Beni-Suif National University
College of Computers and AI

Lab Manual
UCCS3105
Artificial Intelligence

Dr. Noha Yahya
T.A. Abdelrahman Hashem

What We'll Build This Course

Weekly Breakdown

Week 1: Introduction to AI

- **History, definition, applications, and scope.**
- **AI vs Human Intelligence.**
- **Lab: Install Python, Jupyter, AI libraries (NumPy, Pandas, Scikit-learn).**

Week 2: Intelligent Agents

- **Types of agents, environments, agent architecture.**
- **Lab: Build a simple rule-based agent.**

Week 3,4: Search Strategies (Uninformed Search)

- **BFS, DFS, Uniform Cost Search.**
- **Lab: Implement search algorithms (maze-solving).**

Week 5: Search Strategies (Informed Search & Optimization)

- **A*, Greedy.**
- **Lab: Solve an 8-puzzle problem using A*.**

Week 6 Local Search

- **Hill-climbing search.**

Week 7: Genetic Algorithm

Week 8: Adversarial Search

- **The Game Tree**

Week 9: CSPs

- **Forward checking, Arc consistency (AC-3).**
- **MRV and LCV.**

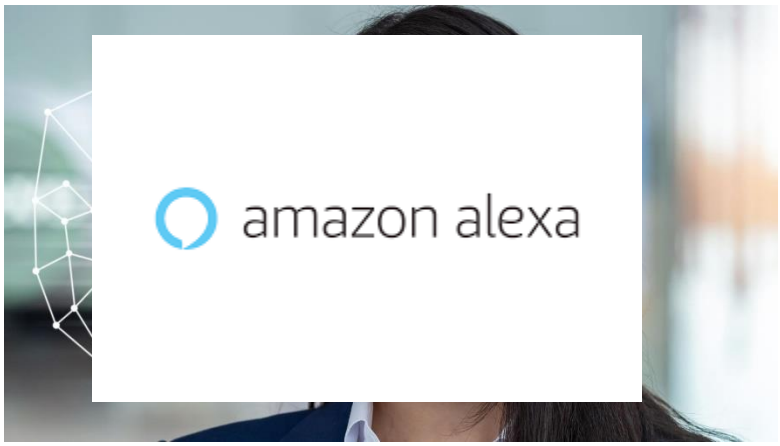
Week 10: Backtracking Search for CSPs

Lab - 1

Introduction to Neural Network

What is Artificial Intelligence?

- In simple terms: **Making computers think like humans.**
- It's about creating machines that can perform tasks that usually require human intelligence.
- **Examples:** Playing chess, recognizing faces in photos, recommending movies on Netflix, or understanding voice commands (like Siri or Alexa).

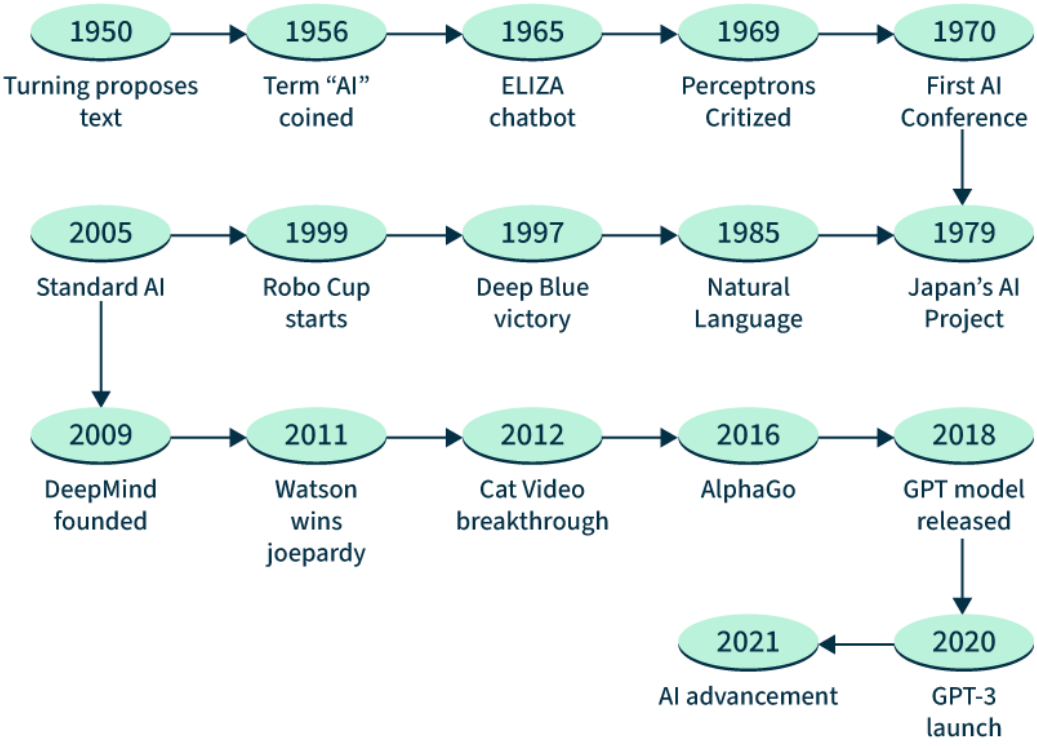


Brief History of AI

Timeline:

- **1950s** : The idea is born. Alan Turing asks, "Can machines think?"
- **1956** : The term "Artificial Intelligence" is officially coined.
- **1997** : AI beats the world chess champion (Deep Blue).
- **2010s-Present** : AI is everywhere! Self-driving cars, smart assistants, and generative AI like ChatGPT.

Evolution of AI



AI vs. Human Intelligence

Artificial Intelligence (AI)	Human Intelligence
Processes information using algorithms and data patterns.	Processes information using reasoning, emotions, intuition, and experiences.
Learns from large datasets; performance depends on data quality.	Learns from real-world experiences, context, and social interactions.
Executes tasks quickly, consistently, and without fatigue.	Limited by physical and mental fatigue, but adaptable and creative.
Excels at repetitive tasks, pattern recognition, and computation.	Excels at abstract thinking, creativity, and complex problem-solving.
Lacks self-awareness, emotions, and consciousness.	Possesses self-awareness, emotions, consciousness, and empathy.
Cannot truly understand meaning—interprets patterns statistically.	Understands meaning, context, humor, irony, and subtle cues naturally.
Highly scalable—can operate across many machines simultaneously.	Limited to a single biological brain.
Requires power, hardware, maintenance, and updates.	Biological—self-healing and energy-efficient.
Makes decisions based solely on programmed rules and learned data.	Makes decisions influenced by ethics, morals, values, and judgment.
Can be biased depending on training data.	Can be biased due to personal experiences and beliefs.

Where Do We See AI? (Applications)

Healthcare

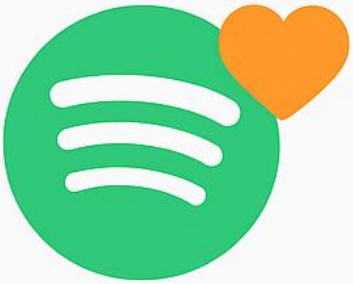
Helping doctors diagnose diseases from X-rays.

Finance

Detecting credit card fraud.

