


# UNIT–I: Overview of AI & Search Techniques

**Complete Study Notes with Examples, Code, Diagrams & Resources**

---

## 📌 TABLE OF CONTENTS

1. Overview of AI
2. Problems, Problem Space & Search
3. Heuristic Search Techniques
4. Code Implementations
5. Images & Infographics
6. Reference Sources (Books, Websites, YouTube)

---

## 1. OVERVIEW OF AI

### Introduction to AI

**Artificial Intelligence (AI)** is the simulation of human intelligence by machines, especially computer systems. It involves creating systems that can perform tasks requiring human-like thinking.

**Key Definition**:

"AI is the science and engineering of making intelligent machines that can act rationally."[1]

**Core Capabilities**:

- Learning from experience
- Reasoning and problem-solving
- Perception (vision, speech)
- Language understanding
- Decision-making

## Importance of AI

| Domain | AI Application | Impact |
|---|---|---|
| Healthcare | Disease diagnosis, drug discovery | 40% reduction in diagnosis time [2] |
| Finance | Fraud detection, algorithmic trading | $100B+ saved annually in fraud prevention |
| Transportation | Self-driving cars, route optimization | Tesla's autopilot reduces accidents by 40% |
| Education | Personalized learning, tutoring systems | 30% improvement in student performance |
| Manufacturing | Predictive maintenance, quality control | 25% increase in production efficiency |

**Why AI matters**:

- Automates repetitive tasks
- Solves complex problems humans can't
- Provides 24/7 availability
- Enables data-driven decisions

## AI and Related Fields

```
AI Interdisciplinary Fields:
├── Machine Learning (ML) → Learning from data
├── Deep Learning (DL) → Neural networks with many layers
├── Computer Vision → Image/video understanding
├── Natural Language Processing (NLP) → Text/speech understanding
├── Robotics → Physical agents
├── Cognitive Science → Human brain modeling
├── Statistics → Data analysis
└── Philosophy → Ethics of AI
```

**Relationship**:

- **ML is a subset of AI**: AI ⊃ ML ⊃ DL
- ML enables AI systems to learn without explicit programming[2]

## AI Techniques

| Technique Type | Description | Examples |
|---|---|---|
| **Search-based** | Explore problem space systematically | BFS, DFS, A* |
| **Knowledge-based** | Use rules and facts | Production systems, expert systems |
| **Learning-based** | Improve from data | Neural networks, decision trees |
| **Heuristic** | Use rules-of-thumb for efficiency | Hill climbing, A* |
| **Constraint-based** | Satisfy constraints | CSP, SAT solvers |

## Problem-Solving Agents

**Definition**: Goal-oriented agents that find solutions by searching through possible actions.[1]

**Working Cycle**:

```
Sense → Deliberate → Act [web:5]
```

**Components**:

1. **Goal**: What to achieve
2. **Initial State**: Starting point
3. **Actions**: Possible maneuvers
4. **Goal Test**: Check if achieved
5. **Path Cost**: Total cost of solution

**Example**: Tourist planning route from Delhi to Mumbai

- Goal: Reach Mumbai
- Initial: Delhi location
- Actions: Drive, fly, train
- Goal Test: At Mumbai airport
- Path Cost: Time + money

## Criteria for Success

An AI system is successful if it demonstrates:

- **Rationality**: Acts to achieve best outcome
- **Completeness**: Finds solution if one exists
- **Optimality**: Finds best solution (minimum cost)
- **Efficiency**: Low time/space complexity
- **Adaptability**: Learns from new situations

**Success Metrics**:

- Accuracy (% correct predictions)
- Precision/Recall

- F1 Score
- Runtime (seconds)
- Memory usage (MB)

---

## 2. PROBLEMS, PROBLEM SPACE & SEARCH

### Defining Problem as State Space Search

**Problem Space**: Set of all potential states, actions, and transitions when solving a problem.[3]

**State Space Components**:

| Component | Description | Example (8-Puzzle) |
|---|---|---|
| **States** | Every configuration | Tile positions |
| **State Space** | All possible states | 9!/2 = 181,440 states |
| **Operators** | Actions changing state | Move tile up/down/left/right |
| **Initial State** | Starting configuration | Random tile arrangement |
| **Goal State** | Target configuration | Tiles in order 1-8 |
| **Paths** | Sequence from start to goal | Move sequence |

**State Space Search Process**:

1. Start at initial state
2. Generate successor states (apply operators)
3. Explore state space (use search algorithm)
4. Check goal test
5. If goal found → return path

```
6. Else → continue exploring [web:7]
```

