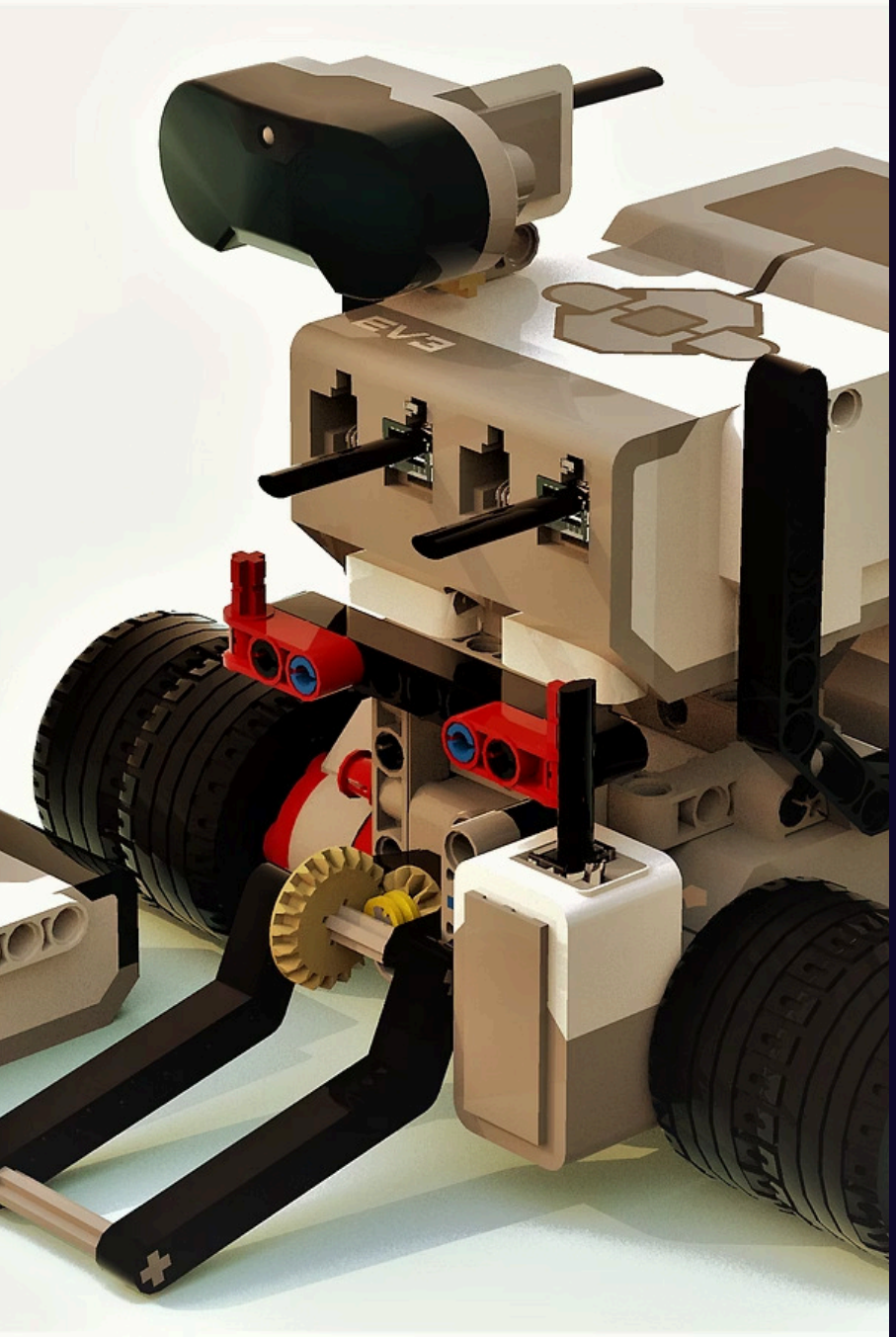

Artificial Intelligence: Problem-Solving Through Search

Welcome to Week 2 of your AI journey! This week, we're diving into one of the most fundamental concepts in artificial intelligence: how computers solve problems by searching. Think of it like finding your way through a maze or searching for the best route to your destination. In this lesson, we'll explore the core techniques that power AI systems to tackle complex problems efficiently.