Transportation

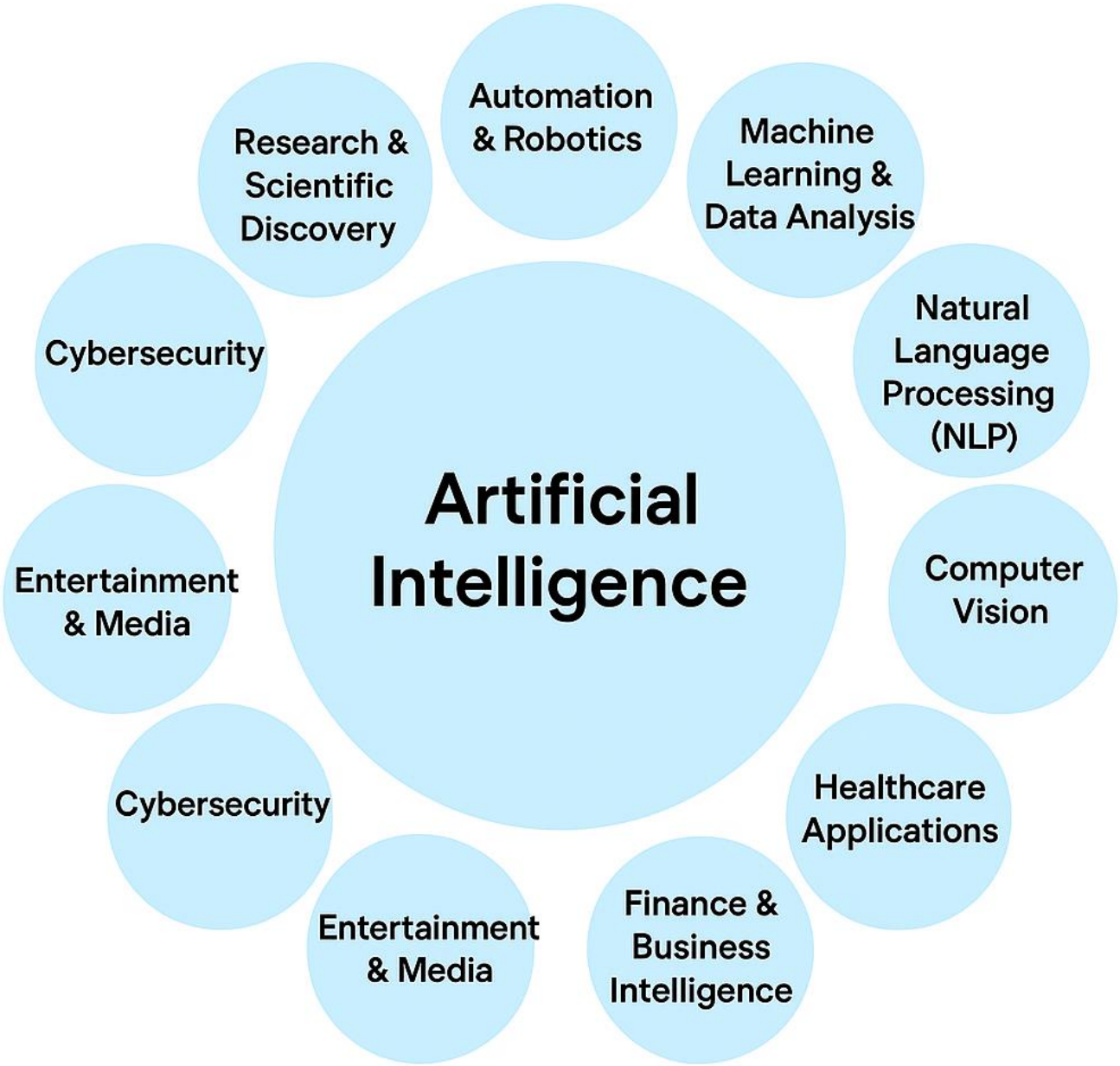
Powering the navigation in Google Maps.



Entertainment

Recommending your next favorite song on Spotify.

The Scope of AI



LAB: Setting Up Our Tools

Let's Get Our Hands Dirty!

Tools:

1. **Python: The programming language we will use (it's like English but for computers!).**
2. **Jupyter Notebook: An interactive "lab notebook" for writing and testing code step-by-step.**
3. **Libraries: Pre-built toolkits so we don't have to reinvent the wheel.**

installation steps

Step 1: Installing Python & Jupyter

Instructions:

1. **Go to python.org and download Python (version 3.x recommended).**
2. **IMPORTANT: Check the box that says "Add Python to PATH" during installation.**
3. **Open your terminal (command prompt).**
4. **Type this command and press Enter to install Jupyter.**

```
pip install jupyter
```

5. **To launch Jupyter, type:**

```
jupyter notebook
```

Step 2 :Installing AI libraries:

- **NumPy: For doing math with lists of numbers (arrays).**
- **Pandas: For organizing data into tables (like Excel in Python).**
- **Scikit-learn: The main toolbox for simple Machine Learning algorithms.**

```
pip install numpy panadas scikit-learn
```

Our Very First Code!

Let's open a new Jupyter Notebook and write a simple program to make sure everything works.

```
# Import the numpy toolbox and give it a short nickname 'np'
import numpy as np

# Create a simple list of numbers
my_list = [1, 2, 3, 4, 5]

# Convert it into a numpy array (makes math easy!)
my_array = np.array(my_list)

# Print the average of the numbers in the array
print("The average is:", np.mean(my_array))

# Print each number multiplied by 2
print("Multiplied by 2:", my_array * 2)
```

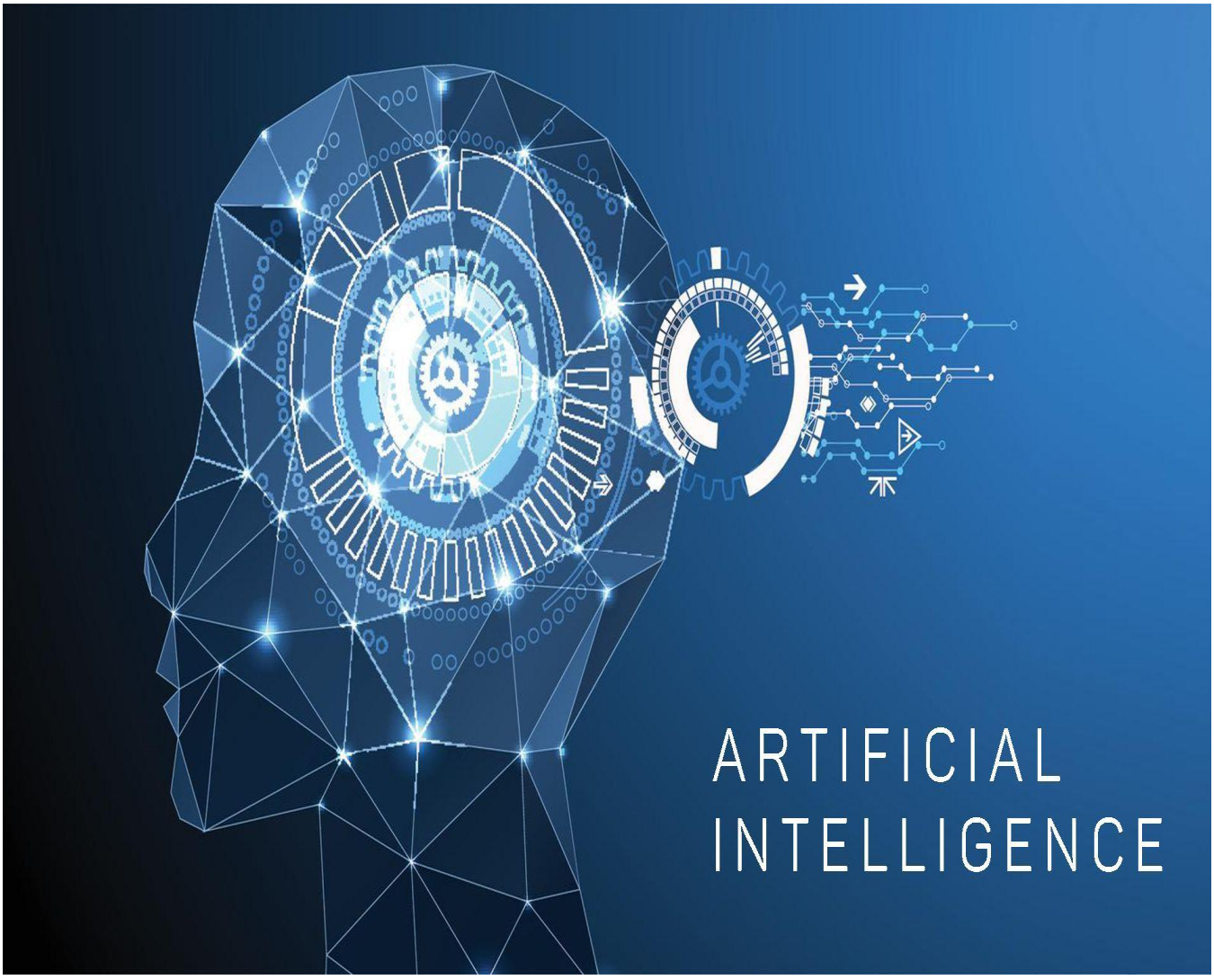
Expected output:

```
The average is: 3.0
Multiplied by 2: [ 2  4  6  8 10 ]
```

Summary

Recap:

- **AI is about making computers perform intelligent tasks.**
- **It's different from human intelligence (faster data vs. common sense).**
- **We have successfully installed Python, Jupyter, and our core AI libraries!**



HOW TO USE THIS MANUAL

Each section is a self-contained lab covering one or more algorithms. You will find: (1) Core concepts and definitions, (2) Algorithm pseudocode with explanation, (3) Complete Python implementation, (4) A worked example, and (5) Analysis of complexity and properties. Complete the exercises at the end of each section to solidify your understanding.

Foundational Concepts of Search

Before exploring specific algorithms, it is essential to understand the vocabulary and data structures that all search algorithms share. This section defines the key concepts referenced throughout this manual.

1. The Search Problem

A search problem is formally defined by five components:

Initial State	The starting configuration from which the agent begins searching. Example: the starting city in a route-finding problem.
Actions / Operators	The set of valid moves or transitions available in any given state. ACTIONS(s) returns all applicable actions in state s.
Transition Model	RESULT(s, a) — describes what state is produced when action a is applied to state s. Together with the initial state and actions, it defines the state space.
Goal Test	A function GOAL-TEST(s) that returns true if state s is a goal. The goal may be a single specific state or a set of states satisfying some condition.
Path Cost Function	A function that assigns a numeric cost to each path. Typically written g(n) for a node n — the sum of all step costs along the path from the initial state to n.

