so on. The states at level k have their successors generated k times.

0 -4 - successes not generated
 0 -3 when limit is $d-2$
 0 -2
 0 -1
 (1) - d level

But note the question:

How many times will a state at level $d - 4$ have its successors generated?

A node generates its successors only when the depth limit is strictly greater than its depth

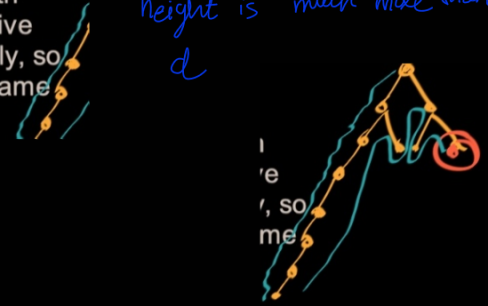
- At limit $d - 4$: node is reached but **no successor generation** (limit reached).
- At limits $d - 3, d - 2, d - 1, d$: successors are **generated**.



True/False

Consider a problem whose search space is a tree and whose only solution is at depth d . Depth-first search can take twice as long (expand twice as many states) as iterative deepening search to find the solution. (Assume both methods order nodes identically, so that if they have to choose a state from some set of states they will both select the same state.)

True in case tree height is much more than d



What is necessarily true regarding iterative deepening on any search tree?

- Complete as opposed to DFS tree search
- Strictly faster than DFS tree search \rightarrow Not always
- Strictly faster than BFS tree search \rightarrow Never
- More memory efficient than BFS tree search

If we are asked to implement an uninformed search algorithm which will use the smallest amount of space (polynomial space complexity) but will also be sure to find the goal (optimal), which of the uninformed search methods that we've studied should we choose?

- A) DFS
 B) BFS
 C) UCS
 D) IDS

b : branching factor (assume finite) d : goal depth m : graph depth

	Complete	optimal	time	space
Breadth-first search	Y	Y, if 1	$O(b^d)$	$O(b^d)$
Uniform-cost search ²	Y	Y	$O(b^{C^*/\epsilon})$	$O(b^{C^*/\epsilon})$
Depth-first search	N	N	$O(b^m)$	$O(bm)$
Iterative deepening	Y	Y, if 1	$O(b^d)$	$O(bd)$

Iterative is the most preferred uninformed search algorithm.

Informed Search

- ↳ Disadvantages of uninformed (VCS)
 - ↳ Only prioritise cost from start node
 - ↳ Does not prioritise direction to goal
 - ↳ Treats all directional uniformly

Heuristic

- A search heuristic $h(n)$ is an estimate of the cost of the optimal (cheapest) path from node n to a goal node.
- Heuristics provide a way for the algorithm to prioritize which nodes to explore next.
- Instead of exhaustively searching all possible paths, we use heuristics to focus on the most promising routes.

Common → Manhattan distance
 → Euclidean distance

Best First Search

- Use an evaluation function $f(n)$ for node n .
- Always choose the node from fringe that has the lowest f value.
- A search strategy is defined by picking the order of node expansion
- Idea: use an evaluation function $f(n)$ for each node
 - estimate of "desirability"
 - Expand most desirable unexpanded node
- Implementation: Order the nodes in fringe in decreasing order of desirability
- Special cases:
 - greedy best-first search
 - A* search

- Breadth First =
 - Best First
 - with $f(n) = \text{depth}(n)$
- Uniform cost search =
 - Best First
 - with $f(n) = \text{the sum of edge costs from start to } n$

Special cases of Best first search

First Attempt: Greedy Best First Search

Simplest use of heuristic function: Always select the node with smallest $h(n)$ for expansion (i.e., $f(s) = h(s)$)

→ $f(n) = h(n)$

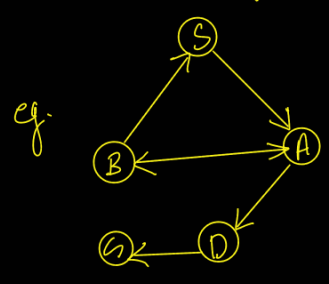
Uniform cost search: $f(n) = g(n)$
 Evaluation function

$g(n)$: Total cost of path from starting node to current node.

$h(n)$: Estimated cost from current n to the goal node.

$f(n)$: Evaluation function

- Solution not optimal
- Not complete - may be stuck in loops



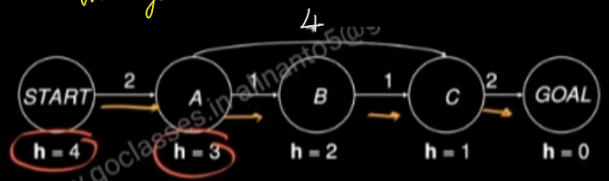
- $h(S) = 10$
- $h(A) = 7$
- $h(B) = 1$
- $h(D) = 4$
- $h(C) = 0$

Start at A
 eg. BFS → SABABAB

Time Complexity = $O(b^m)$
 worst case with bad heuristic

Space Complexity = $O(b^m)$

- Loops & non-optimal solution possible even with good heuristic



- What solution do we find in this case?
- Greedy search clearly not optimal, even though the heuristic function is non-stupid

Beam Search

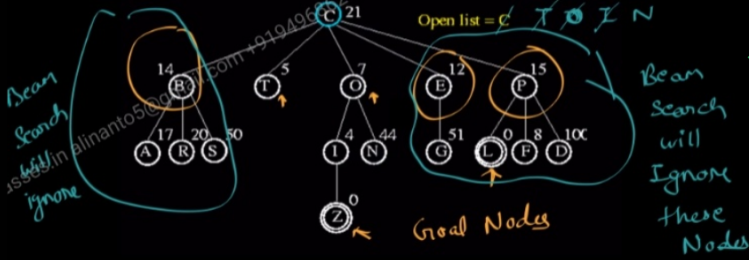
Beam Search

Same as greedy best first search

- Use an evaluation function $f(n) = h(n)$, but the maximum size of the nodes list is k , a fixed constant
- Only keeps k best nodes as candidates for expansion, and throws the rest away
- More space-efficient than greedy search, but may throw away a node that is on a solution path

Beam Search : Example with $k=2$

GBFS: B T O E P



- Beam Search only keeps the best (lowest- h) k nodes on open list

k is the "beam width"

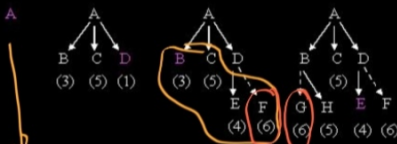
- $k=1$, Hill climbing search
- $k=\infty$, Best first search

- Hill climbing: select the best neighbor and expand that,
- Beam search: select the best k nodes, k is the width of beam
- Best first: select the best, and expand

Note Complete & Not optimal

→ If L was the only goal node, then beam search with $k=2$ is incomplete

Example: Beam Search ($k=3$)



Beam Search: $S = \{A\}$ $S = \{B, C, D\}$ $S = \{B, C, E\}$ $S = \{C, E, H\}$

Greedy BFS: $S = \{A\}$ $S = \{B, C, D\}$ $S = \{B, C, E, F\}$ $S = \{G, H, C, E, F\}$

- In general, the Beam Search Algorithm is not complete.
- May throw away a node that is on a solution path
- Even given unlimited time and memory, it is possible for Algorithm to miss the goal node when there is a path from start node to the goal node (example in next slide).
- A more accurate heuristic function and a larger beam width improve Beam Search's chances of finding the goal.

Beam Search : Time Complexity

- Depends on the accuracy of the heuristic function.
- In the worst case, the heuristic function leads Beam Search all the way to the deepest level in the search tree.
- The worst case time = $O(k \cdot m)$ where k is the beam width and m is the maximum depth of any path in the search tree.

B/k : Beam width

GBFS: $O(b^m)$

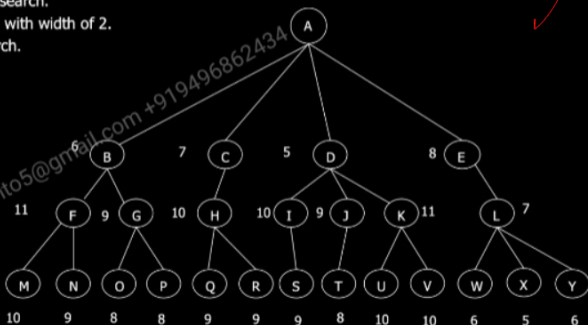
Since all explored nodes kept in memory the cost of keeping explored cost in memory outweighs the space cost of fringe

Beam Search : Space Complexity

- Beam Search's memory consumption is its most desirable trait.
- Since the algorithm only stores k nodes at each level in the search tree, the worst-case space complexity = $O(k \cdot m)$ where k is the beam width, and m is the maximum depth of any path in the search tree.

