Artificial Intelligence (AI)

Artificial Intelligence is a broad field of computer science focused on creating machines or systems that can perform tasks that typically require human intelligence. This includes abilities like **learning**, **reasoning**, **problem-solving**, **perception**, and **language understanding**. The ultimate goal of AI is to create systems that can operate autonomously and adapt to new situations intelligently. Think of it as the science of making machines smart.

Strong AI vs. Weak AI

This is a fundamental philosophical distinction in AI.

- **Weak AI (Artificial Narrow Intelligence - ANI)**: This is the AI we have today. Weak AI is designed and trained to perform a **specific, narrow task**. It doesn't possess consciousness, self-awareness, or genuine understanding. It operates within a pre-defined range and simulates intelligence, but doesn't actually *think* like a human.
    - **Examples**: Siri, Alexa, Google Translate, self-driving cars, and recommendation engines on Netflix or Spotify. They are excellent at their specific jobs but can't perform tasks outside their domain.
- **Strong AI (Artificial General Intelligence - AGI)**: This is a theoretical form of AI where a machine would have an intellect equal to a human's. It would possess consciousness, self-awareness, and the ability to **understand, learn, and apply its intelligence to solve any problem**, just like a person. AGI doesn't exist yet and is a major goal for many AI researchers.

Levels of AI: ANI, AGI, and ASI

This classification describes the capability of an AI system.

1. **Artificial Narrow Intelligence (ANI)**:

    As explained above, this is AI that specializes in **one area**. It's also known as Weak AI. This is the only type of AI we have successfully created so far.

2. **Artificial General Intelligence (AGI)**:

    Also known as Strong AI, this is an AI that can understand, learn, and perform any intellectual task that a human being can. It has a **human-level intelligence** and can generalize its learning from one domain to another.

3. **Artificial Super Intelligence (ASI)**:

   This is a hypothetical AI that would surpass human intelligence across virtually every field, including scientific creativity, general wisdom, and social skills. ASI would be an intellect far smarter than the best human brains in every way. The concept of the "singularity" is often associated with the creation of ASI.

Intelligent and Rational Agents

In AI, an **agent** is anything that can perceive its environment through sensors and act upon that environment through actuators.

- **Intelligent Agent**: This is a general term for an autonomous entity that observes and acts upon an environment to achieve its goals. A human is an intelligent agent, as is a robotic vacuum cleaner.
  - **Components**:
    - **Percepts**: The agent's perceptual inputs at any given instant.
    - **Percept Sequence**: The complete history of everything the agent has ever perceived.
    - **Agent Function**: A mapping from any given percept sequence to an action.
- **Rational Agent**: This is a specific type of intelligent agent that always takes the action expected to **maximize its performance measure**, given the evidence provided by its percept sequence and any built-in knowledge. In simple terms, a rational agent acts to achieve the **best possible outcome**. It's not necessarily the most "intelligent" in a human sense, but it is the most logical and goal-oriented. For example, a chess-playing AI is a rational agent because it always chooses the move that it calculates will give it the highest probability of winning.

Turing Test

The **Turing Test**, developed by Alan Turing in 1950, is a test of a machine's ability to exhibit intelligent behavior equivalent to, or indistinguishable from, that of a human.

- **How it works**: A human evaluator engages in a natural language conversation with two other parties, one a human and the other a machine. All participants are separated from each other. If the evaluator **cannot reliably tell the machine from the human**, the machine is said to have passed the test.
- **Purpose**: It doesn't test for correct answers but rather how "human-like" the machine's answers are. It's a test for **imitation**, not necessarily for consciousness or true understanding.

AI vs. Machine Learning (ML) vs. Deep Learning (DL)

These terms are often used interchangeably, but they have distinct meanings. They are subsets of one another.

- **Artificial Intelligence (AI)**: The **broadest concept** of creating intelligent machines that can simulate human thinking and behavior. It's the overall umbrella.
- **Machine Learning (ML)**: A **subset of AI**. ML is an approach to achieve AI where we don't explicitly program rules. Instead, we provide algorithms with large amounts of data and let them **learn patterns** from it to make predictions or decisions.
- **Deep Learning (DL)**: A **subset of Machine Learning**. DL uses complex, multi-layered neural networks (inspired by the human brain's structure) to learn from vast amounts of data. It's the powerhouse behind recent breakthroughs like advanced image recognition and large language models (like me!).

Analogy: Think of them as Russian nesting dolls. 🪆

- The largest doll is **AI**.
- Inside it, you find the **ML** doll.
- Inside the ML doll, you find the smallest one, **DL**.

