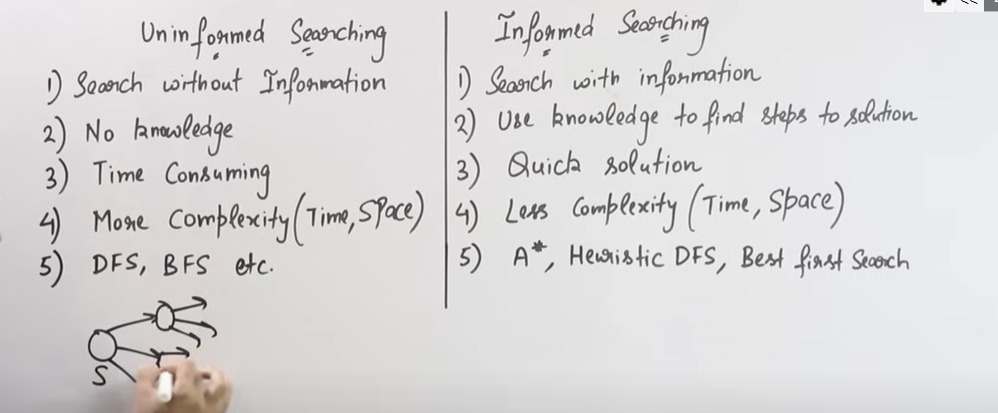
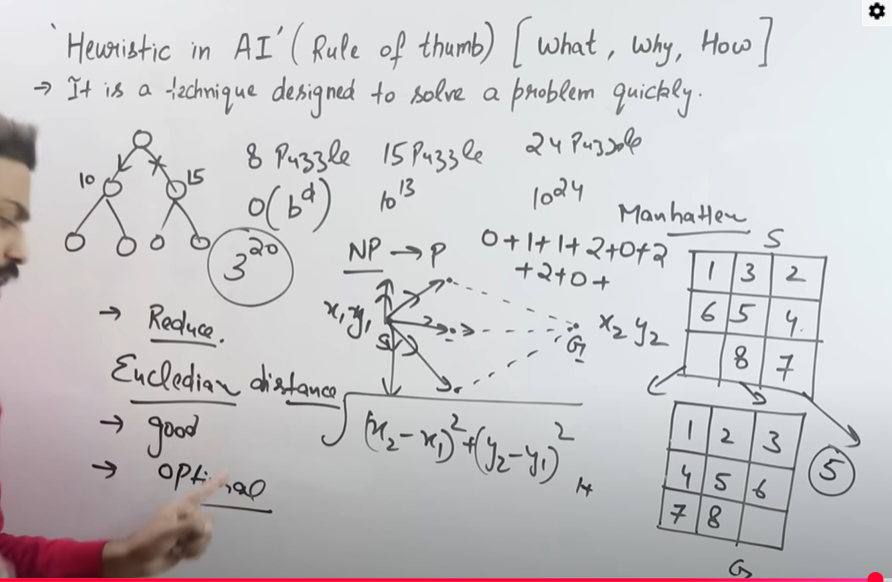
| **Point** | **BFS (Breadth-First Search)** | **DFS (Depth-First Search)** |
| --- | --- | --- |
| **How it works** | Explores level by level → first all neighbors, then their children. | Goes deep along one branch, then backtracks. |
| **Search Type** | **Uninformed** (blind, no extra info). | **Uninformed** (blind, no extra info). |
| **Completeness (will it always find a solution?)** | ✅ Yes, if a solution exists, BFS will definitely find it. | ❌ No, DFS can get stuck in infinite loops if path is too deep. |
| **Optimality (best/shortest solution)** | ✅ Yes, BFS finds the shortest path when all steps have equal cost. | ❌ No, DFS may give a longer or non-optimal path. |
| **Time Complexity** | O(bd)O(b^d) → grows very fast with depth. bb = branching factor, dd = depth of solution. | O(bm)O(b^m) → depends on maximum depth. mm can be much bigger than dd. |
| **Space Complexity (memory use)** | High → stores all nodes at the current level. | Low → stores only the current path + some siblings. |
| **When to use** | When you need the **shortest solution** and memory is not a problem. | When **memory is limited** and you just need *any solution*. |
| **Example (maze solving)** | Explores all shortest routes first → finds exit in minimum steps. | May go deep into a wrong path before backtracking, so might take longer or miss shortest path. |

Use of **Heuristics** - informed search algorithms use heuristics, or additional information, to guide the search process and prioritize which nodes to expand.

Blind Search Tech Heuristic S.T





**NP problems** are problems that are very hard to solve exactly in reasonable time

**Exact solution** → can take **exponential time** (like checking every possible combination).

tries to find a **good-enough solution** in **polynomial time** by making smart guesses instead of brute-force.

**-: A\* :-**

Absolutely! Let’s do a **step-by-step dry run of A\*** on a simple **8-puzzle example** using **Manhattan distance** so you can visualize it like boxes.

