



Contents

Unit II – Heuristic Search Techniques

1. Issues in the Design of Search Programs
2. Heuristic Search
3. Generate-and-Test Strategy and its applications
4. *Hill Climbing Algorithm* and its variants
5. *Best-First Search* Algorithm
6. *A* Algorithm*
7. *Applications of A* Algorithm*
8. Problem Reduction
9. *AO* Algorithm* for Problem Reduction
10. Constraint Satisfaction Problems (CSPs)
11. Means-Ends Analysis

1. Issues in Search Program Design

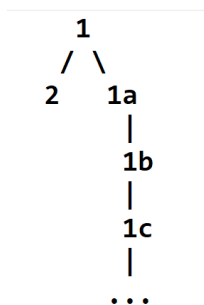
Key Considerations:

- **Completeness:** Will the algorithm find a solution if one exists?
- **Optimality:** Will it find the best solution?
- **Time Complexity:** How long does it take?
- **Space Complexity:** How much memory is needed?

1. Completeness

- **Definition:** Completeness asks **whether the search algorithm is guaranteed to find a solution if one exists.**
- **Why it matters:** If an algorithm is not complete, it might keep **searching indefinitely** even if a solution exists.
- **Examples:**
 - **Breadth-First Search (BFS):** Complete for finite graphs (it will eventually find a solution).
 - **Depth-First Search (DFS):** Not complete in infinite or cyclic graphs (it can get stuck going down one infinite path). If the graph has **cycles** or is **infinite**, DFS might keep going in circles and never return to explore other nodes.

Infinite Branching Tree:



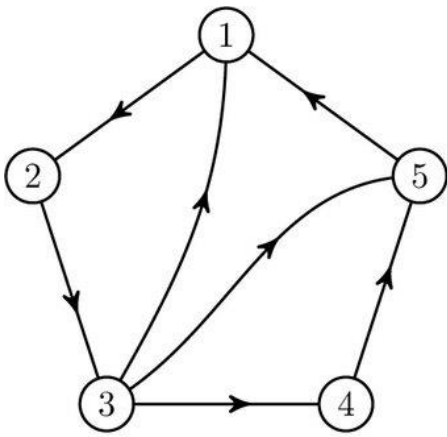
From 1, DFS chooses the right branch 1a.

- Then it sees 1b, then 1c, then 1d ... and so on.
- This branch is **infinite**, meaning DFS never "hits the bottom" to backtrack.

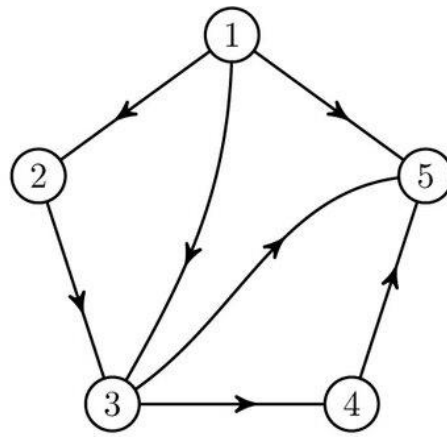
So **DFS gets trapped**, never exploring the left child 2, even though the solution was right there.

- **Key consideration:** You must know whether your **search space is finite or infinite** before choosing the algorithm.

A **cyclic graph** has cycles meaning you can start from some node and follow a path such that you arrive at the same node you began.

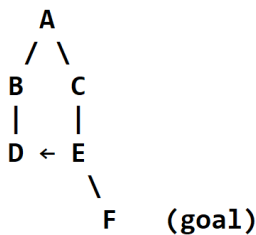


(a) A directed cyclic graph.



(b) A directed acyclic graph.

We know that BFS explores the search space **level by level**, starting from the initial state. It checks all immediate neighbors first before going deeper.



Let us consider the above Graph:

DFS without cycle checking:

1. Start at **A**
2. Go left → **B**
3. From B → **D**
4. From D → **E**
5. From E → back to **B** (cycle starts: B → D → E → B → ...)

DFS gets **stuck in the loop** and never explores F.

Key Point:

- Without cycle checking, DFS can loop forever.
- With cycle checking, DFS is fine on finite graphs.

Real-life example of BFS:

Imagine you are in a **building trying to find an exit**:

1. You start at your current room.
2. You first check all rooms **directly connected** to your room.
3. If the exit isn't there, you check rooms connected to those rooms (second level), and so on.
4. Because you are checking **all rooms level by level**, you are guaranteed to find the exit eventually if the building has a finite number of rooms.

Key point: BFS will not get lost in a loop and will find a solution if it exists.

Unlike DFS, BFS **won't get stuck down a single infinite branch**, because it explores all nodes at a given depth before going deeper.

Real-life example of DFS:

Imagine you are exploring a **maze with infinite tunnels**:

1. You pick a path and keep going forward without checking alternatives.
 2. If this path goes infinitely or loops back on itself, you will **never reach the exit**, even if it exists in another branch.
- In a finite maze, DFS will eventually find the exit, but in an infinite maze or one with cycles, DFS **might never come back to explore other paths**.

Key point: DFS can fail in infinite or cyclic search spaces.

Imagine the situation

You are in a huge (possibly infinite) city, trying to find your friend's house. You don't have a map — you only know that your friend is *somewhere in the city*.

DFS in the City

- With **Depth-First Search (DFS)**, your strategy is:
 - Pick one street and keep going **straight ahead**.
 - If the city is infinite (like an endless highway), you might **walk forever** in one direction, never turning back to check other streets.
 - Even if your friend's house is just a block behind you, you may **never see it**, because you're lost in a single infinite path.

Problem: DFS is not *complete* in infinite spaces — you may never find your friend, even if they exist.

Key Consideration

- Before choosing a search algorithm, **know whether the search space is finite or infinite**.
 - **Finite graph:** BFS and DFS both can work; BFS guarantees the shortest path, however occupies more memory.
 - **Infinite graph:** BFS is better if you want completeness; DFS may get stuck.

2. Optimality

- **Definition:** Optimality asks **whether the search algorithm finds the best solution**, typically meaning the solution with the least cost (e.g., shortest path, minimum time, lowest energy).
- **Why it matters:** In many applications, finding any solution is not enough—you want the best one.
- **Example:**

- **DFS/BFS:** BFS is optimal if all steps cost the same; DFS is generally not optimal.
- **Key consideration:** If you care about quality of the solution, you need an algorithm that guarantees optimality.

3. Time Complexity

- **Definition:** Time complexity measures **how long the algorithm takes to run** relative to the size of the search space.
- **Why it matters:** Some algorithms may take impractically long for large problems.

Amazon Delivery Plan as a Search Problem

- **Initial state:** Products in Amazon warehouse.
- **Goal state:** Products delivered to customer.
- **Search:** Choosing routes and steps (delivery vehicles, hubs, stops).



At Bengaluru



At Hyderabad



At Visakhapatnam



Logistics



Loca Distribution Hub

The **time complexity** here means:

How many possible routes or delivery plans Amazon needs to consider before finding the correct/optimal one.

1. Branching Factor (b)

At each step in planning, Amazon can choose between many routes:

- Different courier partners (BlueDart, Amazon Logistics, India Post, etc.).
- Different distribution hubs.
- Different delivery vans.

If there are b choices at each step, that's the branching factor.

By definition, **branching factor** is the *average number of successors (child nodes)* from any node in the search tree/graph.

In theory, we often assume a **constant branching factor** (say, b) for simplicity when analyzing **time/space complexity**.

But in practice?

- Each node **can have a different number of successors**.
- Example (Amazon analogy):
 - From a **central warehouse**, there may be 5 possible hubs (branching factor = 5).
 - From a **local hub**, there may be only 2 possible delivery vans (branching factor = 2).
 - From the **last-mile delivery van**, there's only 1 customer (branching factor = 1).

So, **branching factor varies per node** in real-world problems.

How do we handle this in analysis?

- To simplify, we usually take the **maximum branching factor** or **average branching factor** and call it b .
- This helps us express **time complexity** like $O(b^d)$, even though not every node has exactly b children.

2. Depth of Solution (d)

Depth = number of steps needed to deliver.

Example:

Warehouse \rightarrow City Hub \rightarrow Local Hub \rightarrow Customer = depth 3.

If the delivery requires d steps, then the number of routes explored depends on both b and d .

Breadth-First Search (BFS)

- Amazon checks all possible first-step hubs, then all possible second-step hubs, and so on.
- Worst case, time complexity = $O(b^d)$.
- **Analogy:** Amazon tries all nearby hubs first before moving deeper. Guaranteed to find the shortest delivery plan, but it takes a lot of checking if the network is huge.

Depth-First Search (DFS)

- Amazon picks one possible delivery route and follows it until the end.
- Worst case, time complexity = $O(b^m)$, where m = maximum depth of the delivery network.
- **Analogy:** Amazon picks one route (say: Warehouse \rightarrow Hub A \rightarrow Hub B \rightarrow ...) and keeps going. If wrong, it backtracks. If the network is very deep, it may waste a lot of time.