2. The Search Tree & State Space

The state space is the abstract graph of all reachable states and the actions connecting them. A search tree is the tree generated by the search algorithm as it explores this space.

Key Data Structures

Search Node (n): A data structure with four fields:

- n.STATE — the state this node represents
- n.PARENT — the node that generated this one
- n.ACTION — the action applied to reach this node
- n.PATH-COST — total cost g(n) from root to this node

Frontier	The set of all leaf nodes available for expansion — nodes generated but not yet expanded. Also called the "open list". Implemented as a queue (FIFO, LIFO, or priority queue depending on the algorithm).
Explored Set	The set of all states that have already been expanded. Also called the "closed list". Prevents re-expanding the same state. Implemented as a hash table for O(1) lookup.
Expanding a Node	Applying all legal actions to a node's state, generating a set of child nodes. These children are added to the frontier.
Branching Factor (b)	The maximum number of successors (children) any single node can have. A key parameter in complexity analysis.
Depth (d)	The depth of the shallowest goal node — i.e., the minimum number of steps required to reach a goal from the initial state.
Path Length (m)	The maximum length of any path in the state space. May be infinite for unbounded spaces.

3. Performance Evaluation Criteria

All search algorithms are evaluated on four criteria:

The Four Criteria

Completeness: Is the algorithm guaranteed to find a solution when one exists?

Optimality: Does the algorithm find the lowest-cost (optimal) solution?

Time Complexity: How many nodes are generated/expanded in the worst case?

Space Complexity: What is the maximum number of nodes stored in memory simultaneously?

4. Tree-Search vs. Graph-Search

Two fundamental variants of search exist. Tree-Search does not track visited states — it may revisit the same state multiple times via different paths, possibly looping forever. Graph-Search maintains an explored set to avoid revisiting states, making it complete in finite state spaces at the cost of extra memory.

```
function TREE-SEARCH(problem):
    frontier ← {initial state}
    loop:
        if frontier is empty → return FAILURE
        node ← choose a node from frontier
        if GOAL-TEST(node.state) → return SOLUTION(node)
        expand node → add children to frontier

function GRAPH-SEARCH(problem):
    frontier ← {initial state}
    explored ← {}   # the closed list / explored set
    loop:
        if frontier is empty → return FAILURE
        node ← choose a node from frontier
        if GOAL-TEST(node.state) → return SOLUTION(node)
        explored.add(node.state)
        expand node → add children NOT in frontier or explored
```

5. Uninformed vs. Informed Search

This manual covers two major categories of search strategies:

Uninformed Search (Blind Search)
The algorithm has NO information about the goal beyond what the problem definition provides.
It cannot estimate how "close" any state is to the goal.
Strategies: BFS, DFS, DLS, UCS, Bidirectional Search

Informed Search (Heuristic Search)
The algorithm uses a heuristic function h(n) that ESTIMATES the cost from node n to the goal.
h(n) = estimated cost of cheapest path from state at n to a goal state.
h(n) = 0 for any goal node (by definition).
Strategies: Greedy Best-First Search, A* Search

Heuristic Function h(n)	A problem-specific function that estimates the cost to reach the goal from node n. Example: straight-line distance (SLD) to the goal city in a route-finding problem. A heuristic is admissible if it never overestimates the true cost — i.e., $h(n) \leq h^*(n)$ for all n.

UNINFORMED SEARCH STRATEGIES

This section covers several search strategies that come under the heading of **uninformed search** (also called **blind search**). The term means that the strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state. All search strategies are distinguished by the *order* in which nodes are expanded. Strategies that know whether one non-goal state is "more promising" than another are called **informed search** or **heuristic search** strategies

SECTION 2 | Breadth-First Search (BFS)

Overview

Breadth-First Search (BFS) is the most fundamental uninformed search strategy. It explores all nodes at depth d before any node at depth d+1. Imagine a wave front expanding outward from the start state, layer by layer.

> **Core Idea**
>
> BFS uses a FIFO (First-In, First-Out) queue as the frontier.
>
> New nodes (children) are always deeper than their parents → they go to the BACK of the queue.
>
> The shallowest nodes are always at the FRONT → expanded first.
>
> Goal test is applied when a node is GENERATED (not when selected for expansion) — this finds the goal faster.

Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breadth-first search is an instance of the general graph-search algorithm in which the *shallowest* unexpanded node is chosen for expansion. This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is *generated* rather than when it is selected for expansion. This decision is explained below, where we discuss time complexity. Note also that the algorithm, following the general template for graph search, discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found. Thus, breadth-first search always has the shallowest path to every node on the frontier

Algorithm Pseudocode

```
function BFS(problem):
    node ← make_node(problem.initial_state, path_cost=0)
    if GOAL-TEST(node.state): return SOLUTION(node)
    frontier ← FIFO_queue containing [node]
    explored ← empty set
    loop:
        if EMPTY?(frontier): return FAILURE
        node ← POP(frontier)       # removes from FRONT
        add node.state to explored
        for each action in ACTIONS(node.state):
            child ← CHILD-NODE(problem, node, action)
            if child.state not in explored and not in frontier:
                if GOAL-TEST(child.state): return SOLUTION(child)
                INSERT(child, frontier)  # adds to BACK of queue
    return FAILURE
```

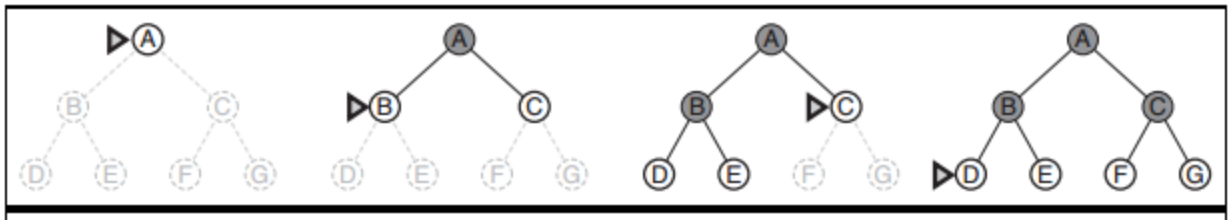


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Python Implementation

```
def BFS(graph,start_node):
    path = []
    queue.append(start_node)
    visited.add(start_node)
    print(f"start BFS from node:{start_node}")
    print('-'*20)
    while queue:
        current_node = queue.popleft()
        path.append(current_node)
        print(f"visiting {current_node}")
        # lets get adjacent nodes 'neighbors'
        neighbors = graph.get(current_node)
        print(f"neighbors are: {neighbors}")
        for neighbor in neighbors:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
                print(f"{neighbor} added to queue")

        print(f"queue status: {queue}")
        print(f"visited status: {visited}\n")

    return
```

Worked Example

Consider a binary tree with levels: root A at depth 0; B and C at depth 1; D, E, F, G at depth 2. Goal = G.

Step-by-Step Trace

Start: frontier = [A], explored = {}

Expand A → generate B, C. frontier = [B, C], explored = {A}

Expand B → generate D, E. frontier = [C, D, E], explored = {A, B}

Expand C → generate F, G. G is goal! Return solution: [A → C → G]

Complexity Analysis

Algorithm	Complete?	Optimal?	Time	Space
BFS	Yes (if b finite)	Yes (unit costs)	O(b^d)	O(b^d)

Key insight: BFS stores ALL generated nodes. At depth d=10 with branching factor b=10, memory required is approximately 10 terabytes. This makes BFS impractical for deep searches despite its optimality guarantees.

SECTION 3 | DFS, Depth-Limited Search & Bidirectional Search

Depth-First Search (DFS)

DFS always expands the deepest node in the frontier first. Instead of a FIFO queue, it uses a LIFO stack — the most recently added node is the first to be expanded, driving the search as deep as possible before backtracking.

Core Idea

Uses a LIFO stack (or recursive calls) as the frontier.

Explores one branch completely before trying another.

Main advantage: O(b·m) space complexity — stores only ONE path from root to leaf.

Main risk: Can follow an infinite path and never return (not complete in tree-search form).

Depth-first search always expands the *deepest* node in the current frontier of the search tree. The progress of the search is illustrated in Figure 3.16. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors.

The depth-first search algorithm is an instance of the graph-search algorithm, whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue.