**Initial Setup**

**Start state (0 = empty):**

1 2 3

4 0 6

7 5 8

**Goal state:**

1 2 3

4 5 6

7 8 0

**Step 1: Compute Heuristic (Manhattan Distance)**

* Tile 5: current (2,1), goal (1,1) → distance = 1
* Tile 8: current (2,2), goal (2,1) → distance = 1

**h(start) = 1 + 1 = 2**  
**g(start) = 0** → **f = g + h = 0 + 2 = 2**

**Step 2: Generate Possible Moves from Start**

Empty tile 0 at position (1,1). Moves: Up, Down, Left, Right

| **Move** | **Resulting State** | **g** | **h (Manhattan)** | **f = g + h** |
| --- | --- | --- | --- | --- |
| Up | Move 0 up → swap with 2 → |  |  |  |

1 0 3

4 2 6

7 5 8

| g = 1 | h = 4 | f = 5 |  
| Down | Move 0 down → swap with 5 →

1 2 3

4 5 6

7 0 8

| g = 1 | h = 1 | f = 2 |  
| Left | Move 0 left → swap with 4 →

1 2 3

0 4 6

7 5 8

| g = 1 | h = 3 | f = 4 |  
| Right | Move 0 right → swap with 6 →

1 2 3

4 6 0

7 5 8

| g = 1 | h = 3 | f = 4 |

**Step 3: Pick Node with Lowest f**

* Pick **Down move** → f = 2 (lowest)

**State:**

1 2 3

4 5 6

7 0 8

* g = 1, h = 1, f = 2

**Step 4: Generate Next Moves**

Empty tile 0 at (2,1)

| **Move** | **Resulting State** | **g** | **h** | **f** |
| --- | --- | --- | --- | --- |
| Up | swap with 5 → back to previous → ignore visited | 2 | 2 | 4 |
| Down | invalid (out of bounds) | - | - | - |
| Left | swap with 7 → |  |  |  |

1 2 3

4 5 6

0 7 8

g = 2, h = 3 → f = 5

| Right | swap with 8 →

1 2 3

4 5 6

7 8 0

g = 2, h = 0 → f = 2

**Step 5: Pick Node with Lowest f**

* Pick **Right move** → f = 2

**State: Goal reached!**

1 2 3

4 5 6

7 8 0

* g = 2 → solution found in 2 moves.

**Step 6: Path Summary**

| **Step** | **Move** | **Puzzle State** |
| --- | --- | --- |
| 0 | Start | 1 2 3 / 4 0 6 / 7 5 8 |
| 1 | Down | 1 2 3 / 4 5 6 / 7 0 8 |
| 2 | Right | 1 2 3 / 4 5 6 / 7 8 0 |

**Solution:** Down → Right

✅ **Key Points from Dry Run**

1. **A\*** always chooses the state with **lowest f = g + h**.
2. **Manhattan distance guides** A\* to states that are closer to the goal.
3. **Visited states** prevent loops.
4. This is **much faster than BFS**, because BFS would explore all nodes level by level, even suboptimal paths.

**Short Summary:-**

 First program → **basic Tic-Tac-Toe with two players**, using **2D array, loops, if-else**.

 Second program → **Tic-Tac-Toe with AI using minimax**, which uses **recursion and backtracking** to make the best move.

 **#include <iostream>** → lets us use cin and cout for input/output.

 **#include <vector>** → gives us the **vector** (dynamic array). Unlike normal arrays, vector can grow/shrink in size.

 **#include <queue>** → gives us **queue** (First-In-First-Out, like a line at ticket counter). Used for BFS.

 **#include <stack>** → gives us **stack** (Last-In-First-Out, like a pile of plates). Used for DFS.

 **#include <set>** → gives us a container that stores **unique values** only. Used to remember visited states.

 **#include <algorithm>** → gives us extra functions like find() to search in vectors.

You call:

vector<vector<int>> nextMoves = getMoves(state);

You now get a list of valid moves (i.e., **swapping 0** with an adjacent tile), such as:

{

{1, 0, 3, 4, 2, 6, 7, 5, 8}, // 0 moved left (swapped with 2)

{1, 2, 3, 0, 4, 6, 7, 5, 8}, // 0 moved up (swapped with 4)

{1, 2, 3, 4, 6, 0, 7, 5, 8}, // 0 moved right (swapped with 6)

{1, 2, 3, 4, 5, 6, 7, 0, 8} // 0 moved down (swapped with 5)

}

Each of those is a **new puzzle configuration** (a vector<int>), and you're storing all of them in a vector<vector<int>>.

The keyword **auto** is a **C++ feature (since C++11)** that tells the compiler:

“Figure out the type of this variable automatically based on the right-hand side.”

ueue<pair<vector<int>, int>> q;

That means each item in the queue is a pair:

The first part is the puzzle state (vector<int>)

The second part is the number of steps (int)

----  
**The for(auto move : nextMoves) loop:**

Goes through each of these possible new puzzle states.

**💬 stateToString(move):**

Turns the puzzle into a string like:

"103426758"

This is used so we can easily store it in a set<string> to track visited states.

----  
bool operator>(const Node& other) const { return f > other.f; }

🔹 What it means

This is called operator overloading.

Normally > compares two numbers.

But here we are telling C++: “When you compare two Node objects using >, compare them by their f value.”

So if we have:

cpp

Copy code

Node a, b;

a.f = 10;

b.f = 20;

cout << (a > b); // uses our custom > operator

👉 It will check 10 > 20 → false (0).

**👉 *“Sir, here we overloaded the > operator so that when the priority queue compares two puzzle nodes, it looks at their f value (g+h). This makes A* always expand the node with smallest cost first.”\***

**“Sir, this is a binary operator overloading of the relational operator >. It compares two Node objects based on their f value.”**