Suppose Amazon is planning deliveries.

- Branching factor (b) = 3 (on average, 3 choices at each hub).
- Depth of actual delivery path (d) = 4 (it takes 4 steps to reach customer).
- But maximum depth of the network (m) = 10 (some routes are 10 steps long, even if the customer is closer).

Then:

- BFS: $O(b^d) = O(3^4) = O(81) \rightarrow$ will find the customer after checking ~ 81 possibilities.

- DFS: $O(b^m) = O(3^{10}) = O(59,049) \rightarrow$ may go unnecessarily deep, exploring up to 59,049 routes before backtracking.

4. Space Complexity

- **Definition:** Space complexity measures how much memory the algorithm uses.
- **Why it matters:** Some algorithms may run out of memory before they finish.

Amazon Delivery Example

Think of Amazon **planning deliveries** in advance:

Breadth-First Search (BFS)

- Amazon must keep track of all delivery routes at the current level before moving deeper.
- Example: From the warehouse, there are 3 hubs. From each hub, 3 sub-hubs. Very quickly, the number of routes grows large.
- **Memory requirement** = $O(b^d)$ (**can be huge**).
- Analogy: Amazon writes down *all possible delivery options* on paper for every city, every hub, every van — and keeps them all until the delivery is finalized. This takes a lot of memory.

Think of Amazon delivery planning like BFS:

1. **Branching Factor (b):**
Each hub \rightarrow connects to many sub-hubs, each sub-hub \rightarrow many trucks, each truck \rightarrow many delivery routes.
2. **Depth (d):**
How many "steps" from the warehouse to the customer (Warehouse \rightarrow City Hub \rightarrow State Hub \rightarrow Regional Hub \rightarrow Local Station \rightarrow Customer).
3. **What BFS does:**
 - Amazon's software **writes down every possible route** from Bangalore to every possible city and every possible neighbourhood.
 - It doesn't throw away old routes — it **keeps all routes in memory** until it finds the customer's exact address.

So, If BFS method is used, Amazon's software keeps track of *all vans, hubs, routes at once*, which makes memory huge. That's why BFS is often impractical for large-scale real-world problems like nationwide delivery planning.

Depth-First Search (DFS)

- Amazon only keeps track of one delivery route at a time (the path it is currently following).
- **Memory requirement** = $O(b \cdot m)$ (**much smaller**).
- Analogy: Amazon says: *"Let's try this one route first. If it doesn't work, erase it and try another."*

Example:

Amazon picks **one delivery van and one path**, and says:

"Let's follow this van all the way from Bangalore \rightarrow Hyderabad \rightarrow Vizag \rightarrow MVP Colony."

- If that route fails, Amazon **backtracks** and tries another path.
- At any point, Amazon is only keeping track of:

- Current route (one van's journey)
- Depth (how many hubs/stops so far)
- Much less memory, because it's not storing *all* routes at once — just the current one.

2. Heuristic Search (Informed Search)

What is a Heuristic?

- A "rule of thumb" that guides search
- Provides estimate of cost to reach goal
- Not guaranteed to be perfect, but usually helpful

Imagine this Situation:

You are in a **big library** looking for your favourite comic book.

1. Blind Search (No heuristic)

You don't know where comics are.

- You start checking **every shelf, one by one**.
- This takes a long time because you might check **all books** before finding the comic.

This is like **BFS/DFS** – no clue where the answer is, just brute force.

2. Heuristic Search (With a hint/guide)

Now, your friend tells you:

"Comics are usually kept in the kids' section on the 2nd floor."

- Instead of checking everywhere, you **go straight to that section first**.
- You might find your comic much faster.

This "**hint**" is called a **heuristic**.

It doesn't guarantee the *perfect* solution, but it **guides you toward the goal faster**.

So, in short:

Heuristic Search = **Searching with clues** instead of searching blindly.

Properties of a Good Heuristics

A good heuristic is one that helps the algorithm find the shortest path **by exploring the least number of nodes possible**.

- **Admissibility:** Never overestimates the actual cost.

It's like asking your friend, "*How far is the shop?*"

If they say "**5–10 minutes**" when it's actually 10 → okay, you'll not miss the shortest path.

If they say "**30 minutes**" when it's only 10 → you may avoid that shop and miss the best option.

- **Efficiency:** Heuristic must be **fast to calculate (don't waste time guessing)**.

Like using **Google Maps ETA** (Estimated Time of Arrival) → quick guess of travel time.
Instead of asking 100 people along the way and averaging their answers → wasteful!

- **Consistency:** Satisfies/follows triangle inequality.

In a triangle, the direct side is always \leq sum of the other two sides. Similarly, the heuristic estimate must be no greater than “step cost + neighbour’s heuristic.”

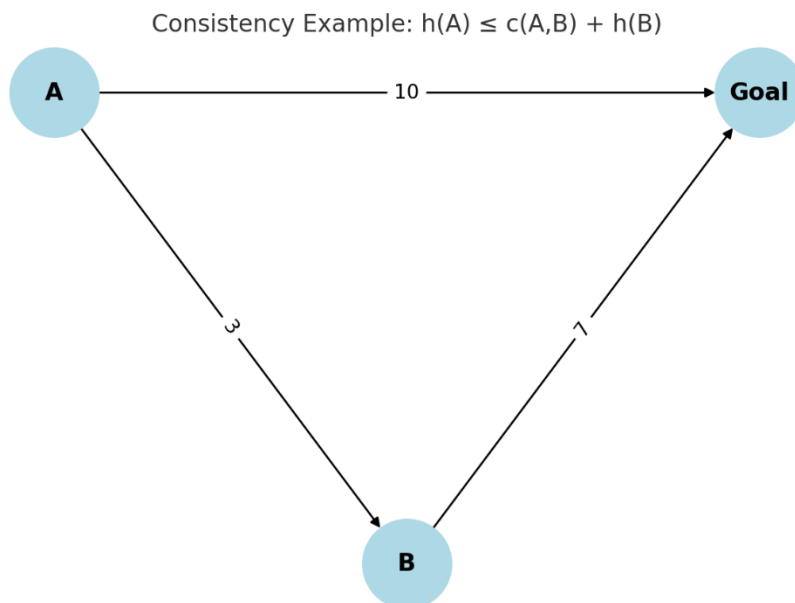
- **Informative:** Provides useful guidance/hints, not random guesses.
Like asking a local for directions:

“Take the highway, it’s faster.” (good guidance)

“Just keep walking, you’ll eventually get there.” (useless!)

Explaining “Consistency” with Map Navigation example:

Example1:



Ex: Pizza Delivery

- If delivery from your **house (A)** to the **pizza shop (goal)** is estimated at **10 minutes**,
- and your friend’s house (B) is **3 minutes away** from you, and from B the estimate is **7 minutes**,
- then your estimate should not say **more than 3 + 7 = 10**.

Otherwise, it’s like saying “From my house it takes 15 minutes, but from my friend’s house (closer) it takes 7 minutes.” That makes no sense.

Example2:

Suppose you want to estimate distance from **City A → Goal City G**:

- **Heuristic** estimate $h(A)$ = straight-line distance from A to G = 120 km
- Neighbour B is on the way:
 - Actual step **cost** from A to B (c) = 50 km

- Estimate from B to G ($h(B)$) = 60 km

Check consistency:

$$h(A) \leq c(A,B) + h(B)$$

$$120 \leq 50 + 60$$

$$120 \leq 110 \quad \text{Incorrect}$$

So, this heuristic is **not consistent** (it overestimated).

If instead:

$$h(A) = 100 \text{ km}$$

$$\text{Then: } 100 \leq 50 + 60 = 110 \text{ Correct} \rightarrow \textbf{Consistent}$$

A **consistent heuristic** is one that never “skips reality” — it respects the triangle inequality, meaning **direct estimate \leq step cost + next estimate**.

What is a Heuristic Search

Definition: Uses **extra knowledge (heuristics)** to guess which path is promising.

Analogy: Using a flashlight or hints while searching in the dark.

Some popular Heuristic Algorithms

(a) Greedy Best-First Search

- Picks the node that **looks closest to the goal**.
- Fast but not always optimal.

Example: Always choosing the road pointing towards your destination, even if longer overall.

(b) Hill Climbing Algorithm

- Always moves to the **neighbour with best score**.
- Can get stuck at local maxima (not global solution).

Example: Climbing a mountain in fog by always stepping uphill — might end on a smaller peak.

(c) A* Algorithm (A* Search Algorithm)

- Combines cost so far (g) + estimated cost to goal (h).
- **Optimal and efficient** (if heuristic is good).

Example: GPS navigation (considers distance + traffic).

(d) AO* Algorithm

- Works on **AND-OR graphs** (some problems need multiple sub-goals together, not just one choice).
- Uses heuristics to decide which subproblems to solve first.
- Finds the **best solution graph**, not just a single path.

Example:

A robot assembling a table:

- To complete the table, it must attach **all 4 legs (AND)**.
- But for each leg, it can either **screw it on or glue it (OR)**.