Understanding Problem-Solving in AI

Problem-solving in AI is the process of finding a solution to a task by exploring different possibilities. When we face a problem, we need to find the best path from where we start to where we want to be.

Imagine you're at the entrance of a library trying to find a specific book. You could randomly wander through the aisles (not efficient!), or you could use the library's system to search strategically. AI does something similar—it explores options systematically to find the best answer.

Initial State

Where the problem starts

Actions

What moves are possible

Goal State

Where we want to reach

Path Cost

The effort it takes

Key Insight: Every problem-solving task involves finding a sequence of actions that transforms the initial state into the goal state efficiently.

Search Algorithms: The Foundation of AI Problem-Solving

Search algorithms are step-by-step procedures that explore all possible solutions to find the best one. They're like having a map and a strategy for finding your way through unknown territory.

There are two main categories of search algorithms that form the foundation of AI:

Uninformed Search

Also called "blind search," these algorithms have no idea where the goal is. They explore systematically without using any special knowledge about the problem. Like searching for lost keys in a house without knowing which room they're in.

Informed Search

These algorithms use special knowledge (called heuristics) to guide their search toward the goal more efficiently. Like searching for keys while remembering you last used them in the bedroom.

In this lesson, we focus on uninformed searches because they teach us the fundamental techniques that all other algorithms build upon.

Uninformed Search Algorithms: Exploring Without Clues

Uninformed search algorithms (also called blind search) explore the problem space without any special knowledge about where the solution might be. They simply follow rules about how to explore, visiting states in a specific order until they find the goal.

These algorithms don't know how close they are to the solution—they just keep searching. Imagine trying to find someone in a crowded stadium without knowing where they are. You have a system for searching, but no hints about the actual location.

Characteristics

- No advance knowledge of the problem structure
- Uses only the problem definition
- Explores systematically
- Guaranteed to find a solution if one exists
- Often slower than informed methods

Common Types

- Depth-First Search (DFS)
- Breadth-First Search (BFS)
- Uniform Cost Search
- Iterative Deepening

Why Learn These? Understanding uninformed searches builds the foundation for more advanced algorithms. Many complex AI techniques are refinements of these basic strategies.

Depth-First Search (DFS): Going Deep

Depth-First Search (DFS) is a search algorithm that explores as far as possible along each branch before backtracking. Think of it like exploring a cave: you go all the way down one tunnel, and if it's a dead end, you come back and try another tunnel.

How DFS Works:

1 Start at the beginning node

Begin your exploration from the starting point

2 Go deep into one path

Follow one branch all the way down until you hit a dead end or find the goal

3 Backtrack and explore another path

Return to the last decision point and try a different direction

4 Repeat until goal is found

Continue this process until you've found what you're looking for

Real-World Example: Finding a file on your computer by going into folders one by one, going as deep as possible before coming back up to explore other folders.

Implementation: DFS uses a data structure called a **stack** (Last In, First Out) to keep track of which nodes to visit next.

Depth-First Search: Code Example

Here's how DFS looks in simple Python code:

```
def depth_first_search(graph, start, goal):
    """
    Performs DFS to find goal from start node
    graph: dictionary showing connections
    start: beginning node
    goal: what we're looking for
    """
    visited = []  # Track which nodes we've seen
    stack = [start]  # Start with our beginning node

    while stack:  # While there are nodes to explore
        node = stack.pop()  # Get the last node from stack

        if node not in visited:
            visited.append(node)  # Mark as visited
            print(f"Visiting: {node}")

            if node == goal:
                return f"Found goal: {goal}"

            # Add unvisited neighbors to stack
            for neighbor in graph[node]:
                if neighbor not in visited:
                    stack.append(neighbor)

    return "Goal not found"

# Example: Finding a path through a network
network = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['G'],
    'F': [],
    'G': []
}

result = depth_first_search(network, 'A', 'G')
print(result)
```

Key Points: Notice how we use a **stack** to manage which nodes to explore next. The `pop()` function takes from the end, which gives us the most recently added node—that's what makes it "depth-first"!