Introduction to Search Algorithms

In AI, search algorithms are used to solve problems by exploring a **state space**. Imagine you're in a maze. The maze is the state space, your current location is a "state," and each step you take is an "action" that transitions you to a new state. The goal is to find a sequence of actions (a "path") from the start state to a goal state.

Search algorithms are broadly divided into two categories: **Uninformed** and **Informed**.

Uninformed Search (Blind Search)

These algorithms have no additional information about the problem other than the problem definition itself. They don't know if one state is better than another. They simply generate successor nodes and distinguish a goal state from a non-goal state. Think of it as searching a maze with no clues or sense of direction.

Breadth-First Search (BFS)

BFS explores the state space **level by level**. It first explores all the nodes at the current depth before moving on to the nodes at the next depth level.

- **How it works**: It uses a **Queue** (First-In, First-Out).
    - Add the start node to the queue.
    - Dequeue a node and examine it.
    - If it's the goal, you're done!
    - If not, add all its unvisited neighbors to the back of the queue.
    - Repeat until the queue is empty or the goal is found.
- **Key Properties**:
    - **Complete**: Yes, it will always find a solution if one exists.
    - **Optimal**: Yes, but only if all path costs are equal (e.g., each step costs 1). It finds the shallowest solution.
    - **Complexity**: Time and space complexity are high, O(bd), where `b` is the branching factor and `d` is the depth of the solution. It requires a lot of memory.

Depth-First Search (DFS)

DFS explores the state space by going as **deep as possible** down one path before backtracking.

- **How it works**: It uses a **Stack** (Last-In, First-Out).
    - Add the start node to the stack.
    - Pop a node and examine it.
    - If it's the goal, you're done!
    - If not, add all its unvisited neighbors to the top of the stack.
    - Repeat until the stack is empty or the goal is found.
- **Key Properties**:
    - **Complete**: No, it can get stuck in infinite loops if the state space is infinite. (Can be made complete with a visited list).
    - **Optimal**: No, it finds the "leftmost" solution, which is not guaranteed to be the shortest.
    - **Complexity**: Time complexity is O(bm) (`m` is max depth), but its space complexity is much better than BFS, O(b⋅m).

Uniform Cost Search (UCS)

UCS is a variation of Dijkstra's algorithm. It explores the state space by always expanding the node with the **lowest total path cost** from the start node.

- **How it works**: It uses a **Priority Queue**, ordered by path cost (`g(n)`).
- **Key Properties**:
    - **Complete**: Yes.
    - **Optimal**: Yes, it is guaranteed to find the solution with the lowest cost.
    - **Difference from BFS**: While BFS finds the shallowest solution, UCS finds the cheapest one, even if it's deeper.

Depth-Limited Search (DLS)

This is simply DFS with a pre-defined **depth limit**. It stops going down a path once it reaches that limit, forcing it to backtrack. This solves the infinite loop problem of DFS.

- **Key Properties**:
    - **Complete**: No, if the shallowest solution is beyond the depth limit, it won't be found.
    - **Optimal**: No.

Iterative Deepening Search (IDS)

IDS combines the best of both worlds: the **low memory usage of DFS** and the **completeness/optimality of BFS**.

- **How it works**: It repeatedly runs DLS with an increasing depth limit. It first does a DLS with limit 0, then limit 1, then limit 2, and so on, until a solution is found.
- **Key Properties**:
    - **Complete**: Yes.
    - **Optimal**: Yes (for unit-cost paths).
    - **Why it's efficient**: While it seems wasteful to re-explore nodes, the majority of the nodes are at the bottom level of the search tree. So, the overhead of re-generating upper-level nodes is relatively small.

Informed (Heuristic) Search

These algorithms use problem-specific knowledge to guide the search. They use a **heuristic function**, denoted as h(n), which **estimates the cost** of the cheapest path from the current state n to a goal state. Think of this as having a compass in the maze that always points toward the exit.

Greedy Best-First Search

This algorithm tries to expand the node that is believed to be **closest to the goal**. It's "greedy" because it only considers this estimated cost, ignoring the cost already incurred to get to the current node.

- **How it works**: It uses a priority queue ordered by the heuristic, h(n). It always expands the node with the lowest h(n).
- **Key Properties**:
    - **Complete**: No. It can get stuck in loops.
    - **Optimal**: No. It can be tricked into taking a path that seems good at first but is actually a dead-end or very long.
    - **Fast**: It's very fast but trades optimality for speed.