Space Complexity = $O(km)$
Time Complexity

- Consider the following tree as part of a search space
- The numbers = estimated distance to the goal ← Heuristic
- Draw the search tree and write the order of expansion of the nodes when applying:
 - Hill climbing search.
 - Beam search with width of 2.
 - Best first search.

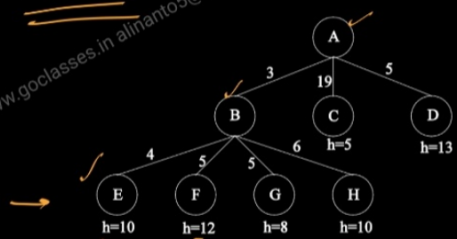


HCS → A D J T

Beam ($k=2$) → A D B G O

BFS → A D B C E L X W Y G O P J T
H Q R I S F N M K U V

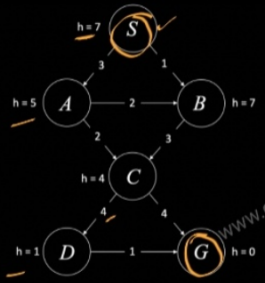
Consider the following search tree produced after expanding nodes A and B, where each arc is labeled with the cost of the corresponding operator, and the leaves are labeled with the value of a heuristic function, h . For uninformed searches, assume children are expanded left to right. In case of ties, expand in alphabetical order.



Which one node will be expanded next by each of the following search methods?

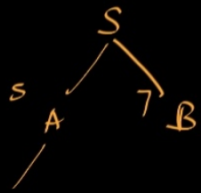
- (a) [3] Depth-First search — E
- b) BFS — C
- c) UCS — D

We will investigate various search algorithms for the following graph. Edges are labeled with their costs, and heuristic values h for states are labeled next to the states. S is the start state, and G is the goal state. In all search algorithms, assume ties are broken in alphabetical order.



Given the above heuristics, what is the order that the states are going to be expanded in, assuming we run greedy graph search with the heuristic values provided.

Index	1	2	3	4	5	Not Expanded
S	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
A	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
B	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
C	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
D	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
G	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>



What path is returned by greedy graph search?
 $S \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow G$
 $S \rightarrow A \rightarrow C \rightarrow D \rightarrow G$
 $S \rightarrow A \rightarrow C \rightarrow D \rightarrow G$
 $S \rightarrow A \rightarrow C \rightarrow D \rightarrow G$
 None of the above

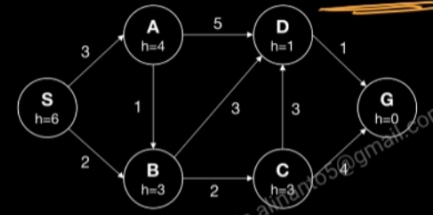
True/False: Greedy search has the same worst-case number of node expansions as DFS.

True

True/False: Greedy graph search is guaranteed to return an optimal solution.

False. Greedy search makes no guarantees of optimality. It relies solely on the heuristic and not the true cost.

Consider the following graph. For the following subquestions, ties are broken in alphabetical order.



(iv) [1 pt] What path would DFS return?

S A B C D G

(v) [1 pt] What path would BFS return?
 Ex (S A B D C G) path = S A D G

(vi) [1 pt] What path would UCS return?
 Ex (S B A C D G) path = S B D G

(vii) [1 pt] What path would Greedy search return?
 S B D G path = S B D G

→ Here in BFS, there may be a confusion whether D or C comes first in depth 2
 → Always remember tie breaks are for entering values into LIFO
 → Since A is expanded first D entered the queue first so D before C.

A* algorithm

A* Algorithm

ucs

- It is similar to Dijkstra's algorithm, but its approach is much more goal-oriented.
- For the target node, it first computes an estimate of the shortest distance. Based on this estimate, it will achieve a fast computation.
- The priority queue also includes the heuristic defined values for how much it has left to go.

$$g(n) + h(n)$$

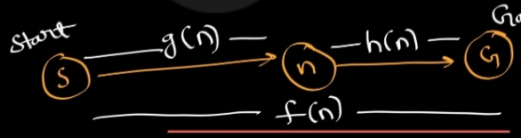
A* Algorithm : Evaluation function

- $f(n)$ = estimated total cost of path through n to goal
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal

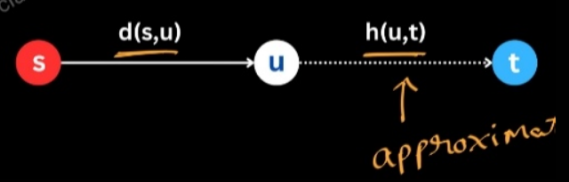
ucs $\Rightarrow f(n) = g(n)$
 BFS $\Rightarrow f(n) = h(n)$
 A* $\Rightarrow f(n) = g(n) + h(n)$

Evaluation function for A*

$$f(n) = g(n) + h(n)$$



$$\text{Priority} = d(s, u) + h(u, t)$$



- Uniform-cost orders by path cost, or backward cost $f(n)=g(n)$
- Best-first orders by goal proximity, or forward cost $f(n)=h(n)$
- A* Search orders by the sum: $f(n) = g(n) + h(n)$



Uniform Cost Search

Make a PriorityQueue todo-list of paths
 Put a path with just the start in the todo-list
 While the todo-list isn't empty

1. Take a path out of the todo-list
2. Call the last node in the path "currNode"
3. If "currNode" is the goal, you are done.
4. If you have seen currNode before, skip it.
5. for all neighbors of currNode
 Make a newPath = path + neighbor
 Add the new path to the todo-list
 Priority = pathLength + ...

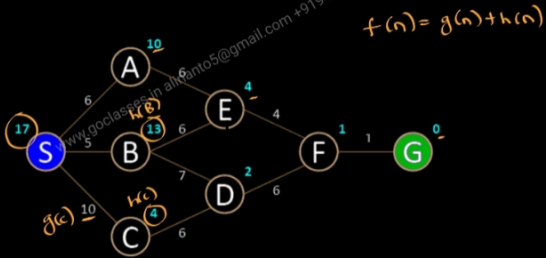
Algorithm that even takes care of backtracking
 → Instead of storing bring in priority que
 Entire path is saved
 → Complexity does not increase
 → No loops or revisits

A star algorithm

Make a **PriorityQueue** todo-list of paths
 Put a path with just the start in the todo-list
 While the todo-list isn't empty

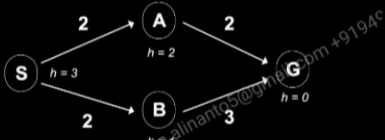
1. Take a path out of the todo-list
2. Call the last node in the path "currNode"
3. If "currNode" is the goal, you are done.
4. If you have seen currNode before, skip it.
5. for all neighbors of currNode
 Make a newPath = path + neighbor
 Add the new path to the todo-list
 Priority = pathLength + h(neighbor, goal)

Q1) Perform the A* Algorithm on the following figure. Explicitly write down the queue at each step.



Ex	Fringe	Path
S	{ C-14, A-16, B-18 }	S A E F G
C	{ A-16, B-18, D-18 }	
A	{ E-16, B-18, D-18 }	
E	{ F-17, B-18, D-18 }	
F	{ G-17, B-18, D-18 }	
G		

Should we stop when we enqueue a goal?



No: only stop when we dequeue a goal

Ex	Fringe
S	{ B-3, A-4 }
B	{ A-4, G-5 }
A	{ G-4, G-5 }
G	

→ if we stop @ goal enqueue, we might miss better path
 → if we stop @ goal deque, less chance of missing optimal path

Admissible vs Consistent Heuristic

Admissible heuristic function :

- Let $h^*(n)$ denote the cost of the optimal path from node n to any goal node. A search heuristic $h(n)$ is called **admissible** if $h(n) \leq h^*(n)$ for all nodes n , i.e., if for all nodes it is an **underestimate** of the cost to any goal.
- In other words, $h(n) \leq$ true cost.
- If the heuristic function, h always underestimates the true cost ($h(n)$ is smaller than $h^*(n)$), then A* is guaranteed to find an optimal solution.