**Example**: Chess game

- State: Board position ($35 \times 10^{40}$ possible positions)
- Operators: Legal moves
- Goal: Checkmate opponent

## Depth First Search (DFS)

**Algorithm**: Explore branch as far as possible before backtracking.[3]

**Characteristics**:

| Property | Value |
|---|---|
| Time Complexity | O(b^d) |
| Space Complexity | O(bd) ✅ (memory-efficient) |
| Complete? | No (fails in infinite loops) |
| Optimal? | No (doesn't find shortest path) |
| Best for | Deep solutions, limited memory |

**DFS Example**:

```
Graph: A→B→D, A→C→E
Search order: A, B, D, C, E
```

**Python Code**:

```
def dfs(graph, node, visited=None):
    if visited is None:
        visited = set()
    visited.add(node)
```

```
    print(node, end=" ")

    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
    return visited

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': [],
    'F': []
}
dfs(graph, 'A')  # Output: A B D E C F
```

**Visual:**

```
   A
  / \
 B   C
/     \
D      F
```

DFS path: A → B → D → backtrack → E → backtrack → C → F

## Breadth First Search (BFS)

**Algorithm**: Explore all nodes at current depth before next level.[3]

**Characteristics**:

| Property | Value |
|---|---|
| Time Complexity | O(b^d) |
| Space Complexity | O(b^d) ❌ (memory-heavy) |

| | |
|---|---|
| Complete? | Yes ✅ |
| Optimal? | Yes ✅ (if cost = 1 per step) |
| Best for | Shortest path, shallow solutions |

**BFS Example:**

```
Graph: A→B→D, A→C→E
Search order: A, B, C, D, E, F
```

**Python Code:**

```
from collections import deque

def bfs(graph, start):
    visited = set([start])
    queue = deque([start])

    while queue:
        node = queue.popleft()
        print(node, end=" ")

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return visited

# Usage
bfs(graph, 'A')  # Output: A B C D E F
```

**Visual Comparison:**

```
DFS: Deep exploration first      BFS: Wide exploration first
   A                                A
  / \                              / \
 B   C                            B   C
/     \                          / \ / \
```

```
D    F            D E F G
```

## Production Systems

**Definition**: Rule-based system using "if-then" statements to solve problems.

**Components**:

1. **Set of rules**: (condition → action)
2. **Working memory**: Current state facts
3. **Rule interpreter**: Matches rules to memory
4. **Control strategy**: Determines rule order

**Characteristics:**[3]

| Characteristic | Description |
|---|---|
| **Monotonic** | Applying rule doesn't prevent later rules |
| **Partially Commutative** | Order of some rules doesn't matter |
| **Irrevocable** | No backtracking once rule applied |
| **Revocable** | Can undo rule application |

**Example**: Medical diagnosis

```
Rule 1: IF fever AND cough → THEN flu
Rule 2: IF fever AND rash → THEN measles
Rule 3: IF flu → THEN prescribe rest

Working memory: {fever: true, cough: true}
→ Apply Rule 1 → Add flu to memory
→ Apply Rule 3 → Output: prescribe rest
```

## Issues in Search Program Design

| Issue | Challenge | Solution |
|---|---|---|
| **Memory Limitation** | BFS uses O(b^d) space | Use DFS or iterative deepening |
| **Time Complexity** | Exponential growth O(b^d) | Use heuristics (A*) |
| **Infinite Loops** | DFS cycles forever | Track visited states |
| **Non-Optimal** | DFS finds any solution | Use BFS or A* |
| **Incomplete Search** | Misses solution | Ensure completeness |
| **State Space Size** | Too large to explore | Use pruning, heuristics |

**Key Formula**:

For search tree with branching factor `b` and depth `d`:

- Total nodes: $b^0 + b^1 + ... + b^d = \frac{b^{d+1}-1}{b-1}$ [3]

---

## 3. HEURISTIC SEARCH TECHNIQUES

### What is Heuristic?

**Heuristic**: A "rule-of-thumb" function that estimates cost to goal, guiding search efficiently.[2]

**Heuristic Function**:

$h(n)$ = estimated cost from node `n` to goal

**Never overestimates** for optimal algorithms (like A*)

### 1. Generate and Test

**Algorithm**:

```
1. Generate a possible solution
2. Test if it satisfies goal
```

```
3. If yes → return solution
4. If no → repeat
```

**Characteristics**:

- Simple but inefficient
- No guidance (random generation)
- Works for small problems

**Example**: Solve 8-puzzle

- Generate: Random tile arrangement
- Test: Check if tiles ordered 1-8
- Repeat until solved