A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected

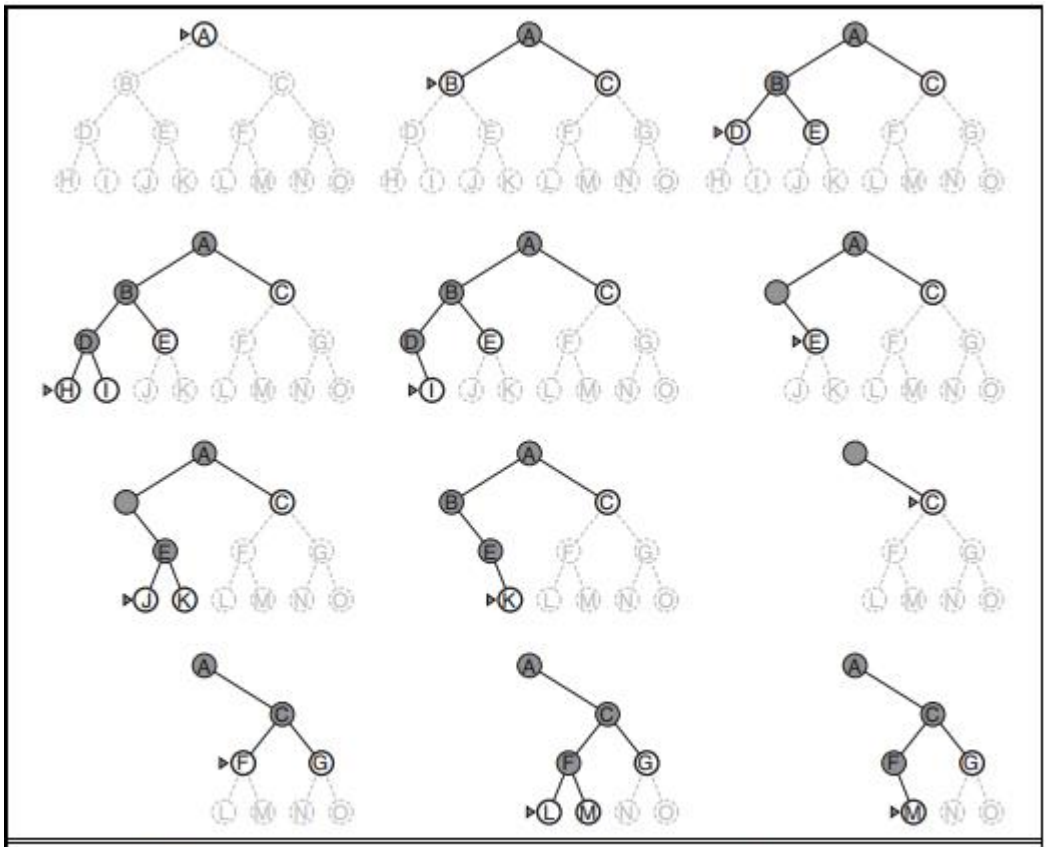


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

DFS Pseudocode

```
function DFS(problem):
    frontier ← LIFO_stack containing [make_node(problem.initial_state)]
    explored ← empty set
    loop:
        if EMPTY?(frontier): return FAILURE
        node ← POP(frontier)           # removes NEWEST node
        if GOAL-TEST(node.state): return SOLUTION(node)
        if node.state not in explored:
            explored.add(node.state)
            for each action in ACTIONS(node.state):
                child ← CHILD-NODE(problem, node, action)
                INSERT(child, frontier)  # push onto stack
    return FAILURE
```

Python Implementation

```
def dfs(problem):
    """Depth-First Search (Graph Search version)."""
    root = Node(problem.initial)
    frontier = [root]          # LIFO stack (use .append and .pop)
    explored = set()

    while frontier:
        node = frontier.pop()  # pop from END = LIFO behavior
        if problem.goal_test(node.state):
            return node.solution()
        if node.state not in explored:
            explored.add(node.state)
            # Push children — last action explored first
            for action, child_state, cost in problem.successors(node.state):
                child = Node(child_state, node, action, node.path_cost + cost)
                if child.state not in explored:
                    frontier.append(child)
    return None

# Recursive DFS variant (very common in practice):
def dfs_recursive(problem):
    def dls(node, explored):
        if problem.goal_test(node.state): return node.solution()
        explored.add(node.state)
        for action, child_state, cost in problem.successors(node.state):
            if child_state not in explored:
                child = Node(child_state, node, action, node.path_cost + cost)
                result = dls(child, explored)
                if result is not None: return result
        return None
    return dls(Node(problem.initial), set())
```

Depth-Limited Search (DLS)

DLS fixes the primary weakness of DFS in infinite spaces by imposing a depth cutoff ℓ. Any node at depth ℓ is treated as having no successors, preventing infinite descent.

> **Key Points**
> Solves the infinite-path problem of DFS.
> Introduces a new problem: if ℓ < d (shallowest goal depth), DLS returns FAILURE even when a solution exists.
> Returns two possible failure types: FAILURE (no solution exists) and CUTOFF (no solution within limit ℓ).
> DFS is a special case of DLS with ℓ = ∞.

Iterative Deepening Search (IDS)

IDS repeatedly runs DLS with an increasing depth limit (0, 1, 2, ...) until a solution is found. At first glance this seems wasteful, but it is actually the preferred uninformed search when depth is unknown. The key insight is that most nodes are at the deepest level — re-generating upper levels has negligible cost.

> **Why IDS Beats Both BFS and DFS**
> vs. BFS: IDS uses only O(b·d) space (linear!) instead of O(b^d). For d=16, b=10: 156 KB vs. 10 exabytes.
> vs. DFS: IDS is complete (BFS-like) and optimal for unit step costs. DFS is neither.
> Overhead: The top level (depth 1) nodes are generated d times. But since most nodes are at depth d, the total overhead factor is only b/(b-1) ≈ 1.11 for b=10. Negligible.

DLS Pseudocode

```
function DLS(problem, limit):
    return RECURSIVE-DLS(MAKE-NODE(problem.initial_state), problem, limit)

function RECURSIVE-DLS(node, problem, limit):
    if GOAL-TEST(node.state): return SOLUTION(node)
    elif limit == 0: return CUTOFF
    else:
        cutoff_occurred ← false
        for each action in ACTIONS(node.state):
            child ← CHILD-NODE(problem, node, action)
            result ← RECURSIVE-DLS(child, problem, limit - 1)
            if result == CUTOFF: cutoff_occurred ← true
            elif result != FAILURE: return result  # solution found
        if cutoff_occurred: return CUTOFF
        else: return FAILURE
```

Python Implementation (DLS + IDS)

```
FAILURE = "FAILURE"
CUTOFF  = "CUTOFF"

def dls(problem, limit):
    """Depth-Limited Search."""
    def recursive_dls(node, limit):
        if problem.goal_test(node.state):
            return node.solution()
        if limit == 0:
            return CUTOFF
        cutoff_occurred = False
        for action, child_state, cost in problem.successors(node.state):
            child = Node(child_state, node, action, node.path_cost + cost)
            result = recursive_dls(child, limit - 1)
            if result == CUTOFF:
                cutoff_occurred = True
            elif result != FAILURE:
                return result   # found a solution
        return CUTOFF if cutoff_occurred else FAILURE

    return recursive_dls(Node(problem.initial), limit)

def ids(problem):
    """Iterative Deepening Search — combines BFS completeness with DFS space."""
    for depth in range(0, float("inf")):   # increase limit: 0, 1, 2, 3, ...
        result = dls(problem, depth)
        if result != CUTOFF:
            return result   # either FAILURE or a valid solution list
        # if CUTOFF: try again with a deeper limit
```

Bidirectional Search

Instead of searching from start to goal, bidirectional search runs two simultaneous searches — one forward from the start, one backward from the goal — hoping they meet in the middle.

Core Idea & Motivation

A single BFS from start reaches b^d nodes.

Two half-searches each reach b^(d/2) nodes.

Total: 2 × b^(d/2) << b^d (exponentially fewer!)

For d=6, b=10: BFS generates ~1,111,110 nodes; Bidirectional: ~2,220 nodes.

Implementation: Replace goal test with a check if the frontiers intersect.

```
def bidirectional_bfs(problem):
    """
    Bidirectional BFS. Requires that the problem supports
    both forward successors and backward predecessors.
    """
    # Forward search from initial state
    fwd_frontier   = {problem.initial: Node(problem.initial)}
    fwd_explored   = {}

    # Backward search from goal state
    bwd_frontier   = {problem.goal: Node(problem.goal)}
    bwd_explored   = {}

    def extract_solution(meeting_state, fwd_nodes, bwd_nodes):
        """Stitch together the two half-paths."""
        fwd_path = fwd_nodes[meeting_state].solution()  # start → meeting
        bwd_path = bwd_nodes[meeting_state].solution()  # goal  → meeting
        return fwd_path + list(reversed(bwd_path))      # reverse backward half

    while fwd_frontier or bwd_frontier:
        # Expand forward frontier one level
        new_fwd = {}
        for state, node in fwd_frontier.items():
            fwd_explored[state] = node
            for action, child_state, cost in problem.successors(state):
                if child_state not in fwd_explored and child_state not in
fwd_frontier:
                    child = Node(child_state, node, action, node.path_cost + cost)
                    new_fwd[child_state] = child
                    if child_state in bwd_frontier or child_state in bwd_explored:
                        return extract_solution(child_state,
{**fwd_explored,**new_fwd},
                                                {**bwd_explored,**bwd_frontier})
        fwd_frontier = new_fwd

        # Expand backward frontier one level (using predecessors)
        new_bwd = {}
        for state, node in bwd_frontier.items():
            bwd_explored[state] = node
            for action, pred_state, cost in problem.predecessors(state):
                if pred_state not in bwd_explored and pred_state not in
bwd_frontier:
                    pred = Node(pred_state, node, action, node.path_cost + cost)
                    new_bwd[pred_state] = pred
                    if pred_state in fwd_frontier or pred_state in fwd_explored:
                        return extract_solution(pred_state,
{**fwd_explored,**fwd_frontier},
                                                {**bwd_explored,**new_bwd})
        bwd_frontier = new_bwd
    return None
```

Important Limitations

Requires a method for computing PREDECESSORS (backward action model).