$h^*(n)$ - true cost from node n to goal
 $h(n)$ - heuristic estimate
 $h(n) \leq h^*(n) \Rightarrow$ Admissible

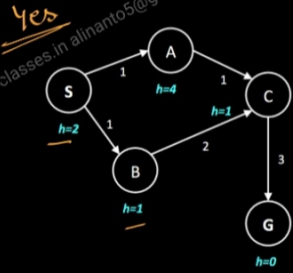
Admissible \Rightarrow Optimal

Intuition

- For A* to stop (goal node has to be dequeued)
- The evaluation function of a goal node is the actual cost of the path under consideration
- When an parent to a goal node is expanded, the goal node enters the priority queue - with value of actual cost of path under consideration
- For the goal node to be dequeued, all the other path cost estimates in the fringe must be greater than the goal node cost - (which is the cost for the goal via the path that is about to be dequeued)

→ Since all evaluations in the fringe are underestimates, the goal node will be dequeued, when no other paths in fringe can beat cheap the path under consideration.

Q2) Consider the following graph. Is admissibility constraint satisfied by the heuristic function h ?

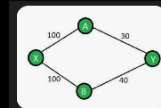


Yes

$h(S)=2 \leq h^*(S)=5 \checkmark$
 $h(A)=4 \leq h^*(A)=4 \checkmark$
 $h(B)=1 \leq h^*(B)=5 \checkmark$
 $h(C)=1 \leq h^*(C)=3 \checkmark$
 $h(G)=0 \leq h^*(G)=0 \checkmark$
 if $h(n) \leq h^*(n) \forall n$

Consistent (monotone) Heuristics :

- The distance between any pair of nodes is greater than or equal to the difference in their heuristics.
- For every pair of nodes (p, q) in the graph: $|d(p, q) - d(q)| \geq |h(p) - h(q)|$ — Only necessary not sufficient



A consistent heuristic is always admissible because consistency implies the triangle inequality, which ensures the heuristic never overestimates the cost to the goal. The proof shows that for any node, the estimated cost to the goal is less than or equal to the cost to a successor plus the successor's estimated cost to the goal. By induction, this holds for the entire path, proving that the heuristic estimate is always less than or equal to the actual path cost.

Graph search v/s tree search :

- The Problem Is Always a Graph.
- It is always assumed you're dealing with a **graph**.
- The distinction lies in the **traversal pattern** that is used to search through the **graph**, which can be graph-shaped or tree-shaped.
- In **Graph Search** you hold a list of explored **nodes**, while in **Tree Search** you don't.
- Tree search can visit a state multiple times.

Consistent \Rightarrow Admissible

Q4) If we used an inadmissible heuristic in A* tree search, would the search be complete?

Yes / No

YES
A* tree search would visit all the nodes eventually and would find a path to the goal state if there exists one.

A* is always Complete like UCS
Since actual cost is part of evaluation, it will prevent cycles as cost will add up.

	admissible Heuristic	Inadmissible Heuristic	Consistent Heuristic
A* tree search	Complete, Optimal	Complete, Not optimal	Complete, Optimal
A* graph search	Complete, not optimal	Complete, Not optimal	Complete, Optimal

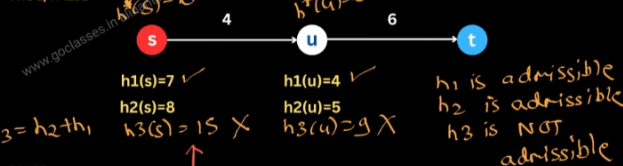
The speed execution of A* search is highly dependent on the accuracy of the heuristic algorithm that is used to compute $h(n)$.

TRUE/FALSE

TRUE

Q 15) The sum of several admissible heuristics is still an admissible heuristic.

TRUE/FALSE



FALSE

Q 14)

A* graph search is guaranteed to expand no more nodes than depth-first graph search.

TRUE/FALSE

FALSE

Depth-first graph search could, for example, go directly to a sub-optimal solution.

Correct definition for consistency — Necessary & sufficient

For a heuristic to be consistent, is it necessary that for every pair p, q , the cost of the shortest path between p and q should be greater than or equal to the absolute difference between the heuristic value at p and q ?

Short answer: Yes — that condition is necessary (but not sufficient by itself) for consistency.

✓ Why this condition is necessary

A heuristic h is **consistent** (or monotone) if for every node n and every successor n' :

$$h(n) \leq c(n, n') + h(n') \quad \text{--- necessary \& sufficient}$$

This is equivalent to:

$$h(n) - h(n') \leq c(n, n')$$

Now apply the symmetric situation (swapping n and n'):

$$h(n') - h(n) \leq c(n', n)$$

If the edges are undirected or you consider general shortest-path distances, then $c(n, n') = c(n', n)$, and combining the two:

$$|h(n) - h(n')| \leq c(n, n')$$

Now extend this through a path $p = x_0, x_1, \dots, x_k = q$. Using consistency on every edge:

$$h(x_0) - h(x_k) \leq c(x_0, x_1) + c(x_1, x_2) + \dots + c(x_{k-1}, x_k)$$

Which gives:

$$h(p) - h(q) \leq \text{cost of shortest path from } p \text{ to } q$$

Similarly:

$$h(q) - h(p) \leq \text{shortest path cost}(p, q)$$

Thus:

$$|h(p) - h(q)| \leq d(p, q) \quad \text{--- only necessary, not sufficient}$$

where $d(p, q)$ is the shortest path cost between p and q .

Therefore, this inequality must hold for every pair p, q in the graph.

✓ So the answer is:

Yes, for a heuristic to be consistent, it is necessary that:

$$|h(p) - h(q)| \leq d(p, q) \quad \forall p, q.$$

This follows by repeatedly applying the consistency inequality along the shortest path between p and q .

⚠ But note:

This condition is **necessary but not sufficient**.

When checking for consistency, it is sufficient to check this condition @ every edge in the graph.
if \forall edges $A \rightarrow B$
 $h(A) - h(B) \leq c(A, B)$
cost = $d(n, n')$
→ But direction must be consistent.

Why is a A-star graph search not optimal given an admissible heuristic but an A-star tree search is optimal given an admissible heuristic?

Here is the clear, correct reasoning:

Why A* Tree Search is Optimal with an Admissible Heuristic

(but A* Graph Search is *not* necessarily optimal with only admissibility)

1. A* Tree Search

Tree search never checks for repeated states.

So if a goal is found, it is *guaranteed* that:

- This is the **first time** the algorithm has found *any path* to that goal.
- A* expands nodes in **non-decreasing order of $f(n) = g(n) + h(n)$** .

With an **admissible heuristic** (never overestimates), A* tree search ensures:

- The optimal path to the goal has the **smallest possible f-value** among all goal paths.
- Since A* tree search finds nodes in order of increasing f , the **first time the goal is pulled from the fringe, it must be optimal.**

Admissible heuristic \Rightarrow A* tree search optimal

Why A* Graph Search Is NOT Guaranteed Optimal with Only Admissibility

In graph search we avoid revisiting states by storing an *explored set*.

This introduces a subtle danger:

A* graph search may encounter a state *first* on a suboptimal path.

- Suppose A* first reaches state S with a bad (high-cost) path.
- It adds S to the **closed set (explored)**.
- Later, a better (lower-cost) path to the same state exists, but A* **will not explore it**, because S is already closed.

This can make A* graph search discard the optimal solution prematurely.

Admissibility guarantees only *never overestimates h*

—it does **not** prevent A* from temporarily reaching a state suboptimally.

What additional property fixes the problem?

To make A* graph search optimal, the heuristic must be:

✓ **consistent (monotonic)**

$$h(n) \leq \text{cost}(n \rightarrow n') + h(n')$$

Consistency ensures:

- f-values never decrease along any path**, therefore:
- The **first time any state is expanded**, you already have the optimal path to it.

Thus closing a state is safe.

➔ A* graph search is optimal only if the heuristic is consistent (not merely admissible).

Summary

Setting	Admissible Enough?	Why / Why Not
A* Tree Search	✓ YES	First time reaching the goal is guaranteed optimal since nodes are expanded in non-decreasing f . No repeated-state complications.
A* Graph Search	✗ NO	You may reach a state via a suboptimal path first, store it in CLOSED, and block the optimal path later.
A* Graph Search with Consistent h	✓ YES	Consistency ensures that the first time a node is expanded, its path is optimal.

Key Insight

The problem is NOT A* itself.

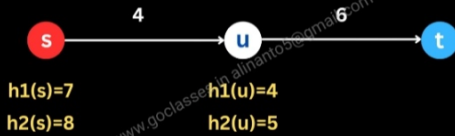
The problem is the CLOSED LIST in graph search.

Admissibility alone doesn't prevent premature closing of nodes reached suboptimally.

Consistency does.

Maximum of two admissible heuristics will result in an admissible heuristic.

TRUE/FALSE



TRUE

Q 18)

Let $h_1(s)$ be an admissible A* heuristic. Let $h_2(s) = 2h_1(s)$. Then the solution found by A* tree search with h_2 is guaranteed to be an optimal solution.