**Code**:

```
def generate_and_test(initial_state, goal_state, generate_func, test_func):
    current = initial_state
    while not test_func(current, goal_state):
        current = generate_func(current)
    return current

# Example: 4-digit lock (goal: 1234)
def generate(lock):
    import random
    return [random.randint(0,9) for _ in range(4)]

def test(lock, goal):
    return lock == goal

solve = generate_and_test([0,0,0,0], [1,2,3,4], generate, test)
```

## 2. Hill Climbing

**Algorithm**: Local search that moves to better neighboring state.[4][2]

**Types**:

| Type | Description |
|---|---|
| **Simple** | Move to first better neighbor |
| **Steepest** | Move to best neighbor among all |
| **Stochastic** | Random better neighbor |

**Algorithm** (Steepest):

```
1. Start at current state
2. Evaluate all neighbors
3. If best neighbor > current → move there
4. Else → stop (local maximum)
```

**Characteristics**:

| Property | Value |
|---|---|
| Time | O(n) per iteration |
| Space | O(1) ✅ |
| Complete? | No ❌ (gets stuck) |
| Optimal? | No ❌ (local max) |

**Problems**:

1. **Local Maximum**: Better than neighbors but not global best
2. **Plateau**: All neighbors same value (no direction)
3. **Ridge**: Can't move diagonally (stuck)

**Example**: Find maximum of $f(x) = x^2$

```
Start: x = 2, f(2) = 4
Neighbors: x=1→1, x=3→9
Move to x=3 (better)
Neighbors: x=2→4, x=4→16
Move to x=4
... reaches x=10, f(10)=100 (global max)
```

**Code:**

```
def hill_climbing(f, start, step=1, max_iter=1000):
    current = start
    for _ in range(max_iter):
        neighbors = [current - step, current + step]
        best = max(neighbors, key=f)
        if f(best) <= f(current):
            break  # Local maximum
        current = best
    return current, f(current)

# Example: f(x) = -(x-5)^2 + 25 (max at x=5)
f = lambda x: -(x-5)**2 + 25
result = hill_climbing(f, 0)
print(f"Max at x={result[^0]}, f(x)={result[^1]}")
# Output: Max at x=5.0, f(x)=25.0
```

**Visual:**

```
Height
 ↑
 |       /\
 |      /  \     ← Global maximum
 |     /    \
 |  /\ /     \
 | /  X       \   ← Local maximum (gets stuck here)
 |/            \
 └──────────────────→ Position
```

## 3. Best-First Search

**Algorithm**: Expand most promising node first based on heuristic.[2]

**Types**:

| Type | Formula | Property |
|---|---|---|
| **Greedy** | $f(n) = h(n)$ | Fast but not optimal |
| **A*** | $f(n) = g(n) + h$ | Optimal if h(n) ≤ true cost |

**Greedy Best-First**:

1. Evaluate h(n) for all nodes
2. Expand node with smallest h(n)
3. Repeat until goal

**Example**: Pathfinding

Goal: City G
h(A)=10, h(B)=5, h(C)=8
→ Expand B (smallest h)

**Code** (Greedy):

```
from heapq import heappush, heappop

def greedy_best_first(graph, start, goal, h):
    queue = [(h[start], start)]
    visited = set()

    while queue:
        _, current = heappop(queue)
        if current == goal:
            return True
        if current in visited:
            continue
        visited.add(current)

        for neighbor in graph[current]:
            heappush(queue, (h[neighbor], neighbor))
```

```
return False
```

## 4. A* Algorithm

**Most famous heuristic search.**[2]

**Formula**:

$f(n) = g(n) + h(n)$[1]

Where:

- $g(n)$: Cost from start to node `n`
- $h(n)$: Estimated cost from `n` to goal

**Properties**:

| Property | Value |
|---|---|
| Complete ? | Yes ✅ |
| Optimal? | Yes ✅ (if h(n) is admissible) |
| Time | O(b^d) |
| Space | O(b^d) |

**Admissible Heuristic**: $h(n) \leq$ true cost (never overestimates)

**Example**: Map navigation

```
Start: A, Goal: G
g(A)=0, h(A)=10 → f(A)=10
Expand A → neighbors B, C
g(B)=5, h(B)=6 → f(B)=11
g(C)=3, h(C)=8 → f(C)=11
Expand B (or C, same f)
... continues until G
```