Breadth-First Search (BFS): Exploring Layer by Layer

Breadth-First Search (BFS) explores all nodes at the current level before moving to the next level. Imagine exploring a building floor by floor—you completely explore Floor 1, then Floor 2, then Floor 3, and so on.

How BFS Works:

1 Start at the beginning node

Begin your exploration from the starting point

2 Explore all immediate neighbors

Visit all nodes directly connected to your starting point

3 Move to the next layer

Now explore all nodes connected to those neighbors

4 Continue level by level

Keep expanding outward until you find the goal

Real-World Example: In a social network, BFS helps find the shortest connection between two people. First check direct friends, then friends of friends, and so on.

Implementation: BFS uses a data structure called a **queue** (First In, First Out) to keep track of which nodes to visit next. This ensures we explore level-by-level.

Breadth-First Search: Code Example

Here's how BFS looks in simple Python code:

```
from collections import deque

def breadth_first_search(graph, start, goal):
    """
    Performs BFS to find goal from start node
    graph: dictionary showing connections
    start: beginning node
    goal: what we're looking for
    """
    visited = []  # Track which nodes we've seen
    queue = deque([start])  # Start with our beginning node
    visited.append(start)

    while queue:  # While there are nodes to explore
        node = queue.popleft()  # Get first node from queue
        print(f"Visiting: {node}")

        if node == goal:
            return f"Found goal: {goal}"

        # Add unvisited neighbors to queue
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)

    return "Goal not found"

# Example: Finding a path through a network
network = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['G'],
    'F': [],
    'G': []
}

result = breadth_first_search(network, 'A', 'G')
print(result)
```

Key Differences: BFS uses a **queue** (with `deque` for efficiency) and `popleft()` instead of `pop()`. This takes from the front of the line, ensuring we explore level-by-level rather than going deep.

DFS vs BFS: Comparing Search Strategies

Both DFS and BFS are fundamental search algorithms, but they have different strengths and weaknesses. Let's compare them across important factors:

Factor	Depth-First Search	Breadth-First Search
Memory Usage	Low (uses less memory)	High (stores more nodes)
Speed to Goal	Can be slow if goal is on a shallow level	Finds shortest path first
Best For	Limited memory, exploring all paths, detecting cycles	Finding shortest solution, searching shallow trees
Backtracking	Goes deep before backtracking	Backtracks level by level
Data Structure	Stack (LIFO)	Queue (FIFO)
Completeness	Complete if depth is finite	Always complete

Choice Matters: Selecting between DFS and BFS depends on your specific problem. For finding the shortest path, BFS is superior. For problems with limited memory or where you need to explore all possibilities, DFS is better.

Key Takeaways: Your AI Problem-Solving Foundation

Congratulations! You've learned the fundamental search algorithms that power AI systems. Here's what you should remember:

Problem-Solving Foundation

AI solves problems by systematically searching from an initial state to a goal state, exploring possible actions and paths.

Uninformed Search Basics

These algorithms explore without special knowledge about the problem, following systematic rules to find solutions.

DFS: Going Deep

Depth-First Search explores fully down one path before backtracking, using a stack and minimal memory.

BFS: Layer by Layer

Breadth-First Search explores level-by-level, using a queue and guaranteeing the shortest path is found first.

Practical Applications

These search techniques power navigation systems, social networks, puzzle solvers, and countless AI applications.

Next Steps: Next week, we'll explore informed search algorithms that use hints and knowledge to search even more efficiently. For now, practice implementing DFS and BFS with different problem structures—the best way to truly understand these algorithms is to code them yourself!