2. Generate-and-Test Strategy and its applications

(It is like **trial** and **error**)

Algorithm Steps:

1. Generate a possible solution
2. Test if it meets the goal
3. If yes, stop; if no, generate another solution
4. Repeat until solution found

It's the **simplest form of problem solving**:

Keep generating possible solutions, then test each one until you find the correct answer.

Example: Brute-force **password cracking** → try "1234", "1235", ... until success.

- If we **combine heuristics** with Generate-and-Test, then the generation step is **not purely random**.
- Instead of generating *all possible guesses*, we generate **promising guesses first**. (e.g., "Somebody usually uses his/her birth date year in some order"), you can **narrow down guesses** instead of trying everything blindly.

Applications

- **Puzzle solving:** (8-Puzzle), Rubix cube - repeat until success
- **Cryptography/ Password Cracking:** repeat until success
- **Optimization problems:**

Eg: Hyperparameter Tuning in Machine Learning:

Problem: Choosing the best parameters for a model (e.g., learning rate, number of layers).

Generate: Random parameter settings.

Test: Train the model → Check accuracy.

Repeat: Keep improving until best accuracy achieved.

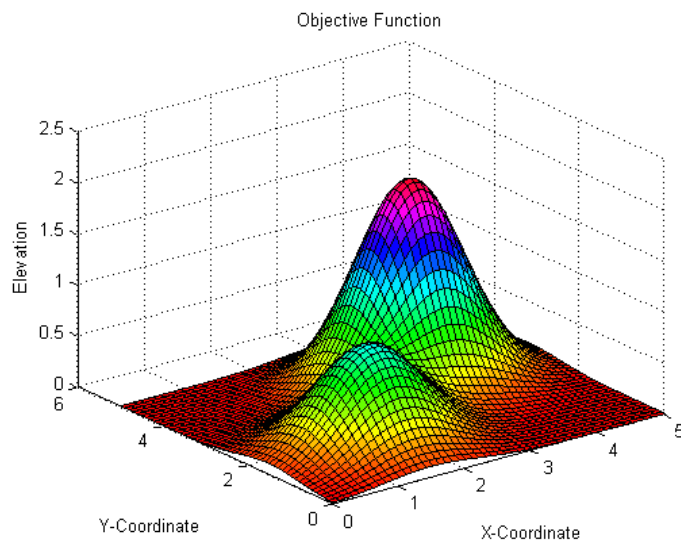
- **Game Playing AI** – Eg: Chess: Test → does this move lead to a win or good position? Select best move.
- **Robot Pathfinding** (generate possible moves – left, right, forward and test does it reach the goal without obstacles, repeat until success)

Step-by-step (trial and error)

- **Generate:** Robot moves forward.
 - **Test:** It hits a wall → Not good.
- **Generate:** Robot turns right, then forward.

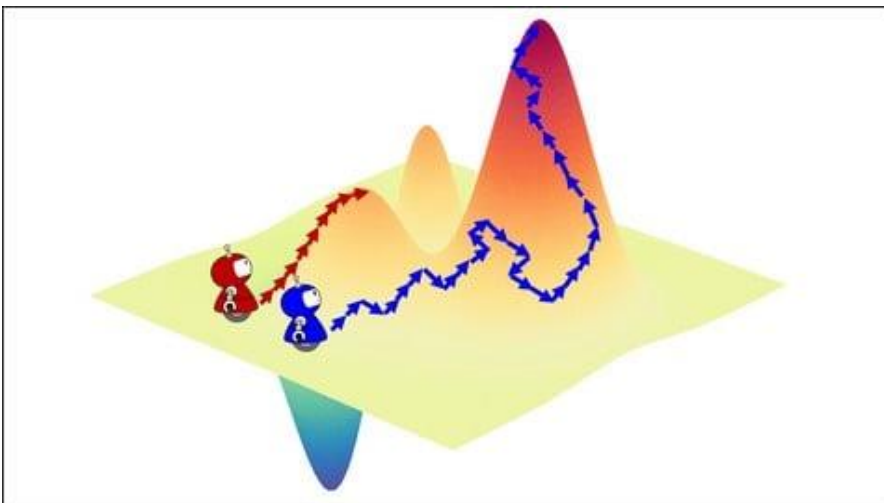
- **Test:** Path is clear → Continue.
- Keep generating and testing until the Goal is reached.

4. Hill Climbing Algorithm



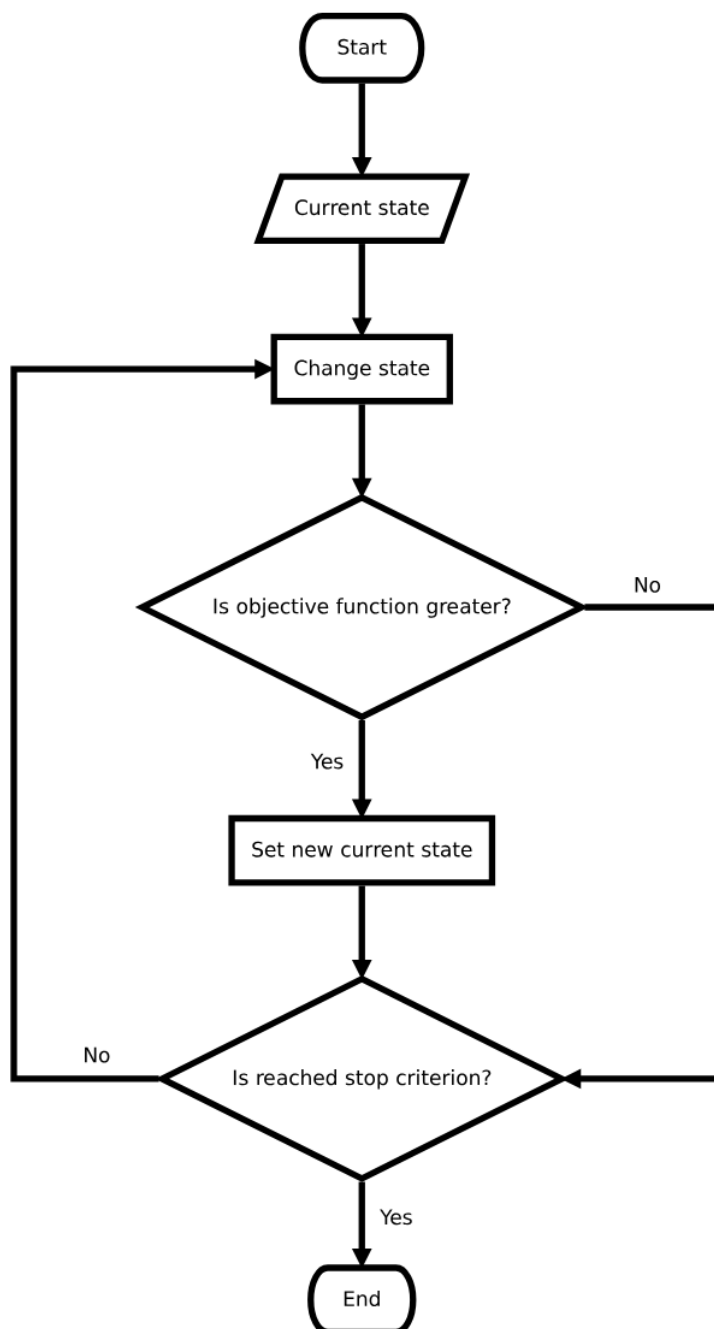
Note:

If the function on Y-axis is **cost** then, the goal of search is to find the **global minimum and local minimum**. If the function of Y-axis is **Objective** function, then the goal of the search is to find the **global maximum and local maximum**.



Basic Hill Climbing Algorithm:

1. Start with current state
2. Generate neighbors
3. Move to best neighbour
4. Repeat until no better neighbour exists



Example: Imagine you are standing in a **hilly area at night with fog**.

- You can't see the whole mountain.
- You only see the **ground just around you**.
- Your goal: **Reach the top of the highest hill**.