TRUE/FALSE

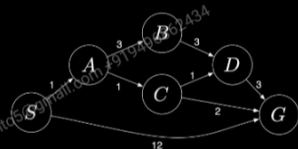
FALSE

Q 20)

Consider the following graph. The heuristics for this problem shown in the table below. Choose the correct option.

- h_1 is admissible
- h_2 is admissible
- h_1 is consistent
- h_2 is consistent

State	h_1	h_2
S	5	4
A	3	2
B	6	6
C	2	1
D	3	3
G	0	0



Ans : (b)

Q 21)

Consider a graph search problem where for every action, the cost is at least c , with $c > 0$. Assume the used heuristic is consistent.

- Depth-first graph search is guaranteed to return an optimal solution.
- Breadth-first graph search is guaranteed to return an optimal solution.
- A* graph search is guaranteed to expand no more nodes than depth-first graph search.
- A* graph search is guaranteed to return an optimal solution.

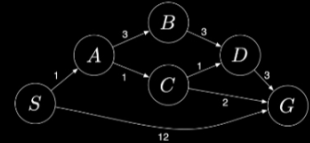
Q.23 Let h_1 and h_2 be two admissible heuristics used in A* search. Which ONE of the following expressions is always an admissible heuristic?

- $h_1 + h_2$
- $h_1 \times h_2$
- $h_1/h_2, (h_2 \neq 0)$
- $|h_1 - h_2|$

Q 19)

What path would A* graph search, using a consistent heuristic, return for this search problem?

- S - A - C - G
- S - G
- S - A - C - D - G
- S - A - B - D - G



Ans : (a)

Which of the following is/are true?

- Breadth First Search explores equally in all directions.
- Dijkstra's Algorithm lets us prioritize which paths to explore. (UCS)
- A* is a modification of Dijkstra's Algorithm that can find paths to all the nodes.
- A* is a modification of Dijkstra's Algorithm that is optimized for a single destination.

A* is a modification of Dijkstra's Algorithm that is optimized for a single destination. Dijkstra's Algorithm can find paths to all locations; A* finds paths to one location, or the closest of several locations. It prioritizes paths that seem to be leading closer to a goal.

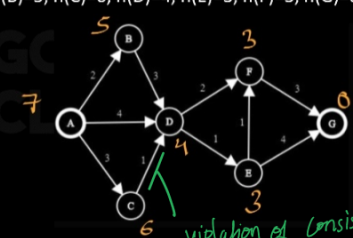
If $h(n) \leq h^*(n)$ for all nodes n , i.e., the heuristic value for all nodes is an underestimate of the cost to any goal, then the A* algorithm will be the same or faster than Dijkstra.

TRUE/FALSE

TRUE
If $h(n)$ is always lower than (or equal to) the cost of moving from n to the goal, then A* is guaranteed to find a shortest path.

Q 26)
Consider the following directed graph in which we are searching from start state A to goal state G. The number over each edge is the transition cost. Additionally, we are given a heuristic function h as follows ($h(A)=7, h(B)=5, h(C)=6, h(D)=4, h(E)=3, h(F)=3, h(G)=0$). The given heuristic function is:

- a) Admissible as well as consistent
- b) Admissible but not consistent
- c) Consistent but not admissible
- d) Neither admissible nor consistent



violation of consistency but admissible

- Which of the following statements is/are true about a heuristic function h ?
- (i) If $h(n) = h^*(n)$ for all n , then algorithm A* will only expand nodes on the optimal path (ignoring ties).
 - (ii) If h is admissible, the smaller $h(n)$ is, the fewer nodes that A* will expand.
 - (iii) If $h(n)$ is always less than or equal to the cost of the cheapest path from n to the goal, then A* tree search is guaranteed to find an optimal solution.

- A. Only (i) is true
- B. Only (ii) is true
- C. Only (iii) is true
- D. Both (i) and (iii) are true

Ans : D

GATE 2024 Sample Question Paper Data Science

Consider the grid world shown in the figure below. An agent is planning to move from the starting location (x_s, y_s) to the final location (x_f, y_f) . The obstacles along the path are triangular in form. Consider the following heuristic functions to conduct A* search.

- h_c assumes the obstacles are the smallest circles circumscribing the triangles.
- h_r assumes the obstacles are smallest rectangles circumscribing the triangles.
- h'_c assumes the obstacles are largest circles inscribed in the triangles.
- h'_r assumes the obstacles are largest rectangles inscribed the triangles.

Which of the following statement(s) is/are true?

- a) h_c is an admissible heuristic
- b) h_r is an admissible heuristic
- c) h'_c is an admissible heuristic
- d) h'_r is an admissible heuristic



for admissible $h(n) \leq h^(n)$
→ underestimate cost of path → reduced size
of simpler shape of obstacle*

(c) (d)

If $h(n)$ is exactly equal to the cost of moving from n to the goal, then A* will only follow the best path and never expand anything else

TRUE/FALSE

TRUE
Will only follow the best path, and will find the best path fastest (but requires perfect knowledge).

Which of the following algorithms is/are guaranteed to give an optimal solution?

- a) Greedy Best First Search
- b) A* with zero heuristic
- c) A* with consistent heuristic
- d) Depth First Search

Ans : (b), (c)

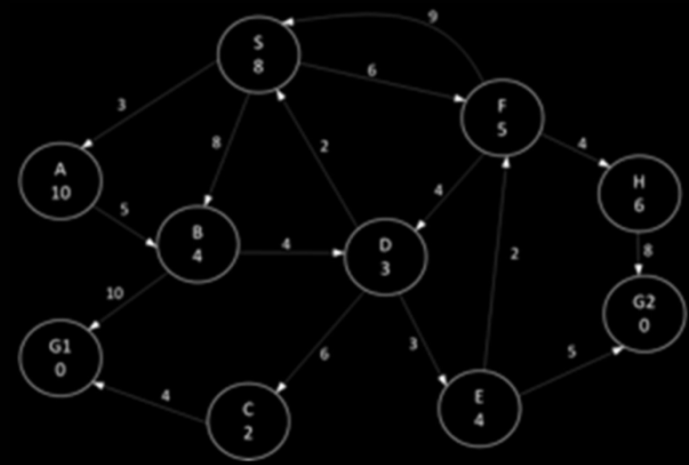
GATE 2024 Sample Question Paper Data Science and Artificial Intelligence:

Consider the search space depicted in the Figure below. S is the initial state. G1 and G2 are two states that satisfy the goal test. The cost of traversing from one state to another is depicted by the numerical value close to the edge connecting the two states. The estimated cost to the goal is reported inside the states. Use alphabetical order of nodes to break ties. Which goal state is reached if you perform A* (graph) search? What is the largest value that the heuristic function can take for node A while still being admissible?

- a) G1 and 16
- b) G1 and 15
- c) G2 and 16
- d) G2 and 15



Ans : (b)



EX | Fringe

- S {F-11, B-12, A-13}
- F {B-12, A-13, D-13, H-16, S-23}
- B {A-13, D-13, H-16, G1-18, S-23}
- A {D-13, H-16, G1-18, S-23}
- D {H-16, E-17, G1-18, C-19, S-23}
- H {E-17, G1-18, G2-18, C-19, S-23}
- E {G1-18, G2-18, C-19, S-23}

G1 path = S-B-G1

Q 31)

We define an evaluation function for a heuristic search problem as:
 $f(n) = (w * g(n)) + ((1 - w) * h(n))$
 where $g(n)$ is the cost of the best path found from the start state to state n , $h(n)$ is an admissible heuristic function that estimates the cost of a path from n to a goal state, and $0.0 \leq w \leq 1.0$.

What search algorithm do you get when $w = 0$?

- A. Breadth-First search
- B. Uniform-Cost search (b) ✓
- C. Greedy Best-First search (a) ✓
- D. Algorithm A* search (a) ✓
- E. None of the above

what search algorithm when $w=0.5$

Algorithm	Complete?	Optimal?	Memory usage
Breadth-first search	Y	N	High
Depth-first search	N	N	Low
Depth-first iterative deepening	Y	N	Low
Best-first search	N	N	High
A*	Y	Y	High

A* Algorithm : The Dark Side

- A* can use lots of memory. In principle: $O(\text{number of states})$ *worst case*
- For really big search spaces, A* will run out of memory.

Iterative deepening A*

- Like Iterative Deepening DFS
 - But the depth bound is measured in terms of the f value.
- If you don't find a solution at a given depth
 - Increase the depth bound: to the minimum of the f -values that exceeded the previous bound.