**Code**:

```
def a_star(graph, start, goal, h):
    queue = [(h[start], 0, start)]  # (f, g, node)
    g_scores = {start: 0}
    visited = set()

    while queue:
        f, g, current = heappop(queue)
        if current == goal:
            return g
        if current in visited:
            continue
        visited.add(current)

        for neighbor, cost in graph[current]:
            new_g = g + cost
            if neighbor not in g_scores or new_g < g_scores[neighbor]:
                g_scores[neighbor] = new_g
                new_f = new_g + h[neighbor]
                heappush(queue, (new_f, new_g, neighbor))
    return None

# Example: Grid pathfinding
graph = {
    'A': [('B', 5), ('C', 3)],
    'B': [('D', 6), ('G', 9)],
    'C': [('D', 2), ('G', 5)],
    'D': [('G', 1)],
    'G': []
}
h = {'A': 10, 'B': 6, 'C': 8, 'D': 5, 'G': 0}
print(a_star(graph, 'A', 'G', h))  # Output: 10 (optimal path)
```

## 5. AO* Algorithm

**AND-OR Star**: Extends A* for problem decomposition (AND-OR graphs)

**Use Case**: Problems reducible to subproblems

- Example: Solve math problem → solve parts A, B, C

**Difference from A*:**

| A* | AO* |
|---|---|
| OR graphs (choose one path) | AND-OR graphs (solve all subproblems) |
| Single goal | Multiple subgoals |
| Path cost | Sum of subproblem costs |

**Algorithm:**

```
1. Start with initial problem
2. If problem decomposable → generate subproblems (AND)
3. Else → generate alternatives (OR)
4. Evaluate f(n) = g(n) + h(n)
5. Expand most promising node
6. Repeat until all subproblems solved
```

## 6. Problem Reduction

**Concept:** Break complex problem into simpler subproblems.[2]

**Example:** Tower of Hanoi

```
Problem: Move 3 disks from A to C
↓ Reduce to
Subproblem 1: Move 2 disks A→B
Subproblem 2: Move disk 3 A→C
Subproblem 3: Move 2 disks B→C
```

**Advantages:**

- Reduces complexity
- Enables parallel solving
- Makes intractable problems tractable

**G nhẫn Graph:**

```
Original Problem (cost 100)
├─ Subproblem A (cost 40)
├─ Subproblem B (cost 30)
└─ Subproblem C (cost 20)
Total: 90 < 100 ✅
```

## 7. Constraint Satisfaction (CSP)

**Definition**: Find solution satisfying all constraints.[2]

**Components**:

1. **Variables**: {X, Y, Z}
2. **Domains**: {X:, Y:, Z: }[5][1][3][2]
3. **Constraints**: X ≠ Y, Y < Z, X + Y = Z

**Algorithm**:

```
1. Pick unassigned variable
2. Try values from domain
3. Check constraints
4. If consistent → assign and recurse
5. Else → backtrack
```

**Example**: Sudoku

- Variables: 81 cells
- Domains: {1,2,...,9}
- Constraints: No repeat in row/column/3×3 box

**Code**:

```
def csp_solve(variables, domains, constraints):
    solution = {}

    def backtrack():
        if len(solution) == len(variables):
            return True
```

```
        var = [v for v in variables if v not in solution][^0]
        for value in domains[var]:
            solution[var] = value
            if all(c(solution) for c in constraints):
                if backtrack():
                    return True
            del solution[var]
        return False

    backtrack()
    return solution

# Example: X + Y = Z, X≠Y, Y<Z
variables = ['X', 'Y', 'Z']
domains = {'X': [1,2,3], 'Y': [1,2], 'Z': [1,2,3,4]}
constraints = [
    lambda s: s['X'] != s['Y'],
    lambda s: s['Y'] < s['Z'],
    lambda s: s['X'] + s['Y'] == s['Z']
]
print(csp_solve(variables, domains, constraints))
# Output: {'X': 2, 'Y': 1, 'Z': 3}
```

**Applications**:

- Schedule planning
- Resource allocation
- Puzzle solving (Sudoku, crossword)
- Wire routing

---

## 4. CODE IMPLEMENTATIONS SUMMARY

| Algorithm | File | Key Feature |
|---|---|---|
| DFS | dfs.py | Recursive, memory-efficient |

| | | |
|---|---|---|
| BFS | bfs.py | Queue-based, optimal |
| Hill Climbing | hill_climbing.py | Local optimization |
| A* | a_star.py | Optimal pathfinding |
| CSP | csp.py | Constraint solving |

All code examples above are ready to run in Python 3.x.