Features of Hill Climbing Algorithm

- **Generate and Test variant:** Hill Climbing is the variant of **Generate and Test** method. The Generate and Test method produce **feedback** which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

How it Works (Step by Step)

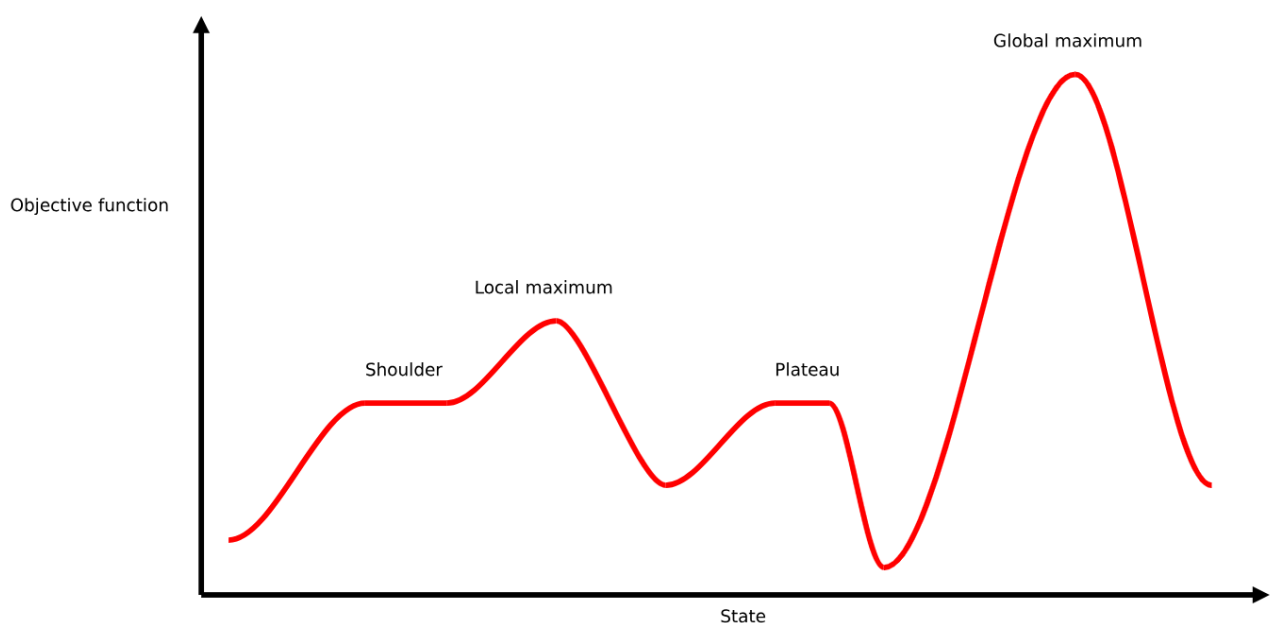
1. **Look Around**
 - At your current position, check which direction looks **higher**.
2. **Take a Step Up**
 - Move in the direction that increases height (improvement).
3. **Repeat**
 - Keep climbing higher and higher until you **can't go up anymore**.
4. **Stop**
 - When no neighbour is higher, you stop → this is your **solution**.

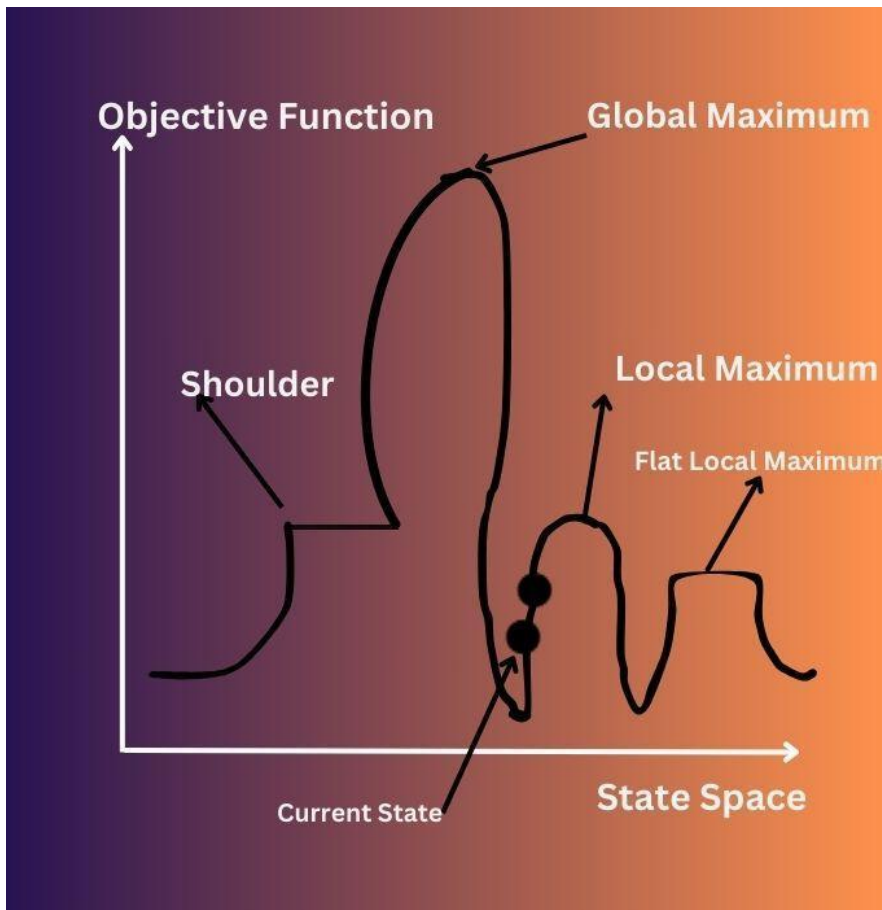
Why it's called "Heuristic Search"?

- Because you use a **heuristic (a guess/estimate)** to decide: "Which direction looks better?"
- You don't explore all possible paths (like BFS/DFS).
- Instead, you **greedily choose the best next step**.

Problems with Hill Climbing:






- **Local Maxima:** Sometimes you stop at a local maximum (a small peak), not the global maximum (tallest peak). That's why it is also known as Local search algorithm. Like climbing a small hill and thinking you're at the top of the world, while the real mountain is further away.
- **Plateaus:** Flat areas with no improvement
- **Ridges:** Difficult to navigate narrow paths to solution.





A ridge is a sequence of local maxima that are not directly adjacent. It is not one smooth peak, but a **sequence of small peaks (local maxima)** that are separated by tiny dips.

In this diagram: Peak → Valley → Peak → Valley → Peak
This satisfies the condition, since there is a local minimum between the maxima.

Think of it as: up  – down  – up  – down  – up  ...

A ridge is like a series of small bumps. Hill climbing gets stuck on the first bump because it only checks next steps and won't go down even a little, so it misses higher peaks further ahead.

Think of climbing a staircase that is slanted diagonally. If you only try to go “straight up,” you'll hit a wall. But if you take a **diagonal step**, combining a bit of sideways + upwards movement, you'll keep climbing.

Plateau

- A **plateau** is a large **flat area** in the search space.
- All neighbouring states have the **same heuristic value** (no improvement, no decline).
- The algorithm **can't see any direction as "better"**, so it may **wander aimlessly** or stop.

Shoulder

- A **shoulder** is a **flat area at the top of a slope**, but from there you can eventually find a **higher slope** if you **keep exploring**.
- The algorithm may think it's at the top (local maximum), but really, there's still a way up.
- The problem is the algorithm might **prematurely stop** at the shoulder.

Note: "The main advantage of the Hill Climbing Algorithm is its low space complexity, since it only stores the current state (and its value), unlike other algorithms that must keep track of multiple paths or nodes."

Example:

- Imagine you're climbing a hill in fog.
- You don't need to remember every path you tried.
- You only need to know:
 - *"Where am I right now?"*
 - *"What's the best direction from here?"*



That's why hill climbing has very **low space complexity**.

Hill Climbing algorithm – step by step:

1. **Choose a start state** (pick a random position on the hill).
2. **Evaluate the current state** (how "high" is it? — this is the objective/score).
3. **Generate neighbors** of the current state (the small moves you can make from here).
4. **Pick a neighbour that has a better score** than the current state.

- In *simple hill climbing*: pick the **first** neighbor that is better.
 - In *steepest-ascent hill climbing*: examine all neighbors and pick the **best** one.
5. **Move to that neighbour** and make it the current state.
 6. **Repeat** steps 2–5 until no neighbor is better than the current state (i.e., you are at a peak).
 7. **Stop** and return the current state as the solution (a local maximum).

Termination conditions (use one or more):

- No neighbour has a better score (local maximum or plateau).
- Maximum number of steps reached.
- Time limit reached.

Hill Climbing Variants:

- Simple Hill Climbing
- First-Choice Hill Climbing
- Steepest Ascent Hill Climbing
- Random Restart Hill Climbing
- Stochastic Hill Climbing

Simple Hill Climbing

- You check neighbors one by one (in some order).
- The first better one you find, you immediately move there.
- If the first neighbour is not better, you check the next neighbour, and so on.

First-Choice Hill Climbing

- Instead of checking neighbors in order, you **pick random neighbors**.
- If the random neighbour is better, you move there (it need not be the tallest).
- If not, you randomly pick another neighbour.
- This approach is efficient, especially when a state has a large number of neighbors, because it doesn't waste time evaluating every single option.

Difference between the above two:

- **Simple Hill Climbing** = systematic neighbour checking (orderly).
- **First-Choice Hill Climbing** = random neighbour checking (faster in large spaces).

Steepest-Ascent Hill Climbing

- At each step: look at **all neighbors**.
- Move to the **best (steepest upward) neighbour**.
- Slower than simple hill climbing, but usually finds better solutions.

Random-Restart Hill Climbing

- Run hill climbing multiple times from different random starting points.
- Helps **avoid getting stuck at local maxima** (small peaks).
- Eventually, one run should find the global maximum. (The best solution found across all runs is the final answer.)