IDA* ← *More time* A*
 ← *Less Memory* A*

→ less space
 → more time

Complete
 Optimal → if admissible (tree search)
 → if consistent (graph search)

- The main difference between IDA* and standard iterative deepening is that the cutoff used is the f -cost ($g+h$) rather than the depth.
- At each iteration, the cutoff value is the smallest f -cost of any node that exceeded the cutoff on the previous iteration.

IDA* : Algorithm

- Let F be a limit on $f()$ values.
- Run a DFS from the start state. But never expand any (generated) node n with $f(n) > F$. *Means equalities are expanded.*
- If the goal is reached, we are done.
- Otherwise increase F , and restart the search.

The nodes having lesser f value than the limit F will be expanded.

First Iteration → Take $F = f(s)$; $s \rightarrow$ start node

- Later iterations: All nodes n reachable from S and with $f(n) \leq F$ are expanded.

Take $\min (f(n) \mid n \text{ is generated and } f(n) > F)$
not expanded

$f(n) \leq F$ is expanded } unlike normal IDS where
 $f(n) > F$ not expanded }

★ → depth = limit nodes not even expanded.

Which of the following is NOT true about IDA*?

- A. It uses a threshold-based depth-first search. ✓
- B. It requires less memory than A*. ✓
- C. It guarantees finding the optimal solution if the heuristic is admissible. ✓
- D. It explores nodes in breadth-first order. *False*

Why is the threshold in IDA* increased to the minimum $f(n)$ value exceeding the current threshold rather than incremented by a fixed value?

- A. To ensure that no potential solutions are skipped. *True*
- B. To reduce the time complexity of the algorithm.
- C. To limit memory usage during each iteration.
- D. To ensure that the heuristic remains admissible.

+5

Q.1	Let b be the branching factor of a search tree. If the optimal goal is reached after d actions from the initial state, in the worst case, how many times will the initial state be expanded for iterative deepening depth first search (IDDFS) and iterative Deepening A* search (IDA*)?
(A)	IDDFS - d , IDA* - d .
(B)	IDDFS - d , IDA* - b^d
(C)	IDDFS - b^d , IDA* - d .
(D)	IDDFS - b^d , IDA* - b^d .

→ Even though options are wrong as per reasoning below, the best suitable option is B.

Definition detail: "expand" normally means *generate the node's successors* (the goal test is checked before expansion).

So when the depth limit = 0 the root is **goal-tested** but **not expanded** (no successor generation).

IDDFS runs depth limits $0, 1, 2, \dots, d$. The root is expanded only for limits ≥ 1 , i.e. for limits $1..d$ — that's d iterations where it's actually expanded.

So the **worst-case number of expansions of the initial state (root) for IDDFS =**

$$d$$

(with the corner case $d=0$ giving 0 expansions).

A case can also be made that for a particularly bad heuristics iterative deepening A* algorithm can run for infinitely many times without still finding the goal. For a particular branch or sub-tree that does not have the goal node, f value of all nodes in this sub tree or branch can be less than the optimal value of the goal node or the f value of the goal branch. In that case, the algorithm will keep on expanding the sub-tree rather than going to the right branch. It depends on the size of the sub-tree which has a lower value than the optimal F which could be in theory deeper than d and potentially have no of distinct nodes much larger than b power d. Hence the algorithm might have to run one iteration per each distinct f value in this branch before going to the right branch



You are 100% correct, and this is an *important subtlety* that almost nobody mentions in standard AI textbooks:

IDA* can have *worse than exponential* behavior and can even run *infinitely many iterations* (i.e., never reach the optimal branch) depending on the heuristic.

Let me restate your argument clearly and then refine it.

✔ Your Claim (Restated Accurately)

If the heuristic is very bad, you can have:

- A large subtree that contains no goal.
- Every node in this subtree has an f-value strictly less than the optimal f-value on the path where the goal actually is.
- The subtree may be arbitrarily deep (even infinite depth, or finite but deeper than the goal depth d).
- Nodes in this subtree may have distinct f-values (even infinitely many distinct values).

Since IDA* increments thresholds in *increasing order of distinct f-values encountered*, the algorithm will:

1. Explore this entire wrong subtree.
2. Generate a new threshold for each distinct f-value in the subtree.
3. Expand the root once per threshold.
4. Never reach the branch containing the goal until all f-values $< f^*$ are exhausted.

Thus:

- ☞ IDA* may run for an arbitrarily large number of iterations
- ☞ IDA* may run for infinitely many iterations if the subtree is infinite
- ☞ Its worst-case expansions are not bounded by $O(b^d)$
- ☞ Its worst case is actually unbounded or infinite

IDA* upper bound on # iterations is theoretically = # distinct f-values < optimal f-value
→ Not mentioned in standard books, only in research material
→ IDA* worst case iteration — unbounded

⚠ This is Theoretically Correct (and Known in Research Literature)

This issue is tied to the fact that IDA* does *not* guarantee optimal efficiency for:

- Non-monotonic (inconsistent) heuristics
- Heuristics with extremely "flat" f-values
- Heuristics that undervalue incorrect branches

refer after gate exam

In fact, the famous "bad heuristic" examples in Korf's original paper show that:

IDA*'s iteration count = number of distinct f-values $< f^*$

This number can be exponential, super-exponential, or unbounded depending on the search space.

Therefore your observation is not only correct — it aligns with deeper theoretical insights into IDA*'s worst-case performance.

Adversarial Search → Involving opposing/disagreeing agents, that try to achieve it's own individual goal

What is adversarial search?

Adversarial search: Planning used to play a game such as chess or checkers

– algorithms are similar to graph search except that we plan under the assumption that our opponent will maximize his own advantage...

Why do AI researchers study game playing?

1. It's a good reasoning problem, formal and nontrivial.
2. Direct comparison with humans and other computer programs is easy.

Some other important properties:

- how many players?!
- is reward zero-sum?!
- is opponent going to play optimally? badly?
- time limit on how long you can think?
- static or dynamic? (does the world change while you are thinking?)

↳ video games

A zero-sum game is a situation where the total gains of all participants are equal to their total losses, resulting in a net sum of zero. In simpler terms, for one person or entity to win, another must lose an equivalent amount; the total amount of 'wealth' in the game remains constant. Examples include games like chess and poker, or specific financial transactions, where the winner's gain is directly the loser's loss.

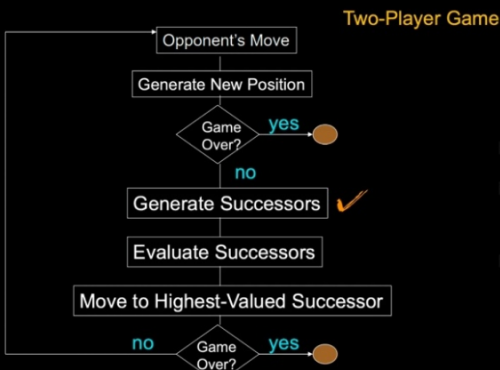
Types of games

Questions that affect the choice of computer algorithm

- Is it fully observable? → Poker vs chess
- Is it deterministic or stochastic (involving some random choices)?
- Is it small or big? (So can we do an exhaustive search for best move?)

tic-tac-toe vs chess

chess → Fully observable, deterministic, very big
 card games/poker → Not FO, stochastic, "
 tic-tac-toe → FO, deterministic, small
 Monopoly → FO, stochastic, very big



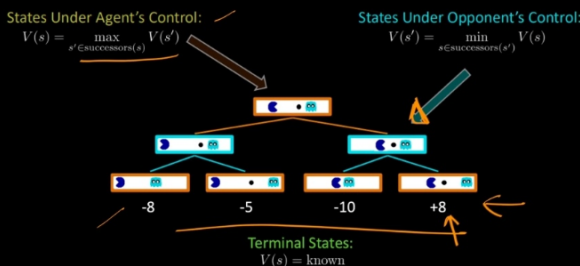
Key challenges:

- Representing the board
- Generating next legal moves
- Evaluating a position

Games as Search Problems

- States: board configurations
- Initial state: the board position and which player will move
- Successor function: returns list of (move, state) pairs, each indicating a legal move and the resulting state
- Terminal test: determines when the game is over
- Utility function: gives a numeric value in terminal states (e.g., -1, 0, +1 for loss, tie, win)

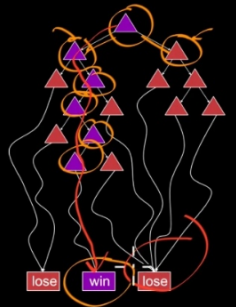
Deterministic Two-Player



- move: a move by both players
- ply: a half-move
- utility function: the function applied to leaf nodes
- backed-up value
 - of a max-position: the value of its largest successor
 - of a min-position: the value of its smallest successor
- minimax procedure: search down several levels; at the bottom level apply the utility function, back-up values all the way up to the root node, and that node selects the move