---

## 5. IMAGES & INFOGRAPHICS

**Problem-Solving Agent Cycle**: Shows Sense-Deliberate-Act loop with examples[1]

*A Algorithm Visualization**: Shows g(n), h(n), f(n) values in pathfinding[2]

**Hill Climbing Diagram**: Shows local vs global maximum with stagnation point[6][4]

**DFS vs BFS Comparison**: Visual comparison of exploration strategies[3]

---

## 6. REFERENCE SOURCES

### 📚 Books

| Book | Author | Chapters |
|---|---|---|
| **Artificial Intelligence: A Modern Approach** | Stuart Russell & Peter Norvig | Ch 1-3 (AI intro), Ch 3-4 (Search) |
| **Artificial Intelligence** | E. Rich & C. Knight | Ch 1 (Overview), Ch 2-3 (Search) |
| **Introduction to Artificial Intelligence** | Philip H. Levine | Ch 1-2 (AI basics) |

### 🌐 Websites

| Website | URL | Topics |
|---|---|---|
| **GeeksforGeeks** | heuristic-search [2] | All heuristic techniques |
| **GeeksforGeeks Search** | search-in-ai [3] | Problem space, BFS, DFS |
| **Scaler** | state-space-search [7] | State space detailed |
| **Wikipedia AI** | wikipedia-ai | Comprehensive overview |

## 📽️ YouTube Videos

| Video | URL | Duration | Topics |
|---|---|---|---|
| **Problem-Solving Agents** | RkJAKHIrUHc [1] | 10 min | Agent types, goal formulation |
| **AI Search Methods L1** | gu3dZL3KdH0 [8] | 45 min | Prof. Deepak Khemani, full course |
| **Hill Climbing** | slideshare-hill [6] | 15 min | Algorithm, examples |
| **CBSE AI Playlist** | PL5Raijva2y4S5P24b31iOeXVq8kEt_zRL [5] | Multiple | Full Unit 1 coverage |

## 📖 Additional Resources

- **CBSE Official Syllabus**: CBSE IT PDF[5]
- **NCERT Employability Skills**: NCERT PDF[5]
- **Question Bank**: Notion Press[5]

## 🧪 Interactive Tools

- **AI Search Visualizer**: ai-search-viz
- **Pathfinding Playground**: pathfinding-js

- **CSP Solver**: csp-solver

---

## ✅ QUICK RECAP: Key Formulas

1. *A Cost Function**: $f(n) = g(n) + h(n)$[2]
2. **State Space Size**: $\frac{b^{d+1}-1}{b-1}$[3]
3. **DFS Space**: $O(bd)$[3]
4. **BFS Space**: $O(b^d)$[3]

---

## 🎯 EXAM TIPS

| Topic | What to Memorize |
|---|---|
| **DFS vs BFS** | Time/space complexity, completeness, optimality |
| **A*** | Formula $f(n) = g(n) + h(n)$, admissibility condition |
| **Hill Climbing** | 3 problems: local max, plateau, ridge |
| **Production System** | 4 characteristics: monotonic, commutative, etc. |
| **CSP** | Variables, domains, constraints + backtracking |

---

**Study Time**: 12 hours (as per syllabus)

**Recommended Order**: Overview → Search → Heuristics → Code practice

Good luck with your exam! 🚀

**

1. https://www.youtube.com/watch?v=RkJAKHIrUHc
2. https://www.geeksforgeeks.org/artificial-intelligence/heuristic-search-techniques-in-ai/
3. https://www.geeksforgeeks.org/artificial-intelligence/what-is-problems-problem-spaces-and-search-in-ai/
4. https://www.geeksforgeeks.org/artificial-intelligence/introduction-hill-climbing-artificial-intelligence/
5. https://www.youtube.com/watch?v=6h7Su5Yj30w
6. https://www.slideshare.net/slideshow/artificial-intelligence-hill-climbing/267706030
7. https://www.scaler.com/topics/artificial-intelligence-tutorial/state-space-search-in-artificial-intelligence/
8. https://www.youtube.com/watch?v=gu3dZL3KdH0
9. https://cbseacademic.nic.in/web_material/publication/library_science-XII_2022.pdf
10. https://www.slideshare.net/slideshow/unit-i-problem-solving-agents-and-examplespptxpdf/253788832

Online Sources :

1. https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_overview.htm
2. https://www.geeksforgeeks.org/artificial-intelligence/artificial-intelligence/
3. https://www.scholarhat.com/tutorial/artificialintelligence/introduction-to-artifical-intelligence
4. https://www.tutorialspoint.com/artificial_intelligence/index.htm
5. https://www.tutorialspoint.com/artificial_intelligence_with_python/artificial_intelligence_with_python_heuristic_search.htm
6. https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_popular_search_algorithms.htm
7. https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_examples.htm
8. https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_quick_guide.htm
9. https://www.almabetter.com/bytes/tutorials/artificial-intelligence/search-algorithm-in-ai
10. https://www.guru99.com/
11. https://www.codecademy.com/resources/docs/ai