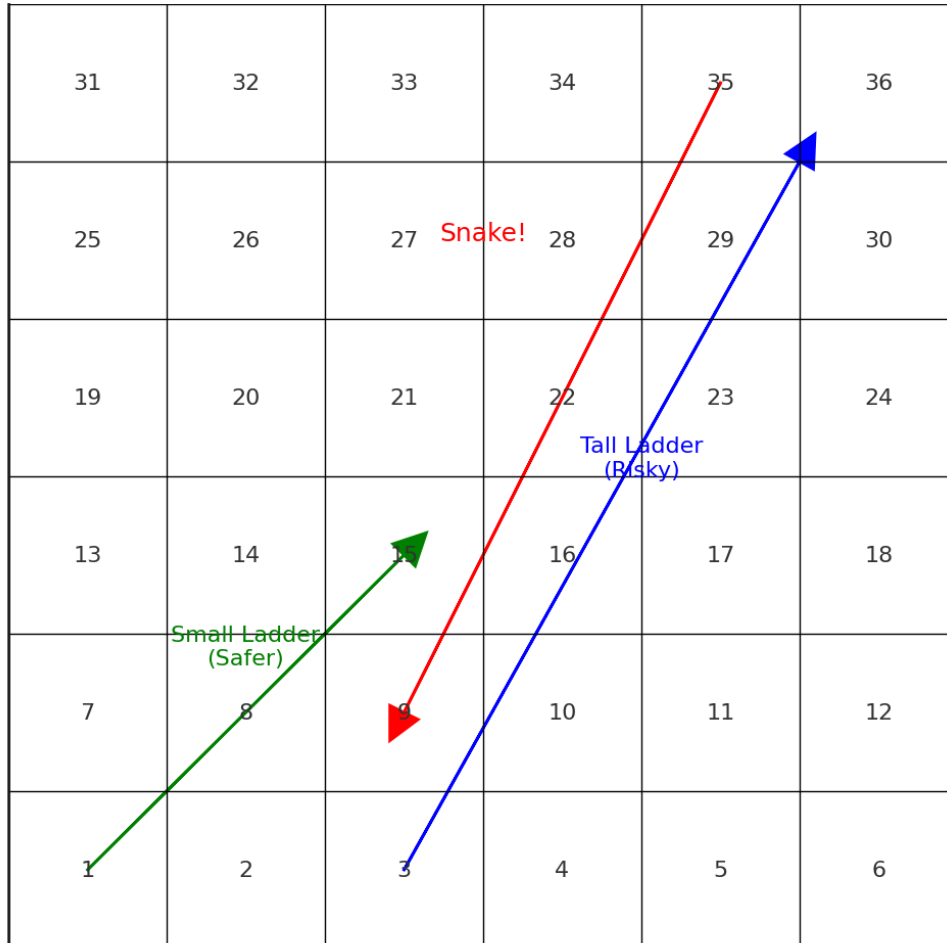
Note: If you get stuck on a small hill, go back down and start climbing again from another place until you reach the tallest mountain.

Stochastic Hill Climbing

- You check a few random neighbors.
- If some are better, you pick one of them at random (not always the best).
- This randomness helps explore different paths instead of always going straight to the steepest.

It is like choosing between several better ladders to climb — instead of always taking the tallest ladder, you sometimes pick a shorter one to see if it leads to a higher place later.

Comparing Snake & Ladder Game with Stochastic Hill Climbing:



4. Best-First Search Algorithm

What is Best First Search?

- **Best First Search** is a **heuristic search strategy** that expands the **most promising node first**.
- It uses a heuristic function $h(n)$ to estimate how close a node is to the goal.
- At each step → choose the node with the **lowest heuristic value** (the one that *looks closest to the goal*).

Key Idea: Always pick the “best-looking option” next.

Algorithm

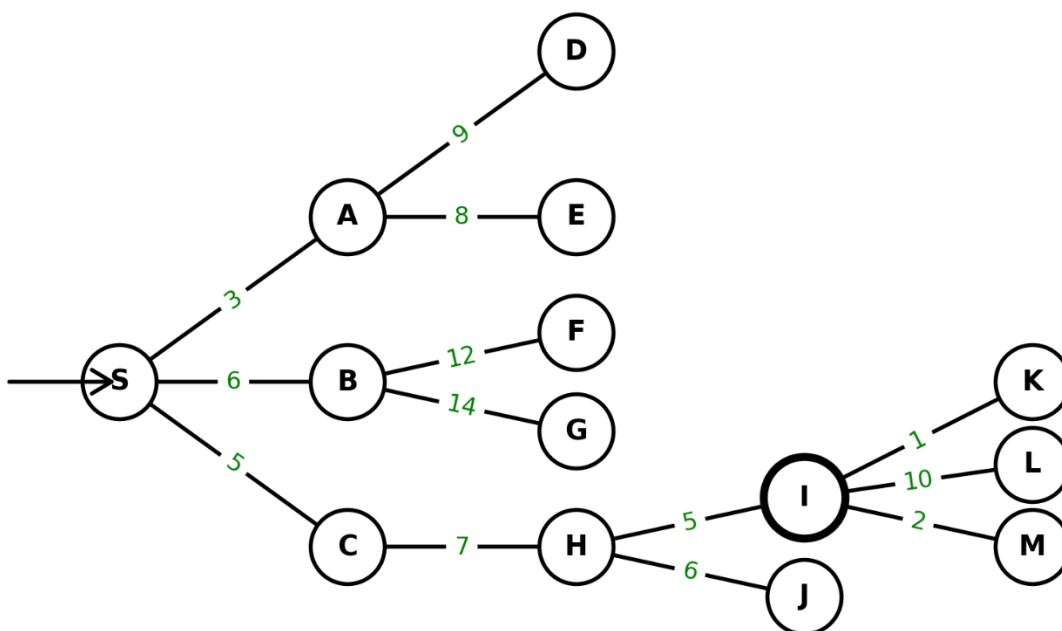
1. Start with the initial node.
2. Put it in a priority queue (ordered by heuristic value).
3. Remove the node with the lowest $h(n)$.
4. If it is the goal → stop.

5. Else, expand it and add its neighbors to the queue.
6. Repeat until the goal is found or queue is empty.

Pseudo code:

- Initialize an empty Priority Queue named **pq**.
- Insert the starting node into **pq**.
- While **pq** is not empty:
 - Remove the node **u** with the lowest evaluation value from **pq**.
 - If **u** is the goal node, terminate the search.
 - Otherwise, for each neighbour **v** of **u**: If **v** has not been visited, Mark **v** as visited and Insert **v** into **pq**.
 - Mark **u** as examined.
- End the procedure when the goal is reached or **pq** becomes empty.

Let's consider the following Graph:



1. Start

- We start from source "S" and search for goal "I" using given costs and Best First search.
- pq initially contains S
pq = {S}
- We remove S from pq and process unvisited neighbors of S to pq.
Remove S → expand neighbors A, B, C.
- pq now contains {A, C, B} (C is put before B because C has lesser cost)
pq = {A(3), C(5), B(6)}
We sort by cost → lowest first.

2. Expand A

- Remove **A (3)** → expand its neighbors D(9), E(8).
- pq = {C(5), B(6), E(8), D(9)}

3. Expand C

- Remove **C (5)** → expand neighbour H(7).

- $pq = \{B(6), H(7), E(8), D(9)\}$

4. Expand B

- Remove **B (6)** → expand neighbors F(12), G(14).
- $pq = \{H(7), E(8), D(9), F(12), G(14)\}$

5. Expand H

- Remove **H (7)** → expand neighbors I(5), J(6).
- $pq = \{I(5), J(6), E(8), D(9), F(12), G(14)\}$

6. Goal Found!

- Node **I** is our goal → search ends successfully.

In short:

Best First Search = *“Always pick the node that looks closest to the goal.”*

Here, that strategy led us from **S** → **A** → **C** → **B** → **H** → **I**.

So:

- **S** → **A** → **C** → **B** → **H** → **I** = **node expansion order (search process)**.
- **S** → **C** → **H** → **I** = the actual **solution path to the goal**.

Expansion order shows *all the exploration the algorithm had to do* before finding the goal.

Solution path is just the *shortest/greedy path the algorithm discovered*.

Types of Best-First Search Algorithms

These are of two types:

- **Greedy Best-First Search**
- **A* Search Algorithm**

What we have just seen in the previous example is **Greedy Best First Search (GBFS)**.

5. A* Search Algorithm

What is A* Search Algorithm?

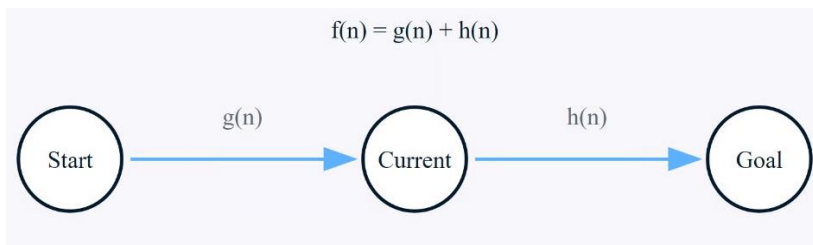
A* is a **pathfinding** and **graph traversal algorithm** that finds the **shortest path** from a start node to a goal node.