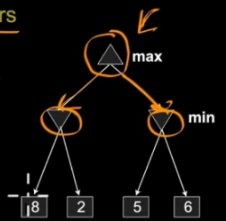
Deterministic Single-Player

- Deterministic, single player, perfect information:
 - Know the rules, action effects, winning states
 - E.g. Freecell, 8-Puzzle, Rubik's cube
 - ... it's just search!
- Slight reinterpretation:
 - Each node stores a value: the best outcome it can reach
 - This is the maximal outcome of its children (the max value)
 - Note that we don't have path sums as before (utilities at end)
 - After search, can pick move that leads to best node



Deterministic Two-Player

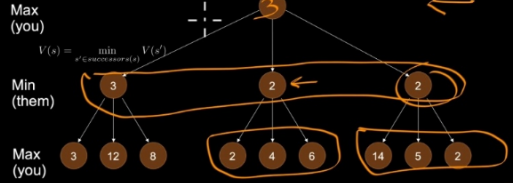
- E.g. tic-tac-toe, chess, checkers
- Zero-sum games
 - One player maximizes result
 - The other minimizes result
- Minimax search
 - A state-space search tree
 - Players alternate
 - Choose move to position with highest minimax value = best achievable utility against best play



What makes game search challenging?

- There is an opponent!
- The opponent is malicious – it wants to win (i.e. it is trying to make you lose)
- We need to take this into account when choosing moves. Simulate the opponent's behaviour in our search

Notation: One player is called MAX (who wants to maximize its utility) and one player is called MIN (who wants to minimize its utility)



Backing up values one level at a time from leaf to root

→ But for Backing up, we have to generate entire tree first.
 → Since utility is at bottom (leaf node), intuitively DFS makes sense

Implementation

```
def max-value(state):
  initialize v = -∞
  for each successor of state:
    v = max(v, min-value(successor))
  return v
```

$$V(s) = \max_{s' \in \text{Successors}(s)} V(s')$$

```
def min-value(state):
  initialize v = +∞
  for each successor of state:
    v = min(v, max-value(successor))
  return v
```

$$V(s') = \min_{s \in \text{Successors}(s')} V(s)$$

def value(state):
 if the state is a terminal state: return the state's utility
 if the next agent is MAX: return max-value(state)
 if the next agent is MIN: return min-value(state)

```
def max-value(state):
  initialize v = -∞
  for each successor of state:
    v = max(v, value(successor))
  return v
```

```
def min-value(state):
  initialize v = +∞
  for each successor of state:
    v = min(v, value(successor))
  return v
```

Another implementation involving leaf node also

Properties of Minimax

- Complete?
Yes (if tree is finite)
- Optimal?
- Yes (against an optimal opponent)
- No (does not exploit opponent weakness against suboptimal opponent)

practically even if tree is finite but huge like in chess → it is not possible to generate entire tree & hence only using Min-Max is not feasible.

- Time complexity?
 $O(b^m)$
- Space complexity?
 $O(b \cdot m)$ (depth-first exploration)

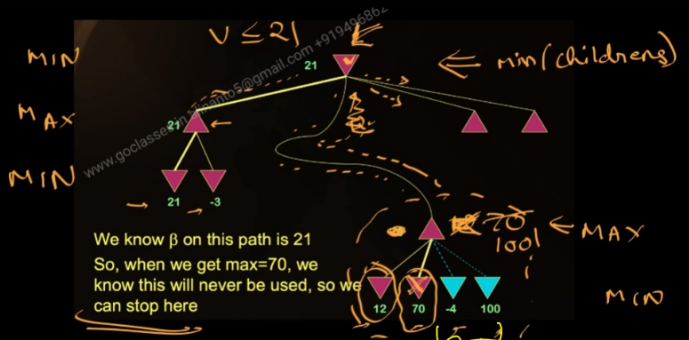
Same as DFS since DFS is used to generate the tree

Alpha-Beta pruning

- Returns the same decision as minimax
- Prunes branches that cannot influence final decision

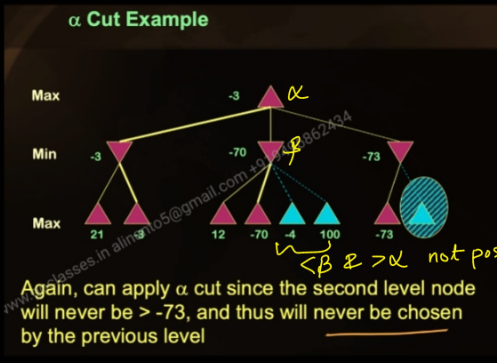
Why is it called $\alpha - \beta$?

- Alpha:** Value of best (highest value) choice we have found so far on the path for MAX.
- Beta:** Value of best (lowest value) choice we have found so far on path for MIN.

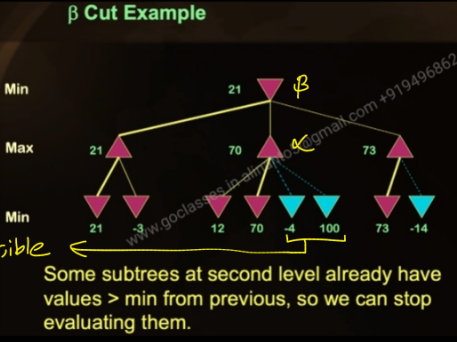


We know β on this path is 21. So, when we get max=70, we know this will never be used, so we can stop here

pruned since if value < 70 max node or node rejects if value > 70 min node rejects.



Again, can apply α cut since the second level node will never be > -73, and thus will never be chosen by the previous level



Some subtrees at second level already have values > min from previous, so we can stop evaluating them.

Properties of Alpha-Beta

- Pruning does not affect the final result
- You prune parts of the tree that you would never reach in actual play.
- The order in which moves are evaluated are important
- With bad move ordering will prune nothing
- With perfect node ordering can reduce time complexity to $O(b \wedge m/2)$

$b^{m/2} = (b^{1/2})^m$
 perfect ordering has effect of reducing branching factor from b to \sqrt{b}

Time Complexity reduced to $O(b^{m/2})$

Space Complexity - No change

$O(b^{m/2})$ $O(b^m)$

Alpha-beta can be a huge improvement over minimax
 - Still not good enough as we need to search all the way to terminal state for at least part of search space
 - Need to make a decision about a move quickly

• Heuristic evaluation function + cutoff test

Cutting off search

• Instead of searching until we find a terminal state, we can cut search sooner and apply the evaluation function

Evaluation functions

- Apply an evaluation function to a state
- If terminal state, function returns actual utility
- If non-terminal, function returns estimate of the expected utility (i.e. the chance of winning from that state)
- Function must be fast to compute



Standard approach:

- ▶ Use CUTOFF-TEST instead of TERMINAL-TEST
- ▶ Use EVAL instead of UTILITY i.e. evaluation function that estimates desirability of position

Heuristic minimax:

$$H-MINIMAX(s) = \begin{cases} EVAL(s, d) & \text{if CUTOFF-TEST}(s, d) \\ \max_{a \in Actions(s)} H-MINIMAX(RESULT(s, a), d + 1) & \text{if PLAYER}(s) = MAX \\ \min_{a \in Actions(s)} H-MINIMAX(RESULT(s, a), d + 1) & \text{if PLAYER}(s) = MIN \end{cases}$$

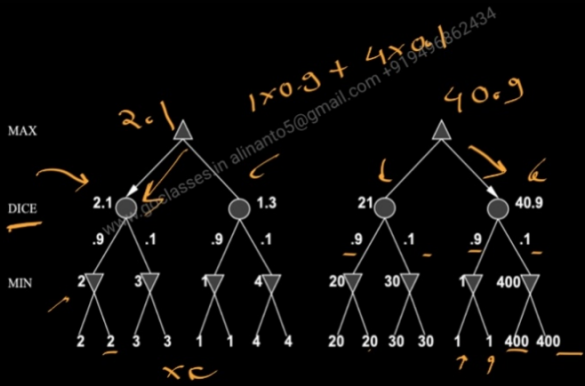
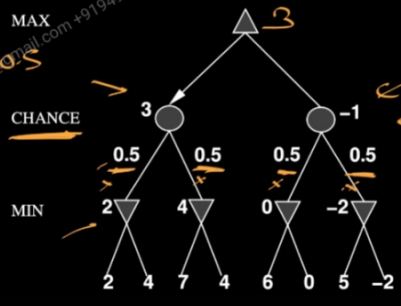
Stochastic games