A* Algorithm

A* (A-star) is one of the most popular and powerful search algorithms. It combines the strengths of UCS and Greedy Best-First Search. It evaluates nodes by combining the **cost to reach the node** (`g(n)`) and the **estimated cost to get from the node to the goal** (`h(n)`).

- **Evaluation Function**: f(n)=g(n)+h(n)
    - g(n): The actual cost from the start node to node `n` (like in UCS).
    - h(n): The estimated cost from node `n` to the goal (the heuristic).
- **How it works**: It uses a priority queue ordered by the lowest f(n) value.
- **Key Properties**:
    - **Complete**: Yes.
    - **Optimal**: Yes, provided the heuristic h(n) is **admissible** (it never *overestimates* the true cost to the goal) and, for graphs, **consistent**.

IDA* (Iterative Deepening A*)

This is the iterative deepening version of A*. It's a memory-efficient way to implement A* search.

- **How it works**: It works like IDS, but instead of a depth limit, it uses an **`f-cost` limit**. It starts with a cost limit equal to the f(n) of the start node. In each iteration, it performs a DFS, pruning any path that exceeds the current cost limit. If no solution is found, the new limit becomes the smallest cost that was pruned in the previous iteration.
- **Key Properties**: It has the same properties as A* (complete, optimal) but with the low memory requirements of DFS.

Hill Climbing

This is a **local search** algorithm, not a pathfinding one. It's used for optimization problems where you want to find the best state (e.g., the "highest peak").

- **How it works**: It's a "greedy" local search. It starts with a random solution and then iteratively makes small changes (moves to a neighbor state). It only accepts the change if the new state is better than the current one.
- **Key Problem**: It can get stuck in **local maxima**. It finds a "peak," but it might not be the highest peak in the entire landscape. It has no way of going "downhill" to explore other parts of the search space.

Beam Search

Beam search is a heuristic search algorithm that explores a graph by expanding the most promising nodes in a limited set.

- **How it works**: It's a variation of Best-First Search that limits the number of nodes it keeps open at each level. It has a parameter `k` called the **beam width**. At each step of the search, it generates all successors of the current `k` states but only keeps the best `k` successors for the next step.

- **Key Properties**: It's not complete or optimal, but it's very memory efficient. It's a trade-off between a greedy search and a complete search. If k=1, it's like Hill Climbing. If k=∞, it's like Best-First Search.

Introduction to Search Algorithms

In AI, search algorithms are used to solve problems by exploring a **state space**. Imagine you're in a maze. The maze is the state space, your current location is a "state," and each step you take is an "action" that transitions you to a new state. The goal is to find a sequence of actions (a "path") from the start state to a goal state.

Search algorithms are broadly divided into two categories: **Uninformed** and **Informed**.

Uninformed Search (Blind Search)

These algorithms have no additional information about the problem other than the problem definition itself. They don't know if one state is better than another. They simply generate successor nodes and distinguish a goal state from a non-goal state. Think of it as searching a maze with no clues or sense of direction.

Breadth-First Search (BFS)

BFS explores the state space **level by level**. It first explores all the nodes at the current depth before moving on to the nodes at the next depth level.

- **How it works**: It uses a **Queue** (First-In, First-Out).
  - Add the start node to the queue.
  - Dequeue a node and examine it.
  - If it's the goal, you're done!
  - If not, add all its unvisited neighbors to the back of the queue.
  - Repeat until the queue is empty or the goal is found.
- **Key Properties**:
  - **Complete**: Yes, it will always find a solution if one exists.
  - **Optimal**: Yes, but only if all path costs are equal (e.g., each step costs 1). It finds the shallowest solution.
  - **Complexity**: Time and space complexity are high, O(bd), where b is the branching factor and d is the depth of the solution. It requires a lot of memory.

Depth-First Search (DFS)

DFS explores the state space by going as **deep as possible** down one path before backtracking.

- **How it works**: It uses a **Stack** (Last-In, First-Out).
    - Add the start node to the stack.
    - Pop a node and examine it.
    - If it's the goal, you're done!
    - If not, add all its unvisited neighbors to the top of the stack.
    - Repeat until the stack is empty or the goal is found.
- **Key Properties**:
    - **Complete**: No, it can get stuck in infinite loops if the state space is infinite. (Can be made complete with a visited list).
    - **Optimal**: No, it finds the "leftmost" solution, which is not guaranteed to be the shortest.
    - **Complexity**: Time complexity is O(bm) (`m` is max depth), but its space complexity is much better than BFS, O(b·m).

Uniform Cost Search (UCS)

UCS is a variation of Dijkstra's algorithm. It explores the state space by always expanding the node with the **lowest total path cost** from the start node.