It combines the strengths of **Dijkstra’s Algorithm** (which uses actual path cost) and **Greedy Best-First Search** (which uses heuristic guess).

A* uses a cost function:

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

- **g(n)**: The actual cost from the start node to the current node n.
- **h(n)**: The heuristic estimate of the cost from n to the goal node.
- **f(n)**: The estimated total cost of the cheapest solution through n.



This means: *Pick the path that seems cheapest overall (past + future).*

A* always expands the node with the lowest $f(n)$.

Note: Finding shortest path by exploring all options would be very slow for a global network. That's why Google Maps uses A* Algorithm instead of Dijkstra's Algorithm.

A* Search is described as a **combination of UCS and Best-First Search**:

A* Search

- A* combines **both ideas**.
- It uses:

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

where

- **$g(n)$** = cost so far (like UCS),
- **$h(n)$** = estimated cost to goal (like Best-First Search).

So, A* is literally:

UCS (real cost so far) + Best-First (guess of remaining cost).

Steps of A* Algorithm

1. **Start at initial node** → put it in a priority queue.
2. For each node:
Calculate:

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

 $g(n)$ = actual cost so far
 $h(n)$ = estimated cost to goal
3. Always expand the node with lowest $f(n)$.
4. If goal is reached → stop (solution found).
5. Else → expand neighbors, update costs, and repeat.

A* = **Smartest search algorithm** → it's like **Google Maps**, always finding the **best route** by combining what you've already travelled and what's left.

Example: MVP Colony → Sanketika Engineering College

- **$g(n)$** : How many kilometers you've already driven from MVP Colony to Hanumanthawaka Junction.
- **$h(n)$** : Your guess of how many kilometers are still left (like looking at Google Maps' "straight line distance").

- **f(n)**: Your estimate of the total trip distance if you continue on the current route.

Both **g(n)** and **h(n)** are distances, but:

- **g(n)** = actual distance covered.
- **h(n)** = estimated distance still to go.

A* Properties:

- **Complete**: Always finds solution if one exists
- **Optimal**: Finds least-cost solution (if heuristic is admissible)
- **Optimally Efficient**: No other algorithm can do better with same heuristic

In grid-based or map-like problems, common heuristic functions include the Manhattan distance and Euclidean distance. For coordinates (x1,y1) of the current node and (x2,y2) of the goal node, these distances are calculated as:

Note:

In the **A* search algorithm**, both **Euclidean distance** and **Manhattan distance** are commonly used as **heuristic functions** h(n) to estimate the cost from a node n to the goal.

Euclidean Distance

- Formula:

$$h(n) = \sqrt{(x_{goal} - x_n)^2 + (y_{goal} - y_n)^2}$$

($x_{goal} - x_n$) = horizontal distance between current node and goal.

($y_{goal} - y_n$) = vertical distance between current node and goal.

- Measures the **straight-line ("as-the-crow-flies") distance** between two points.
- Best suited for problems where movement can occur in **any direction**, including diagonals.
- Example: A robot moving freely in a 2D plane.

Manhattan Distance

- Formula:

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

- Measures the distance if you can only move **horizontally or vertically** (like a grid city street).
- Best suited for **grid-based movements without diagonals**.
- Example: Navigating a 2D grid maze.

A* Algorithm Steps

1. Initialize

- Create an **open list** (nodes to explore) and a **closed list** (already explored nodes).
- Add the **start node** to the open list.

2. **Loop until open list is empty**
 - a. Pick the node in the open list with the **lowest $f(n) = g(n) + h(n)$** . Call it the **current node**.
 - b. If the current node is the **goal**, stop. Reconstruct the path by following parent links.
3. **Expand neighbors**
 - For each neighbour of the current node:
 - i. If it's in the closed list, **skip it**.
 - ii. Compute $g(n)$ (cost from start to neighbour).
 - iii. Compute $h(n)$ (heuristic estimate from neighbour to goal).
 - iv. Compute $f(n) = g(n) + h(n)$.
4. **Add/update neighbors in the open list**
 - If the neighbour is not in the open list, add it.
 - If it is already in the open list but the new $g(n)$ is **lower**, update its $f(n)$ and parent.
5. **Move current node to the closed list**
6. **Repeat** steps 2–5 until:
 - The goal is reached → **return path**
 - Open list is empty → **no path exists**

Example (Google Maps analogy)

- **Start = Home**
- **Goal = School**
- Open list initially = {Home}.
- Pick Home → expand roads → add neighbors (junctions).
- Each junction has cost so far (distance travelled = g) + estimated distance to school (straight-line = h).
- Always choose the junction with **lowest total cost $f = g+h$** .
- Repeat until you reach school.

Traffic Data Sources for Google Maps:

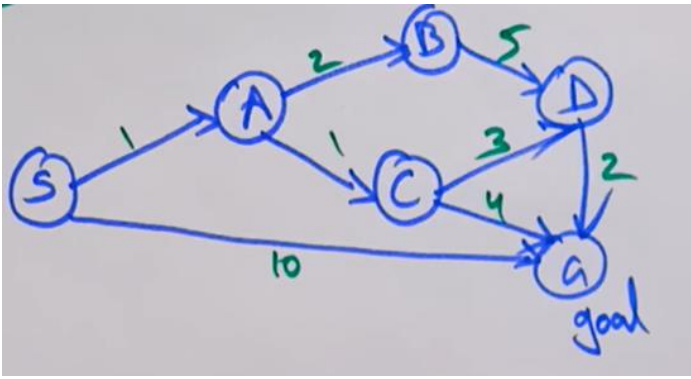
- Anonymous GPS data from smartphones in the traffic
- Historical traffic patterns, like Rush Hours: Eg: morning 8:00 am to 9:45 am, evening 4:30 pm to 6pm.
- Govt. Sensors and toll gates data
- Road conditions

Note: The trickier part is choosing **good $h(n)$ values** for the other nodes — they should never **overestimate** the real distance, otherwise A* might fail to find the optimal path. (This property is called an **admissible heuristic**).

Note: The following condition is always true:

- In any A* search problem,
 - **$h(\text{goal}) = 0$**
- That's a basic rule of heuristics.

Example Graph:



Let us consider the following heuristic values:

Heuristic values ($h(n)$):

- $h(S) = 5$
- $h(A) = 3$
- $h(B) = 4$
- $h(C) = 2$
- $h(D) = 2$
- $h(G) = 0$

In A*, the heuristic must be:

$h(n)$ should be \leq true shortest distance from n to goal

If $h(n) > \text{true distance}$, then the heuristic is **not admissible**, and A* might fail to guarantee the optimal path.

Step1. Start at S:

- $g(S) = 0$
- $f(S) = 0 + h(S) = 5$

Expand S \rightarrow neighbors:

- To A: $g(A) = 1$, $f(A) = 1 + h(A) = 1 + 3 = 4$
- To G: $g(G) = 10$, $f(G) = 10 + h(G) = 10 + 0 = 10$

Open list = { A(4), G(10) }

Note: The Open list is a **priority queue sorted by $f(n)$** , with the **lowest $f(n)$ node in the front**.

Choose **A** (lowest $f = 4$).

Step2. Expand A:

- Current $g(A) = 1$
Neighbors:
- To B: $g(B) = g(A) + 2 = 1 + 2 = 3 \rightarrow f(B) = 3 + h(B) = 3 + 4 = 7$
- To C: $g(C) = g(A) + 1 = 1 + 1 = 2 \rightarrow f(C) = 2 + h(C) = 2 + 2 = 4$

Open list = { C(4), B(7), G(10) }

Choose **C** (lowest $f = 4$).

Step3. Expand C:

- Current $g(C) = 2$
Neighbors:
- To D: $g(D) = g(C) + 3 = 2 + 3 = 5 \rightarrow f(D) = 5 + h(D) = 5 + 2 = 7$
- To G: $g(G) = g(C) + 4 = 2 + 4 = 6 \rightarrow f(G) = 6 + h(G) = 6 + 0 = 6$

Open list = { G(6), B(7), D(11), (old G(10)) }

But we update G to the smaller cost = 6.

So Open list = { G(6), B(7), D(11) }

Choose **G** (lowest $f = 6$).

Reached Goal G!

Final Path

We trace back:

$S \rightarrow A \rightarrow C \rightarrow G$

- Cost:
 - $S \rightarrow A = 1$
 - $A \rightarrow C = 1$
 - $C \rightarrow G = 4$
 - **Total = 6**

Conclusion

- A* found the **optimal path**: $S \rightarrow A \rightarrow C \rightarrow G$ with cost 6.
- Notice how it ignored the direct edge $S \rightarrow G$ (cost 10) because it found a cheaper route using heuristic + actual cost.