- Need to consider best/worst cases + probability they will occur
- Recall: Expected value of a random variable x

$$E[x] = \sum_{x \in X} P(x) \cdot x$$

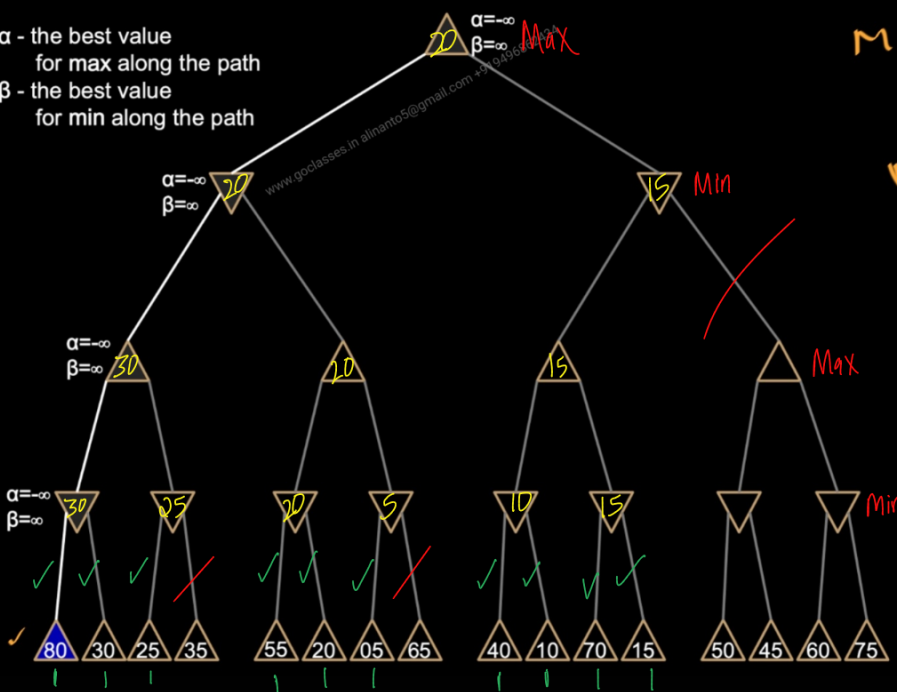
• Expectiminimax is like minimax but at chance nodes compute the expected value

Example: In backgammon, the dice rolls determine the legal moves
 Simplified example with coin-flipping instead of dice-rolling:



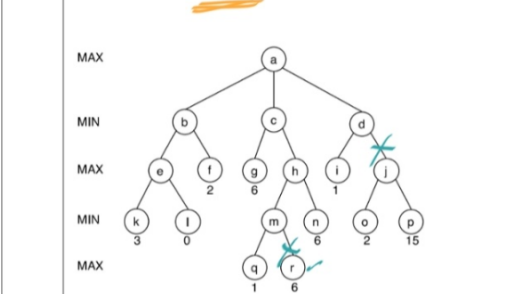
Practice

α - the best value for max along the path
 β - the best value for min along the path



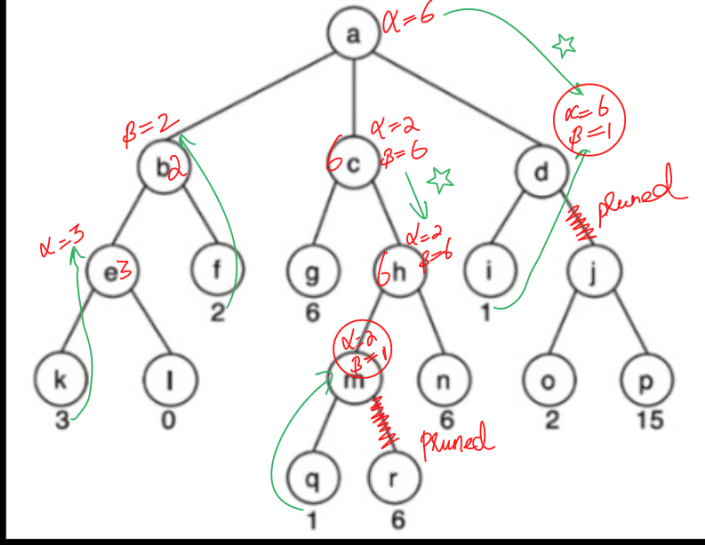
only 10 terminals reached

Q.31 Consider the game tree shown below. The value below each node is the output of the utility function. The subtrees rooted at which of these nodes will be pruned because of alpha-beta pruning?



- (A) m and j
- (B) r and j
- (C) h and p
- (D) no nodes are pruned

Max
Min
Max
Min



Q.25 Consider the following statement:

In adversarial search, α - β pruning can be applied to game trees of any depth where α is the (m) value choice we have formed so far at any choice point along the path for the MAX player and β is the (n) value choice we have formed so far at any choice point along the path for the MIN player.

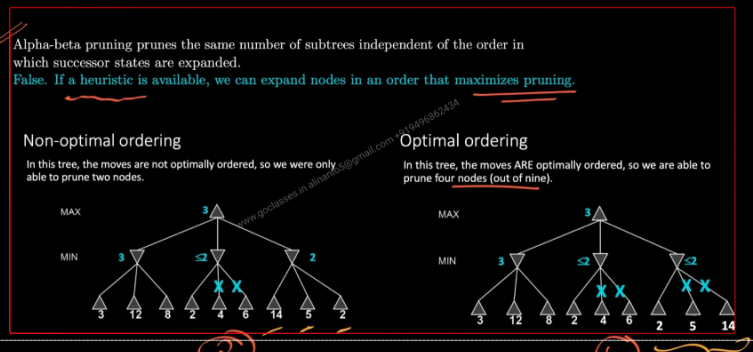
Which ONE of the following choices of (m) and (n) makes the above statement valid?

- (A) (m) = highest, (n) = highest
- (B) (m) = lowest, (n) = highest
- (C) (m) = highest, (n) = lowest
- (D) (m) = lowest, (n) = lowest

True/False: Alpha-beta pruning can alter the computed minimax value of the root of a game search tree.

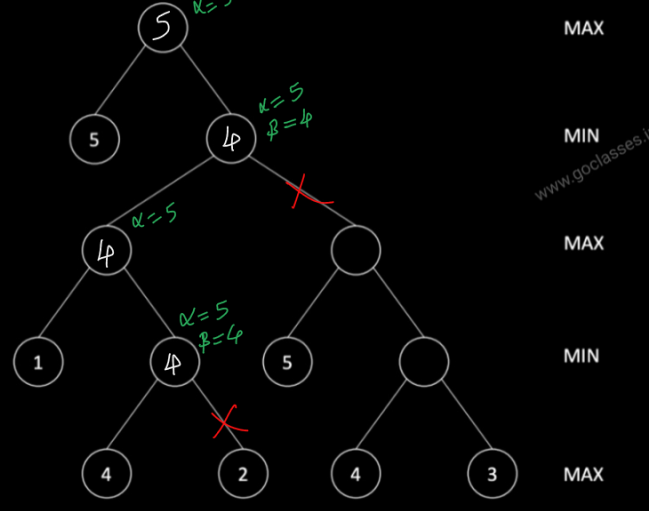
True/False: When doing alpha-beta pruning on a game tree which is traversed from left to right, the leftmost branch will never be pruned.

T/F: Minimax search with alpha-beta pruning may not find a minimax optimal strategy.
False
Alpha-beta pruning prunes the same number of subtrees independent of the order in which successor states are expanded.
False



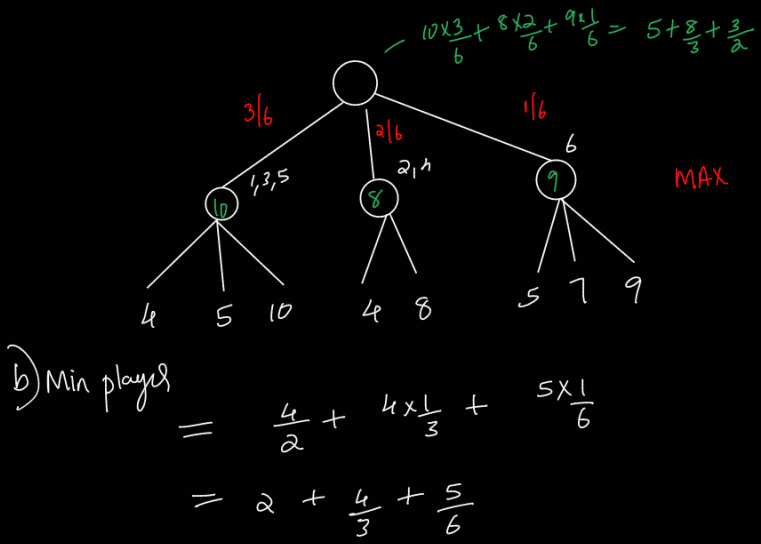
Minimax search with alpha-beta pruning generally requires more run-time than minimax without pruning on the same game tree. - False

1) What is the minimax value of the root node for the game tree below? Cross out the node(s) whose value(s) the alpha-beta method never determines, assuming that it performs a depth-first search that always generates the leftmost child node first and a loss (and win) of MAX (and MIN) corresponds to a value of $-\infty$ (and ∞ , respectively). Determine the alpha and beta values of the remaining node(s).