Works best when there is a single, explicit goal state.

Difficult to apply when the goal is an abstract description (e.g., "no two queens attack each other").

At least ONE frontier must be kept in memory — O(b^(d/2)) space requirement.

Section 1–2 Comparison Table

Algorithm	Complete?	Optimal?	Time	Space
BFS	Yes (b finite)	Yes (unit cost)	O(b^d)	O(b^d)
DFS	No (tree-search)	No	O(b^m)	O(b·m)
DLS (limit ℓ)	No	No	O(b^ℓ)	O(b·ℓ)
IDS	Yes (b finite)	Yes (unit cost)	O(b^d)	O(b·d)
Bidirectional	Yes (BFS dirs)	Yes (BFS dirs)	O(b^d/2)	O(b^d/2)

SECTION 4 | Uniform-Cost Search (UCS)

Overview

Uniform-Cost Search (UCS) generalizes BFS to handle non-uniform step costs. While BFS expands the shallowest node (minimizing number of steps), UCS expands the node with the lowest cumulative path cost g(n). UCS is the classical algorithm by Dijkstra applied to AI search.

> **Why UCS over BFS?**
> BFS is optimal ONLY when all step costs are equal.
> In real problems (route-finding, network routing), costs differ — BFS may return a short but expensive path.
> UCS always finds the lowest-cost path, regardless of step costs (as long as costs ≥ ε > 0).

When all step costs are equal, breadth-first search is optimal because it always expands the *shallowest* unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step-cost function. Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost* g(n). This is done by storing the frontier as a priority queue ordered by g.
In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search. The first is that the goal test is applied to a node when it is *selected for expansion* (as in the generic graph-search algorithm) rather than when it is first generated. The reason is that the first goal node that is *generated* may be on a suboptimal path. The second difference is that a test is added in case a better path is found to a node currently on the frontier.

Both of these modifications come into play in the example shown in Figure 3.15, where the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97 = 177. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99 + 211 = 310. Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+97+101 = 278. Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned

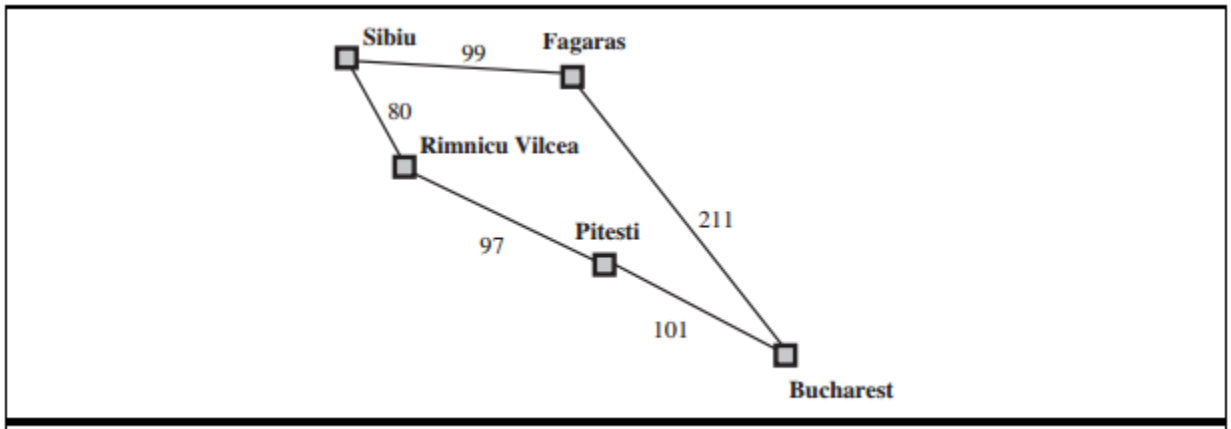


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

How It Works

UCS uses a Priority Queue (min-heap) ordered by path cost g(n) as its frontier. The node with the smallest cumulative cost is always expanded next. Two key differences from BFS:

> **UCS vs BFS: Key Differences**
>
> 1. Goal test is applied when a node is SELECTED for expansion, not when generated.
>
> Reason: The first generated goal node may not be on the optimal path.
>
> 2. If a shorter path to a frontier node is found, that node's cost is updated (priority queue update).

Algorithm Pseudocode

```
function UCS(problem):
    node ← make_node(problem.initial_state, path_cost=0)
    frontier ← priority_queue ordered by PATH-COST, with [node]
    explored ← empty set
    loop:
        if EMPTY?(frontier): return FAILURE
        node ← POP(frontier)          # node with LOWEST g(n)
        if GOAL-TEST(node.state): return SOLUTION(node)
        explored.add(node.state)
        for each action in ACTIONS(node.state):
            child ← CHILD-NODE(problem, node, action)
            if child.state not in explored and not in frontier:
                frontier ← INSERT(child, frontier)
            elif child.state in frontier with higher PATH-COST:
                replace that frontier node with child  # found a cheaper path!
```

Python Implementation

```
import heapq

def ucs(problem):
    """Uniform-Cost Search using a min-heap priority queue."""
    root = Node(problem.initial)
    # heap entries: (cost, counter, node) — counter breaks ties
    counter  = 0
    frontier = [(0, counter, root)]
    # frontier_costs: maps state → best known cost (for update check)
    frontier_costs = {root.state: 0}
    explored = set()

    while frontier:
        cost, _, node = heapq.heappop(frontier)

        # Skip if we already found a cheaper path to this state
        if node.state in explored:
            continue
        if frontier_costs.get(node.state, float("inf")) < cost:
            continue

        # Goal test at SELECTION time
        if problem.goal_test(node.state):
            return node.solution()

        explored.add(node.state)

        for action, child_state, step_cost in problem.successors(node.state):
            new_cost = cost + step_cost
            if child_state not in explored:
                if new_cost < frontier_costs.get(child_state, float("inf")):
                    counter += 1
                    child = Node(child_state, node, action, new_cost)
                    heapq.heappush(frontier, (new_cost, counter, child))
                    frontier_costs[child_state] = new_cost
    return None
```

Worked Example (Romania Map)

Find the shortest-cost route from Sibiu to Bucharest (using the Romania road map).

```
Step 1: frontier = [(0, Sibiu)]
        Pop Sibiu (cost=0)
        Generate: Rimnicu_Vilcea (cost=80), Fagaras (cost=99), Oradea (cost=151)

Step 2: Pop Rimnicu_Vilcea (cost=80) — lowest cost
        Generate: Pitesti (cost=80+97=177), Craiova (cost=80+146=226)

Step 3: Pop Fagaras (cost=99)
        Generate: Bucharest (cost=99+211=310)

Step 4: Pop Pitesti (cost=177)
        Generate: Bucharest via Pitesti (cost=177+101=278) ← CHEAPER!
        Update Bucharest in frontier: 310 → 278

Step 5: Pop Bucharest (cost=278) — GOAL!
Optimal path: Sibiu → Rimnicu_Vilcea → Pitesti → Bucharest (cost=278)
```

Complexity Analysis

Algorithm	Complete?	Optimal?	Time	Space
UCS	Yes (cost ≥ ε)	Yes (general)	O(b^(1+C*/ε))	O(b^(1+C*/ε))

Where C* is the optimal solution cost and ε is the minimum step cost. Note: UCS's complexity is NOT characterized by depth d, but by cost. It may explore many cheap steps before finding the goal.

UCS = A* with h(n) = 0

UCS is a special case of A* search where the heuristic h(n) = 0 for all nodes.

This means UCS guides search purely by path cost g(n), with no estimate of remaining cost.

A* (covered in Section 5) adds h(n) to guide the search more intelligently.

SECTION 5 | Greedy Best-First Search

Introduction to Informed Search

Informed search algorithms use a heuristic function h(n) to guide the search toward the goal more efficiently. The general framework is called Best-First Search, where nodes are selected for expansion based on an evaluation function f(n).

Best-First Search Framework

f(n) = evaluation function — determines which node to expand next.

h(n) = heuristic function — estimated cost from node n to a goal.

g(n) = actual path cost from initial state to node n.

Implementation: same as UCS but priority queue ordered by f(n) instead of g(n).