- **How it works**: It uses a **Priority Queue**, ordered by path cost (`g(n)`).
- **Key Properties**:
    - **Complete**: Yes.
    - **Optimal**: Yes, it is guaranteed to find the solution with the lowest cost.
    - **Difference from BFS**: While BFS finds the shallowest solution, UCS finds the cheapest one, even if it's deeper.

Depth-Limited Search (DLS)

This is simply DFS with a pre-defined **depth limit**. It stops going down a path once it reaches that limit, forcing it to backtrack. This solves the infinite loop problem of DFS.

- **Key Properties**:
    - **Complete**: No, if the shallowest solution is beyond the depth limit, it won't be found.
    - **Optimal**: No.

Iterative Deepening Search (IDS)

IDS combines the best of both worlds: the **low memory usage of DFS** and the **completeness/optimality of BFS**.

- **How it works**: It repeatedly runs DLS with an increasing depth limit. It first does a DLS with limit 0, then limit 1, then limit 2, and so on, until a solution is found.
- **Key Properties**:
    - **Complete**: Yes.

- **Optimal**: Yes (for unit-cost paths).
- **Why it's efficient**: While it seems wasteful to re-explore nodes, the majority of the nodes are at the bottom level of the search tree. So, the overhead of re-generating upper-level nodes is relatively small.

Informed (Heuristic) Search

These algorithms use problem-specific knowledge to guide the search. They use a **heuristic function**, denoted as h(n), which **estimates the cost** of the cheapest path from the current state n to a goal state. Think of this as having a compass in the maze that always points toward the exit.

Greedy Best-First Search

This algorithm tries to expand the node that is believed to be **closest to the goal**. It's "greedy" because it only considers this estimated cost, ignoring the cost already incurred to get to the current node.

- **How it works**: It uses a priority queue ordered by the heuristic, h(n). It always expands the node with the lowest h(n).
- **Key Properties**:
    - **Complete**: No. It can get stuck in loops.
    - **Optimal**: No. It can be tricked into taking a path that seems good at first but is actually a dead-end or very long.
    - **Fast**: It's very fast but trades optimality for speed.

A* Algorithm

A* (A-star) is one of the most popular and powerful search algorithms. It combines the strengths of UCS and Greedy Best-First Search. It evaluates nodes by combining the **cost to reach the node** (`g(n)`) and the **estimated cost to get from the node to the goal** (`h(n)`).

- **Evaluation Function**: f(n)=g(n)+h(n)
    - g(n): The actual cost from the start node to node n (like in UCS).
    - h(n): The estimated cost from node n to the goal (the heuristic).
- **How it works**: It uses a priority queue ordered by the lowest f(n) value.
- **Key Properties**:
    - **Complete**: Yes.
    - **Optimal**: Yes, provided the heuristic h(n) is **admissible** (it never *overestimates* the true cost to the goal) and, for graphs, **consistent**.

IDA* (Iterative Deepening A*)

This is the iterative deepening version of A*. It's a memory-efficient way to implement A* search.

- **How it works**: It works like IDS, but instead of a depth limit, it uses an **`f-cost` limit**. It starts with a cost limit equal to the f(n) of the start node. In each iteration, it performs a DFS, pruning any path that exceeds the current cost limit. If no solution is found, the new limit becomes the smallest cost that was pruned in the previous iteration.
- **Key Properties**: It has the same properties as A* (complete, optimal) but with the low memory requirements of DFS.

Hill Climbing

This is a **local search** algorithm, not a pathfinding one. It's used for optimization problems where you want to find the best state (e.g., the "highest peak").

- **How it works**: It's a "greedy" local search. It starts with a random solution and then iteratively makes small changes (moves to a neighbor state). It only accepts the change if the new state is better than the current one.
- **Key Problem**: It can get stuck in **local maxima**. It finds a "peak," but it might not be the highest peak in the entire landscape. It has no way of going "downhill" to explore other parts of the search space.

Beam Search

Beam search is a heuristic search algorithm that explores a graph by expanding the most promising nodes in a limited set.

- **How it works**: It's a variation of Best-First Search that limits the number of nodes it keeps open at each level. It has a parameter `k` called the **beam width**. At each step of the search, it generates all successors of the current `k` states but only keeps the best `k` successors for the next step.
- **Key Properties**: It's not complete or optimal, but it's very memory efficient. It's a trade-off between a greedy search and a complete search. If k=1, it's like Hill Climbing. If k=∞, it's like Best-First Search.