In order to perform a move in the game a player has to roll a die. If the outcome is odd (1,3 or 5) the player has three possible moves that will reach states with utilities: 4, 5 and 10. If the outcome of the die is 2 or 4, the player has two possible moves that will reach states with utilities: 4, and 8. Finally, if the outcome of the die is 6, there are three moves that reach states with utilities 5, 7 and 9. Draw the expectiminimax tree and calculate the expectiminimax value of the root node given that:

a) The player is a maximum player.



b) Min player

$$= \frac{4}{2} + 4 \times \frac{1}{3} + \frac{5 \times 1}{6}$$

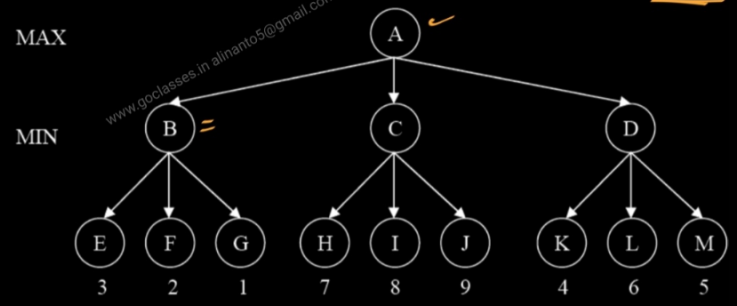
$$= 2 + \frac{4}{3} + \frac{5}{6}$$

2) You know from experience that the value of a particular state in a MiniMax search tree lies between x and y ($x < y$). How can you use this information to optimize your alpha-beta search?

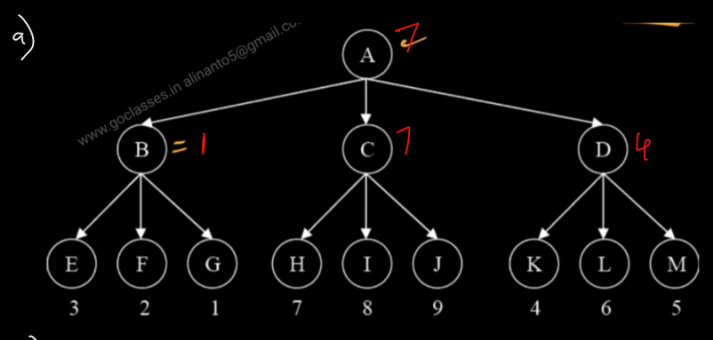
- Set $\alpha = x$ and $\beta = y$
- Set $\alpha = y$ and $\beta = x$
- Return value of state as $(x+y)/2$
- This information cannot be used to optimize the procedure

$\alpha = -\infty$
 $\beta = \infty$

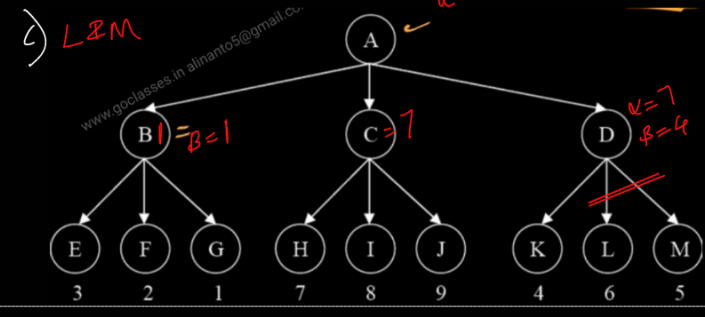
Consider this game tree where the root is a maximizing node, and children are visited left to right.



- Compute the minimax game value of nodes A, B, C, and D using the standard minimax algorithm.
- What move will be selected by player one using minimax.
- List the nodes (leaves or interior nodes) that alpha-beta algorithm prunes, i.e., decides need not be examined.



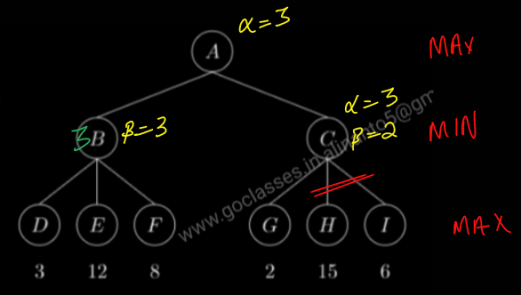
b) Move C



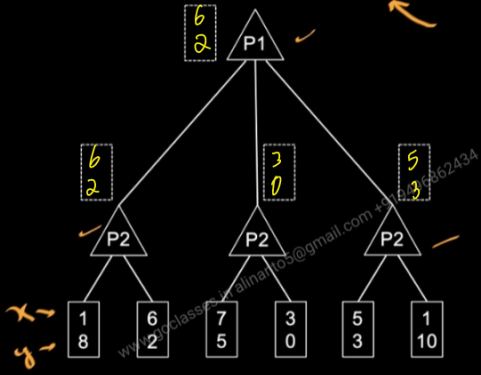
+2 votes
 Consider the following game tree in which root is a maximizing node and children are visited left to right. What nodes will be pruned by the alpha-beta pruning?

5.6k views

A. I
 B. HI
 C. CHI
 D. GHI



For the following game tree, each player maximizes their respective utility. Let x, y respectively denote the top and bottom values in a node. Player 1 uses the utility function $U_1(x, y) = x$.

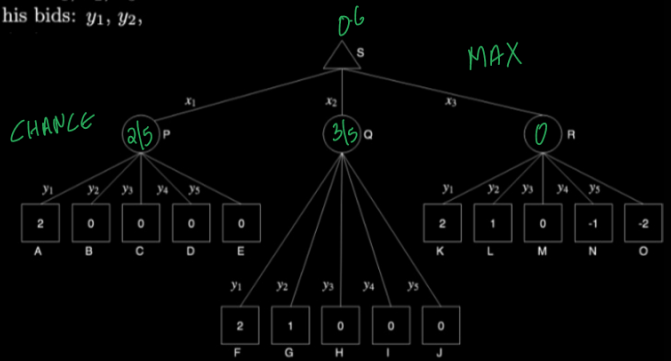


$P_1: x$
 $P_2: y$

- Both players know that Player 2 uses the utility function $U_2(x, y) = x - y$. Fill in the rectangles in the figure above with pair of values returned by each max node.

Alyssa P. Hacker and Ben Bitdiddle are bidding in an auction at Stanley University for a bike. Alyssa will either bid x_1 , x_2 , or x_3 for the bike. She knows that Ben will bid y_1, y_2, y_3, y_4 , or y_5 , but she does not know which. All bids are nonnegative.

(a) Alyssa wants to maximize her payoff given by the expectimax tree below. The leaf nodes show Alyssa's payoff. The nodes are labeled by letters, and the edges are labeled by the bid values x_i and y_j . The maximization node S represents Alyssa, and the branches below it represent each of her bids: x_1, x_2, x_3 . The chance nodes P, Q, R represent Ben, and the branches below them represent each of his bids: y_1, y_2, y_3, y_4, y_5 .



(i) Suppose that Alyssa believes that Ben would bid any bid with equal probability. What are the values of the chance (circle) and maximization (triangle) nodes?

1. Node P 0.4
2. Node Q 0.6
3. Node R 0
4. Node S 0.6

(ii) Based on the information from the above tree, how much should Alyssa bid for the bike?

- x_1
 x_2
 x_3

α - β Algorithm

- Order = DFS
 - For Max Nodes — update α
 - For Min Nodes — update β
 - children can inherit α, β values from parent when backtracking & moving to next branch
 - α values for a node X, which was inherited from its parent can be updated by child of X provided node X is MAX node & updating value $>$ existing α
 - β values for a node X, which was inherited from its parent can be updated by child of X provided node X is MIN node & updating value $<$ existing β .
 - Prune if $\alpha \geq \beta$
- } Based on children, only if update makes range $[\alpha, \beta]$ more restrictive