Admissible Heuristic	A heuristic h(n) is admissible if it NEVER overestimates the true cost to reach the goal: h(n) ≤ h*(n) for all nodes n. An admissible heuristic is "optimistic" by nature. Example: straight-line distance to the goal city is admissible because the actual road distance is always ≥ the straight line.

Consistent Heuristic	A heuristic is consistent (monotone) if for every node n and successor n' via action a: h(n) ≤ c(n, a, n') + h(n'). This is the triangle inequality. Every consistent heuristic is also admissible. Required for A* Graph-Search to be optimal.

Greedy Best-First Search

Greedy Best-First Search expands the node that appears to be closest to the goal, using only the heuristic h(n) as the evaluation function. It is "greedy" because at each step it makes the locally best choice — picking what looks nearest to the goal — without considering the cost already incurred.

Greedy Best-First: Key Property

f(n) = h(n) only — ignores g(n) (path cost so far).

Pros: Very fast in practice; low search cost when h is accurate.

Cons: NOT complete (can get stuck in loops); NOT optimal (may find a longer path).

Worst case: degenerates to DFS — time/space O(b^m).

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

Algorithm Pseudocode

```
function GREEDY-BEST-FIRST-SEARCH(problem, h):
    node ← make_node(problem.initial_state)
    node.f ← h(node.state)
    frontier ← priority_queue ordered by f, with [node]
    explored ← empty set
    loop:
        if EMPTY?(frontier): return FAILURE
        node ← POP(frontier)           # node with LOWEST h(n)
        if GOAL-TEST(node.state): return SOLUTION(node)
        explored.add(node.state)
        for each action in ACTIONS(node.state):
            child ← CHILD-NODE(problem, node, action)
            child.f ← h(child.state)
            if child.state not in explored and not in frontier:
                INSERT(child, frontier)
```

How Greedy Best First Search working:

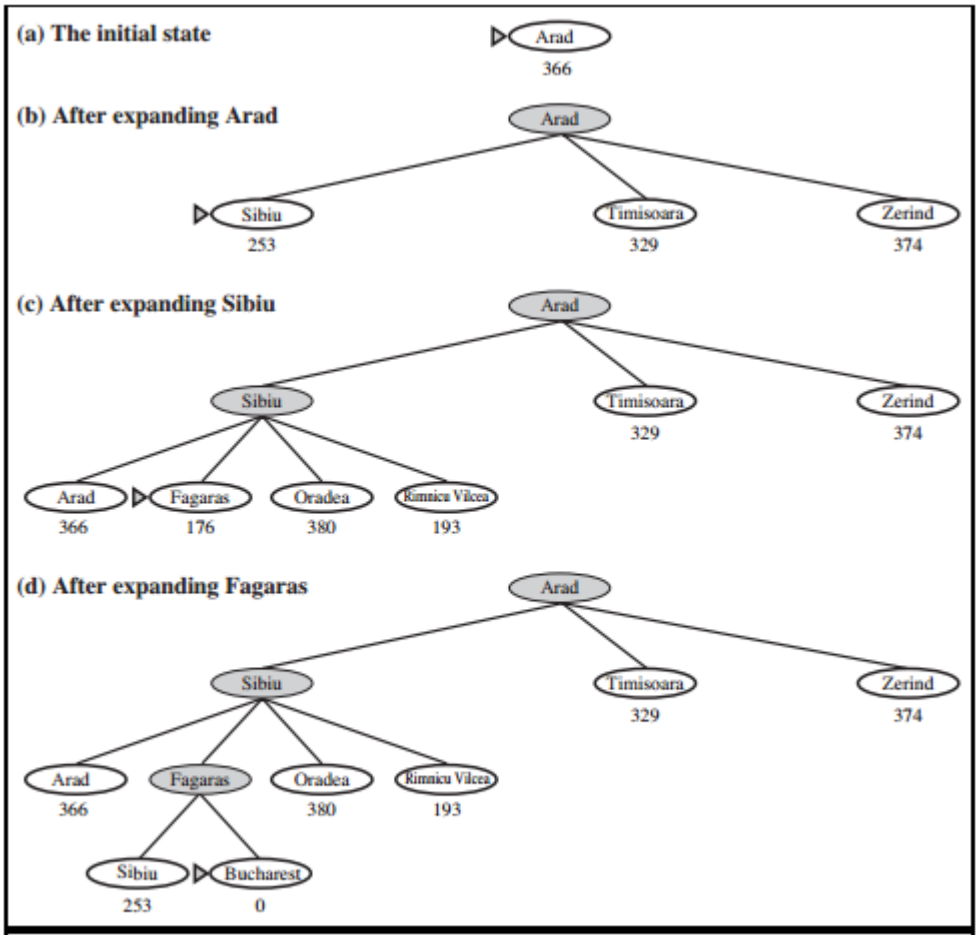


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Python Implementation

```
import heapq

def greedy_best_first(problem, h):
    """
    Greedy Best-First Search.
    h: callable h(state) -> estimated cost to goal
    """
    root = Node(problem.initial)
    counter = 0
    # Priority queue ordered by h(state)
    frontier = [(h(root.state), counter, root)]
    explored = set()
    in_frontier = {root.state}

    while frontier:
        h_val, _, node = heapq.heappop(frontier)
        in_frontier.discard(node.state)

        if problem.goal_test(node.state):
            return node.solution()

        explored.add(node.state)

        for action, child_state, step_cost in problem.successors(node.state):
            if child_state not in explored and child_state not in in_frontier:
                child = Node(child_state, node, action, node.path_cost +
step_cost)
                counter += 1
                heapq.heappush(frontier, (h(child_state), counter, child))
                in_frontier.add(child_state)
    return None

# ── Example: Romania Route Finding ──────────────────────────────
# Straight-line distances to Bucharest (the heuristic h_SLD)
h_SLD = {
    'Arad':366, 'Bucharest':0, 'Craiova':160, 'Drobeta':242,
    'Fagaras':176, 'Giurgiu':77, 'Lugoj':244, 'Mehadia':241,
    'Oradea':380, 'Pitesti':100, 'Rimnicu_Vilcea':193, 'Sibiu':253,
    'Timisoara':329, 'Urziceni':80, 'Vaslui':199, 'Zerind':374
}
# Usage: greedy_best_first(romania_problem, lambda s: h_SLD[s])
```

Worked Example (Romania Map)

Find a route from Arad to Bucharest using Greedy Best-First Search with h_SLD (straight-line distance to Bucharest).

```
frontier = [(366, Arad)]

Step 1: Pop Arad (h=366)
        Successors: Sibiu(h=253), Timisoara(h=329), Zerind(h=374)
```

```
        frontier = [(253,Sibiu), (329,Timisoara), (374,Zerind)]

Step 2: Pop Sibiu (h=253) — smallest h
        Successors: Fagaras(h=176), Rimnicu_Vilcea(h=193), ...
        frontier = [(176,Fagaras), (193,Rimnicu_Vilcea), ...]

Step 3: Pop Fagaras (h=176) — smallest h
        Successors: Bucharest(h=0)
        GOAL FOUND!

Path: Arad → Sibiu → Fagaras → Bucharest  (total road distance: 450 km)
NOTE: Optimal path is Arad → Sibiu → Rimnicu_Vilcea → Pitesti → Bucharest (418 km)
      Greedy found a SUBOPTIMAL solution!
```

Complexity Analysis

Algorithm	Complete?	Optimal?	Time	Space
Greedy Best-First	No (graph: finite)	No	O(b^m)	O(b^m)

SECTION 6 | A* Search

Overview: The Best of Both Worlds

A* (pronounced "A-star") is the most widely known and used informed search algorithm. It combines the path cost g(n) with the heuristic h(n) into a single evaluation function that guides the search optimally.

The A* Evaluation Function

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

g(n) = actual cost from start to node n (known exactly)

h(n) = estimated cost from n to goal (heuristic)

f(n) = estimated cost of the cheapest complete solution THROUGH node n

A* expands the node that minimizes the TOTAL estimated solution cost.

Why is this correct? Because f(n) = g(n) + h(n) estimates the cost of the cheapest path through n. If h is admissible (never overestimates), then f never overestimates either, so A* will always find the optimal solution.