Applications of A* Algorithm

The A* algorithm's ability to find the most efficient path with a given heuristic makes it suitable for various practical applications:

1. **Pathfinding in Games and Robotics:** A* is used in the gaming industry to control characters in dynamic environments as well as in robotics for navigating between points.
2. **Network Routing:** In telecommunications, it helps in finding the shortest routing path that data packets should take to reach the destination.
3. **Map and Navigation Systems:** GPS navigation systems use A* to find optimal driving or walking routes in real time.
4. **Logistics and Supply Chain:** A* helps in optimizing routes for delivery vehicles and warehouse robots to minimize travel time and costs.

6. Problem Reduction

In **heuristic search**, "Problem Reduction" means:

- Break a **big complex problem** into **smaller sub-problems**.
- Solve those smaller parts one by one.
- Combine their solutions to get the answer for the original problem.

Why use it in AI?

Because some problems are too **large and complicated** to solve directly.

By reducing them into smaller, easier ones, we can solve them faster using heuristics (smart guesses).

Example 1 – Solving a Puzzle

Suppose you are solving the **8-puzzle game** (sliding tiles).

- The **big problem**: Arrange all tiles into correct order.
- **Problem Reduction**:
 1. First place the top row correctly.
 2. Then fix the middle row.
 3. Finally arrange the bottom row.
- Each step reduces the original problem into a smaller one.

Example 2 – Pathfinding

- **Big problem**: Find path from City A → City Z.
- **Problem Reduction**:
 1. First reach City M.
 2. From M → City T.
 3. Then from T → City Z.
- Instead of searching all possibilities at once, you reduce the path into **milestones**.

So, Problem Reduction = Solve a complex problem by **breaking it into smaller, easier sub-problems** and combining their solutions.

There are two different ways of Reducing a Problem:

- **OR reduction** → Break the problem into **choices** (pick one).
- **AND reduction** → Break the problem into **components** (all required).

Together, this is called an **AND/OR graph representation of problem reduction**.

What is an AND/OR Graph?

An **AND/OR graph** is a way to represent problems that can be **broken into sub-problems**.

- **OR Node** → "Choose one of the options" (alternative solutions).
- **AND Node** → "You must solve all of these sub-problems together."

Think of it like a **decision tree**, but with some branches that need **all children solved (AND)** and some where you can pick **one child (OR)**.

Example 1 – Studying for Exams

- **Problem**: Pass the exam.
 - **OR node**:
 - Either **study from textbook**, OR
 - **attend coaching**, OR
 - **watch YouTube lectures**.(Any one path can solve the problem.)
 - **AND node**:
 - Prepare **Math**, AND
 - Prepare **Science**, AND

- Prepare **English**.
(Because, we must pass in all exams / All must be solved together.)

Example 2 – AI Puzzle Solving

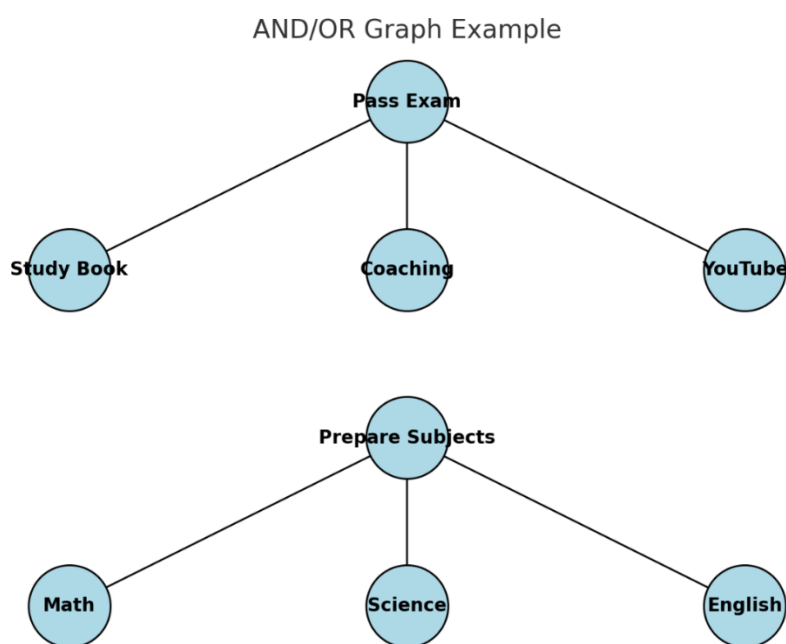
- **OR node:** To place a tile, you may move **left OR right OR up OR down** (choose one).
- **AND node:** To solve the puzzle, you must place **all tiles** correctly.

Example 3 – Robot Task

Problem: Robot needs to clean a room.

- **OR reduction:**
 - Clean using **Vacuum OR Mop OR Blower** (choose one).
- **AND reduction:**
 - To clean completely → Must clean **Floor AND Windows AND Table**.

This way, the robot's task (big problem) is reduced into sub-problems that are connected by **AND/OR** relationships.



AO* Algorithm for Problem Reduction

(AND-OR Star Algorithm)

- It is a **search algorithm** used on **AND/OR graphs**.
- Just like **A*** is used on normal graphs, **AO*** is used when the problem involves **both AND and OR nodes**.
- It finds the **least-cost solution** by considering both alternatives (OR) and combined sub-problems (AND).

Key Idea

1. Start with a **big problem** (root node).
2. Expand it into sub-problems (AND/OR nodes).
3. Use **heuristics** (estimates) to guess which path is better.
4. Keep expanding the most promising path (like A*).
5. Stop when a complete solution (all AND requirements satisfied, or a good OR choice selected) is found.

Example (Robot Cleaning)

Problem: Robot must clean the house.

- From "Clean House" (root):
 - **AND Node** → Must clean **Floor AND Table AND Windows**.
 - **OR Node** for Floor cleaning → Choose **Vacuum OR Mop OR Blower**.

AO* Algorithm will:

- Look at the cost (time/energy) of each cleaning method.
- For the OR node, it will pick the **cheapest method** (say Vacuum).
- For the AND node, it will add up the costs of **Floor + Table + Windows**.
- Finally, it gives the **best overall plan** for cleaning the house.

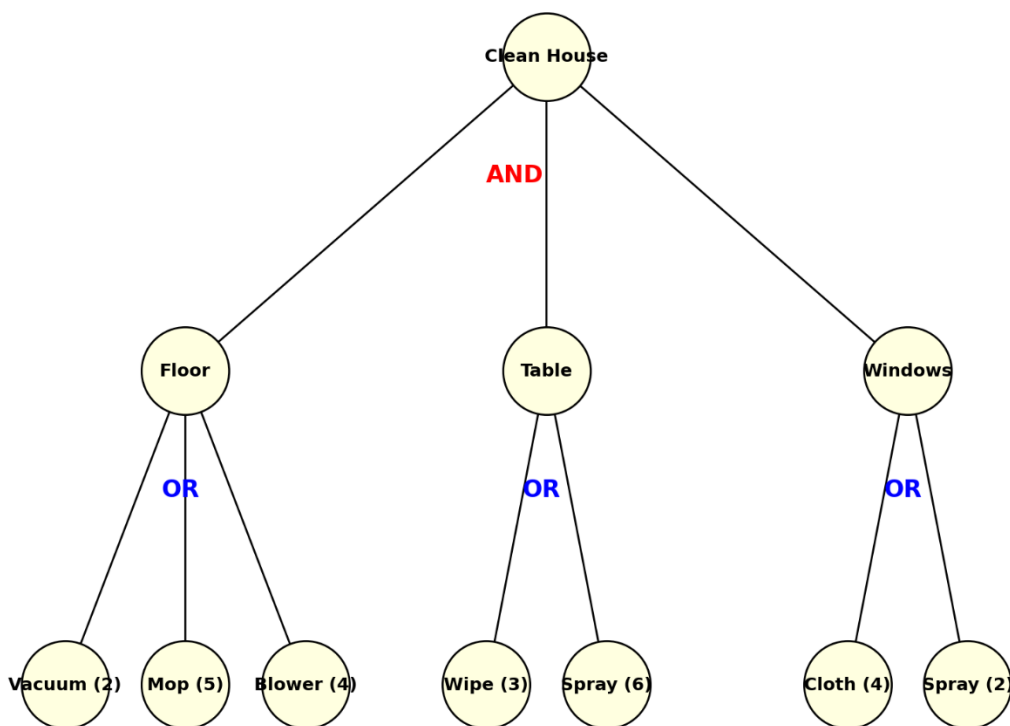
AO* Algorithm steps

1. **Start at the root node** (big problem).
2. **Expand nodes** into sub-problems (using AND/OR rules).
3. **Calculate cost:**
 - OR node → take **minimum cost** among children.
 - AND node → take **sum of costs** of all children.
4. **Use heuristics** to guide expansion.
5. **Repeat** until the solution graph is built with the lowest cost path.