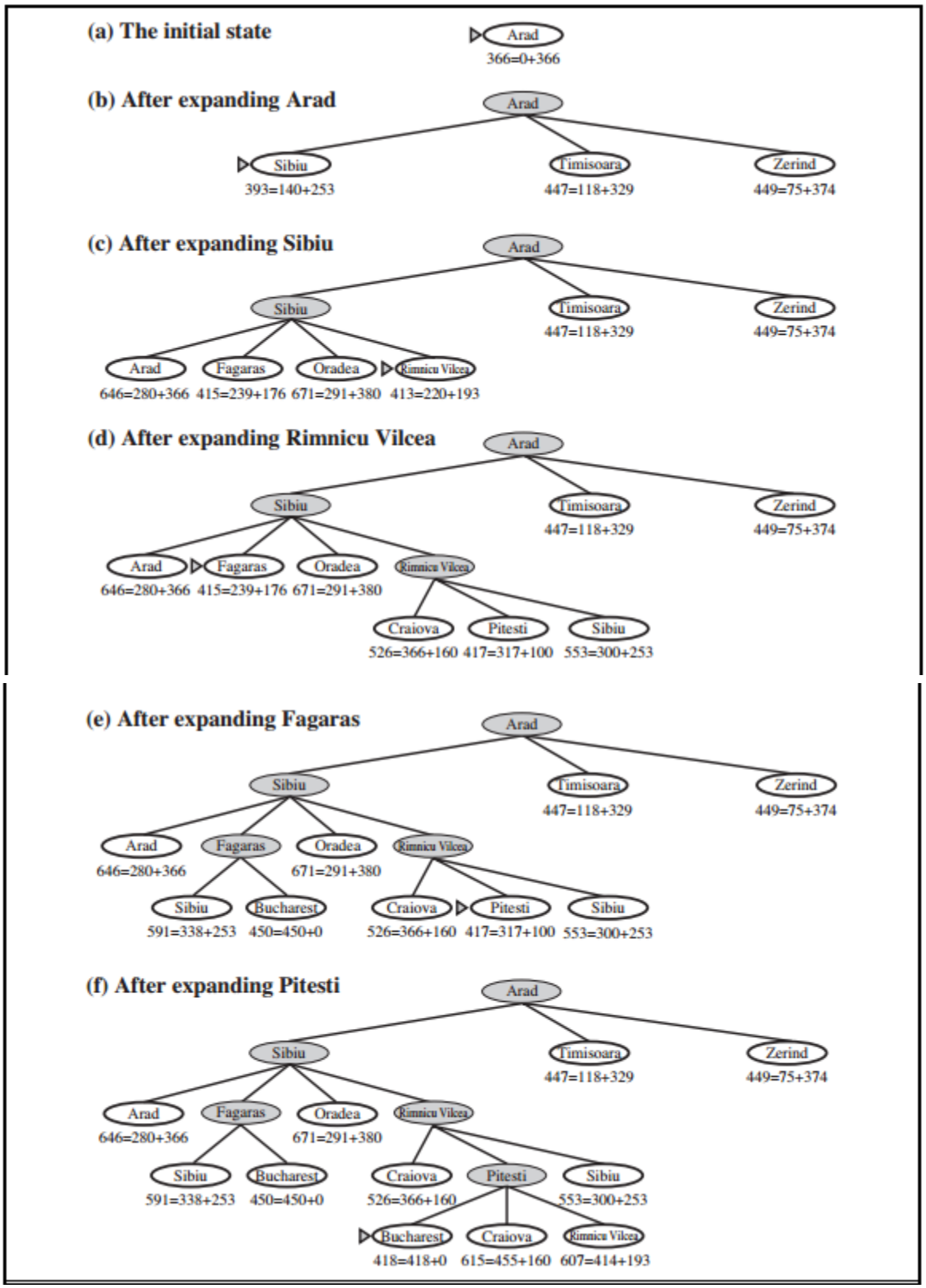
Conditions for Optimality

Admissibility (Tree Search)	For A* tree-search to be optimal, h(n) must be admissible: h(n) ≤ h*(n) for all n. This ensures f never overestimates the true optimal cost through any node.
Consistency (Graph Search)	For A* graph-search to be optimal, h(n) must be consistent: h(n) ≤ c(n, a, n') + h(n') for all successors n'. Consistency guarantees that f values are non-decreasing along any path — so the first time A* selects a node for expansion, it has found the optimal path to that node.

Algorithm Pseudocode

```
function A-STAR(problem, h):
    node ← make_node(problem.initial_state, path_cost=0)
    node.f ← h(node.state)              # f = g(0) + h(initial)
    frontier ← priority_queue ordered by f, with [node]
    explored ← empty set
    loop:
        if EMPTY?(frontier): return FAILURE
        node ← POP(frontier)             # node with LOWEST f(n)
        if GOAL-TEST(node.state): return SOLUTION(node)
        explored.add(node.state)
        for each action in ACTIONS(node.state):
            child ← CHILD-NODE(problem, node, action)
            child.f ← child.PATH-COST + h(child.state)
            if child.state not in explored and not in frontier:
                INSERT(child, frontier)
            elif child.state in frontier with higher f-cost:
                replace that node with child   # found better path
```

How A* working:



Python Implementation

```
import heapq

def Astar(problem, h):
    """
    A* Search (Graph Search version).
    h: callable h(state) -> non-negative estimated cost to goal.
    Requires h to be CONSISTENT for guaranteed optimality.
    """
    root = Node(problem.initial)
    f_root = 0 + h(root.state)   # f = g + h
    counter = 0
    frontier = [(f_root, counter, root)]
    # best_f: maps state → best known f value in frontier
    best_f   = {root.state: f_root}
    explored = set()

    while frontier:
        f_val, _, node = heapq.heappop(frontier)

        # Skip stale entries (we may have updated this node's cost)
        if node.state in explored:
            continue
        if best_f.get(node.state, float("inf")) < f_val:
            continue

        # Goal check at EXPANSION time
        if problem.goal_test(node.state):
            return node.solution()

        explored.add(node.state)

        for action, child_state, step_cost in problem.successors(node.state):
            if child_state in explored:
                continue
            g_child = node.path_cost + step_cost
            f_child = g_child + h(child_state)
            # Only add if we found a better path to this state
            if f_child < best_f.get(child_state, float("inf")):
                child = Node(child_state, node, action, g_child)
                best_f[child_state] = f_child
                counter += 1
                heapq.heappush(frontier, (f_child, counter, child))
    return None

# ─── Example Usage with Romania Map ──────────────────────────────
# (Using the same romania_problem and h_SLD)
# result = Astar(romania_problem, lambda s: h_SLD.get(s, 0))

# ─── 8-Puzzle Example Heuristics ─────────────────────────────────
def manhattan_distance(state, goal):
    """h2 for 8-puzzle: sum of Manhattan distances of each tile."""
    total = 0
    n = int(len(state) ** 0.5)   # grid size (3 for 8-puzzle)
    for tile in range(1, n*n):   # skip blank (tile 0)
        cur_pos  = state.index(tile)
```

```
        goal_pos = goal.index(tile)
        cur_r,  cur_c  = divmod(cur_pos,  n)
        goal_r, goal_c = divmod(goal_pos, n)
        total += abs(cur_r - goal_r) + abs(cur_c - goal_c)
    return total

def misplaced_tiles(state, goal):
    """h1 for 8-puzzle: count of tiles not in their goal position."""
    return sum(1 for s, g in zip(state, goal) if s != g and s != 0)
```

Worked Example (Romania Map)

Find the optimal route from Arad to Bucharest using A* with h_SLD. Each node is labeled f = g + h.

```
State           g (actual)   h (SLD)   f = g + h
──────────────────────────────────────────────────
Arad            0            366       366
Sibiu           140          253       393   ← expand (lowest f)
Timisoara       118          329       447
Zerind          75           374       449
— After expanding Sibiu:
Rimnicu_Vilcea  220          193       413   ← expand next
Fagaras         239          176       415
— After expanding Rimnicu_Vilcea:
Pitesti         317          100       417   ← expand
— After expanding Fagaras:
Bucharest via Fagaras: g=450, h=0, f=450  (generated but not expanded yet)
— After expanding Pitesti:
Bucharest via Pitesti: g=418, h=0, f=418  ← BETTER PATH! Update frontier.
— Pop Bucharest (f=418) → GOAL!

Optimal path: Arad → Sibiu → Rimnicu_Vilcea → Pitesti → Bucharest
Total cost: 418 km  (OPTIMAL)
```

Why A* is Optimal: The Key Insight

A* never expands a node with f(n) > C* (the optimal cost). Because h is consistent, f values are non-decreasing along any path. When A* first selects a node for expansion, it has found the cheapest path to that node. Since goal nodes have h = 0, the first goal node expanded must be optimal.

A* Properties Summary

Complete: Yes (finite branching, step costs ≥ ε > 0).

Optimal: Yes (with admissible h for tree-search; consistent h for graph-search).

Optimally Efficient: Yes — no other optimal algorithm expands fewer nodes for the same heuristic.

Space: O(b^d) — keeps ALL generated nodes in memory. This is its main weakness.

Time: O(b^d) worst case, but much better in practice with a good heuristic.

Section 4–5 Comparison

Algorithm	Complete?	Optimal?	Time	Space
Greedy Best-First	No (finite graphs)	No	O(b^m)	O(b^m)
A* Search	Yes	Yes (admissible h)	O(b^d)	O(b^d)
UCS (A* with h=0)	Yes	Yes	O(b^(C*/ε))	O(b^(C*/ε))

Choosing the Right Algorithm

Algorithm Selection Guide

BFS: Use when all step costs are equal and memory is not a concern. Guarantees optimal (fewest steps) solution.