Difference from A*

- **A*** works on normal graphs (single path to goal).
- **AO*** works on **AND/OR graphs** (multiple sub-problems and choices).

AO* Example: Robot Cleaning with Costs



Here's the **AO*** graph with costs:

- **Clean House (AND)** → Must do **Floor + Table + Windows**.

- **Floor (OR)** → Vacuum (2), Mop (5), Blower (4) → AO* picks **Vacuum (2)**.
- **Table (OR)** → Wipe (3), Spray (6) → AO* picks **Wipe (3)**.
- **Windows (OR)** → Cloth (4), Spray (2) → AO* picks **Spray (2)**.

Final AO* solution path = Vacuum + Wipe + Spray = Total Cost = 7.

Question: Why keep the costlier options?

1. Because we don't know the best option in advance

- At the start, AO* has only **heuristics (estimates)**, not exact costs.
- A choice that looks cheap early may turn out to be worse when combined with other sub-problems.
- Example:
 - Vacuum = 2 (looks cheap), but maybe it requires electricity (extra hidden cost).
 - Mop = 5 (looks expensive), but maybe it works when electricity fails.

2. Because costs can change depending on the context

- What if the window spray bottle is empty? Then "Spray (2)" is not valid anymore, and AO* may need to fall back to "Cloth (4)".
- Example in **pathfinding**:
 - Route A → looks short but has a traffic jam.
 - Route B → looks long but is faster in real time.

7. Constraint Satisfaction Problems (CSP)

It's a way of solving problems by **filling in values (choices)** such that **all the rules/conditions (constraints) are satisfied**.

In artificial intelligence, a **Constraint Satisfaction Problem (CSP)** is a type of problem where a set of variables must be assigned values from their respective domains, such that all defined constraints are simultaneously satisfied.

CSP Components:

- **Variables:** Things we need to assign values to
- **Domains:** Possible values for each variable
- **Constraints:** Rules that limit combinations

Examples

Example 1. Sudoku Puzzle

- **Variables** → Empty cells in the grid
- **Domain** → Numbers 1–9
- **Constraints** →
 - Each row must have 1–9 without repetition
 - Each column must have 1–9 without repetition
 - Each 3×3 block must have 1–9 without repetition

The solution = Fill numbers so that **all constraints are satisfied**.

1	6	8				9		2
			3		1			
	3		6	2				
		9				1		6
		1				3	7	
	4	3	5					9
			8		2	6		
			9		5		2	3
2		6		3		7		

Example2: Scheduling Problem

- **Variables** → Time slots for classes
- **Domain** → {Monday 9AM, Monday 10AM, ...}
- **Constraints** →
 - A teacher can't teach 2 classes at the same time
 - A room can only host 1 class at a time

Solution = Assign times/rooms so **all constraints are respected**.

Example Timetable (Solution)

Time Slot Room101 Lab1

Mon 9AM Math (BVRaju) Science (Sunil)

Mon 10AM English (Prasad) (Free)

Example3: Map Colouring

- **Variables** → Regions on a map
- **Domain** → {Red, Green, Blue, Purple}
- **Constraints** → Neighbouring regions must have different colors

Asian countries example: If India and Nepal are neighbors, they must not share the same color.



Why useful in AI?

Many AI problems can be written as **CSPs (Constraint Satisfaction Problems)**, like:

- Timetabling (schools, airlines)
- Resource allocation (factories, hospitals)
- Logic puzzles (crosswords, Sudoku)
- Planning & scheduling in robotics

CSP Solution Methods (Search based methods):

(a) Backtracking Search

- Start by assigning values to variables one by one.
- If a constraint is violated → backtrack (undo and try another value).
- Example: Sudoku → if “5” doesn’t work in a cell, erase and try “6”.
- Works well with **small problems**, but slow for big ones.

(b) Forward Checking

- When you assign a value, eliminate impossible choices from the neighbors’ domains.
- Example: In map colouring → if India = Red, remove Red from Nepal’s domain before trying.
- Avoids wasting time going into dead ends.

(c) Constraint Propagation (Arc Consistency, AC-3)

- After each assignment, repeatedly enforce consistency across all variables.
- Example: Sudoku → if a row already has {1,2,3,4,5,6,7,8}, then the last empty box **must be 9**.
Eliminating impossible digits from a cell’s possibilities whenever other cells are filled or restricted.
- Helps reduce the search space a lot.

(d) Heuristic Search (MRV, LCV, etc.)

- **MRV (Minimum Remaining Values)**: Choose the variable with the fewest legal values left.

Example (Sudoku)

- Suppose you have 3 empty cells in a row:
 - Cell A → possible values {3, 4, 9}
 - Cell B → possible values {2, 5}
 - Cell C → possible values {7}
- **MRV says**: Start with **Cell C** because it has only **1 legal value (7)**.
- **Why?** Because it’s the most constrained. If you don’t solve it first, you might waste time later.
MRV = “Solve the hardest/most restricted part first.”
- **LCV (Least Constraining Value)**: Rule: When assigning a value, choose the one that leaves the most options open for others.

Example: Suppose you are solving a Sudoku and come to an empty cell **X**.

Cell **X** has possible values = {2, 7, 9}

Neighbors in the same row: **A, B, C**

- If **X = 2** → A, B, C all lose 2.
- If **X = 7** → A, B, C all lose 7.
- If **X = 9** → A, B, C all lose 9.

Suppose the neighbors have these possible values:

- A = {2, 7, 8}
- B = {2, 4, 5}
- C = {7, 8, 9}

Now test assignments:

- **If X = 2** →
 - A loses 2 → {7, 8}

- B loses 2 → {4, 5}
 - C unaffected → {7, 8, 9}
 - (Neighbors still have decent choices)
- **If X = 7 →**
 - A loses 7 → {2, 8}
 - B unaffected → {2, 4, 5}
 - C loses 7 → {8, 9}
 - (Still okay).
- **If X = 9 →**
 - A unaffected → {2, 7, 8}
 - B unaffected → {2, 4, 5}
 - C loses 9 → {7, 8}
 - (Least harmful).

LCV picks 9, because it leaves neighbors with the **most total choices**. LCV is like being polite in Sudoku.

8. Means-Ends Analysis (A problem solving strategy)

Key Idea:

- Reduce difference between current state and goal
- If direct action not possible, set up sub-goals

Algorithm:

It follows a **clear sequence of steps**:

1. **Start** with the current state.
2. **Compare** current state with goal state.
3. **Identify the difference**.
4. **Select an operator (action)** that reduces the difference.
5. **Apply** the operator → reach a new state.
6. **Repeat** until the goal is reached (or no operators are available).

Example1: A Student going to School

Goal: Be in the Classroom at School

- **Current State:** At home, still in night dress
- **Goal State:** Sitting in classroom, ready to listen to the lecture

Step-by-Step with MEA

1. **Compare Current vs Goal**
 - Difference: Not dressed, not prepared.
 - **Sub-goal:** Get ready.
 - **Action:** Brush teeth, take a bath, wear uniform.
 - **New Current State:** Ready at home.
2. **Compare again**
 - Current: Ready at home.
 - Goal: At classroom.
 - Difference: Still far from school.

- **Sub-goal:** Reach school.
 - **Action:** Take bus/ride bicycle.
- **New Current State:** Reached school gate.

3. Compare again

- Current: At school gate.
 - Goal: In classroom.
 - Difference: Not inside classroom.
 - **Sub-goal:** Enter building, go upstairs.
 - **Action:** Walk to classroom.
- **New Current State:** In classroom.

Example2: Robot Serving Tea



Goal: Robot must place a cup of tea on the table in front of the user.

- **Current State:** Robot is in the charging dock, table is empty.
- **Goal State:** Cup of tea on the table.

Step-by-Step MEA

1. Compare Current vs Goal

- Current: Robot in dock, no tea.
 - Goal: Tea on table.
 - **Difference:** Tea is missing.
 - **Sub-goal:** Get tea.
 - **Action:** Move to kitchen.
- **New State:** Robot in kitchen.

2. Compare again

- Current: Robot in kitchen, no tea.
 - Goal: Tea on table.
 - **Difference:** Still no tea.
 - **Sub-goal:** Acquire tea.
 - **Action:** Pick up cup of tea. (we assume that the tea is **already prepared** by someone (a human, or maybe another robot/tea machine) and placed in the kitchen.)
- **New State:** Robot holding tea.

3. Compare again

- Current: Robot holding tea in kitchen.
- Goal: Tea on table.
- **Difference:** Location mismatch.
- **Sub-goal:** Deliver tea to table.
- **Action:** Walk to living room.
- **New State:** Robot near table, holding tea.

4. Compare again

- Current: Robot near table, holding tea.
- Goal: Tea on table.
- **Difference:** Tea not placed.
- **Sub-goal:** Place tea on table.
- **Action:** Put cup down.
- **New State:** Tea on table.

Goal Reached!

Note: This is how many robots and planning systems in AI actually work: they use MEA-like strategies to reduce complex tasks into solvable subgoals.