DFS: Use when memory is very limited and solution depth is known/bounded. Not for optimal solutions.

IDS: Best uninformed algorithm when solution depth is unknown and memory is limited. Use instead of DFS.

UCS: Use when step costs vary and no heuristic is available. Optimal for any non-negative costs.

Bidirectional BFS: Use when the goal is explicit and predecessors are computable. Huge speedup for large spaces.

Greedy: Use when speed matters more than optimality and you have a good heuristic. Not for critical paths.

A*: The general-purpose best choice when a heuristic is available. Optimal and complete.

SECTION 6 | Local Search Algorithms (Chapter 4)

Overview and Motivation

In the search strategies covered so far, we maintained an explicit path from the initial state to the current node. The path itself was part of the solution. Local search takes an entirely different approach: it keeps only a single current state and moves to neighbouring states, discarding the path behind it. This single decision — to forget where you came from — enables two major advantages: near-constant memory use and the ability to operate in state spaces too large for systematic methods.

Local search is the right tool when (1) the path to the solution does not matter — only the final state counts — and (2) finding a good state matters more than proving optimality. Classic applications include the n-queens problem, job-shop scheduling, the travelling salesman problem, and hardware circuit design.

The State-Space Landscape (Figure 4.1)

The conceptual foundation for local search is the state-space landscape. Imagine a physical surface where every point is a state, the horizontal position encodes the state's identity, and the vertical elevation encodes the value of an objective function (or, equivalently, minus the heuristic cost). The search agent's task is to find the highest peak (global maximum) on this surface.

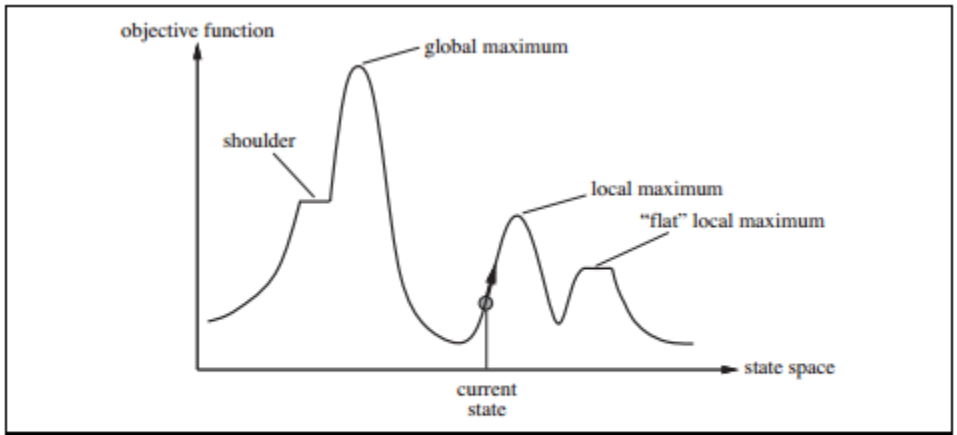


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Key vocabulary for the landscape:

Global maximum: the best state in the entire space — the goal of optimisation.

Local maximum: better than all immediate neighbours but not globally best.

Plateau / flat local max: a region where all neighbours have equal value.

Shoulder: a plateau from which an uphill path continues; progress is possible.

Ridge: a sequence of local maxima connected by narrow paths — hard for greedy methods.

Hill-Climbing Search (Steepest Ascent)

Hill climbing is the simplest local search. At every step it moves to the highest-valued neighbour of the current state. Because it neither looks ahead nor maintains a search tree, it requires O(1) memory. Russell & Norvig describe it as "trying to find the top of Mount Everest in a thick fog while suffering from amnesia."

Pseudocode (Figure 4.2)

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    current ← MAKE-NODE(problem.INITIAL-STATE)
    loop do
        neighbour ← a highest-valued successor of current
        if neighbour.VALUE ≤ current.VALUE then
            return current.STATE        -- no improvement: we are at a peak
        current ← neighbour
```

Python Implementation

```
def hill_climbing(problem):
    """
    Steepest-ascent hill climbing.
    problem must provide:
      - problem.initial: initial state
      - problem.value(state): objective to MAXIMISE
      - problem.neighbours(state): list of successor states
    """
    current = problem.initial
    while True:
        neighbours = problem.neighbours(current)
        if not neighbours:
            return current          # no successors: stuck
        best_neighbour = max(neighbours, key=problem.value)
        if problem.value(best_neighbour) <= problem.value(current):
            return current          # no improvement
        current = best_neighbour

# ── 8-Queens example ──────────────────────────────────────────
import random

class EightQueens:
    """8-Queens as a local search problem."""
    def __init__(self):
        # random initial placement: one queen per column
        self.initial = tuple(random.randint(0, 7) for _ in range(8))

    def value(self, state):
        """Negative number of attacking pairs (maximise → fewest attacks)."""
        attacks = 0
        for i in range(8):
            for j in range(i+1, 8):
                if state[i] == state[j]:          # same row
                    attacks += 1
                if abs(state[i]-state[j]) == j-i: # same diagonal
                    attacks += 1
        return -attacks   # negate: we MAXIMISE, so fewer attacks = higher value
```

```
    def neighbours(self, state):
        """Move any queen to any other row in its column: 8×7 = 56 successors."""
        result = []
        for col in range(8):
            for row in range(8):
                if row != state[col]:
                    new = list(state)
                    new[col] = row
                    result.append(tuple(new))
        return result

problem = EightQueens()
solution = hill_climbing(problem)
print("Queens (row per column):", solution)
print("Attacks:", -problem.value(solution))
```

Variants

Steepest-Ascent: always picks the single best neighbour (described above).

Stochastic Hill Climbing: picks randomly from uphill moves (probability weighted by steepness).

First-Choice: generates neighbours randomly until one is better than current. Good when branching factor is huge (e.g., thousands of successors).

Random-Restart: runs hill climbing repeatedly from randomly generated starts. Trivially complete with probability → 1. For 8-queens: success prob ≈ 0.14 per run → expects ~7 runs → ~25 steps total.

Sideways Moves: allows moves to equal-valued neighbours (plateau traversal). Raises success from 14% to 94% for 8-queens, at the cost of capping consecutive sideways moves.

Simulated Annealing

Simulated annealing (SA) escapes local maxima by occasionally accepting worse moves, with a probability that decreases over time. The name comes from the metallurgical process of slowly cooling heated material to remove defects. If the cooling schedule is slow enough, SA finds a global optimum with probability approaching 1.

Key Mechanics

At each step: pick a random successor, compute ΔE = VALUE(next) − VALUE(current).

If ΔE > 0 (improvement): always accept the move.

If ΔE < 0 (worsening): accept with probability e^(ΔE / T).

T = temperature, decreases per schedule: T → 0 means no bad moves accepted.

High T early: exploratory (accepts many bad moves). Low T late: exploitative (like hill climbing).

Pseudocode

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current ← MAKE-NODE(problem.INITIAL-STATE)
    for t = 1 to ∞ do
        T ← schedule(t)          -- get temperature for step t
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← next.VALUE – current.VALUE
        if ΔE > 0 then current ← next
        else current ← next  only with probability e^(ΔE/T)
```

Python Implementation

```
import math, random

def simulated_annealing(problem, schedule):
    """
    schedule: callable(t) → temperature T
    Returns when T reaches 0 or no change possible.
    """
    current = problem.initial
    for t in range(1, 100_000):
        T = schedule(t)
        if T == 0:
            return current
        neighbours = problem.neighbours(current)
        if not neighbours:
            return current
        next_state = random.choice(neighbours)
        delta_E = problem.value(next_state) - problem.value(current)
        if delta_E > 0:
            current = next_state           # always accept improvements
        else:
            prob = math.exp(delta_E / T)   # accept bad moves with probability
            if random.random() < prob:
                current = next_state
    return current

# ── Geometric cooling schedule ──────────────────────────────
def geometric_schedule(T0=100.0, alpha=0.995):
    """T(t) = T0 * alpha^t — geometric decay."""
    def schedule(t):
        T = T0 * (alpha ** t)
        return T if T > 0.001 else 0   # cut off near zero
    return schedule

# Usage:
# solution = simulated_annealing(EightQueens(), geometric_schedule())
```

Local Beam Search

Local beam search keeps k states simultaneously rather than just one. It begins with k randomly generated states and at each step selects the k best successors from all successors of all k current states. Unlike k

independent random restarts, useful information passes between the k threads: better states attract resources away from poor ones.

> Key difference from random restarts: the k searches communicate. Better-performing states "recruit" search effort from worse ones.
>
> Risk: the k states can quickly cluster in one small region — degenerating to expensive hill climbing.
>
> Stochastic beam search: chooses k successors randomly (probability proportional to value) to preserve diversity.

Genetic Algorithms

Genetic algorithms (GAs) extend stochastic beam search by combining pairs of states (sexual reproduction) rather than modifying a single state. They maintain a population of k individuals, each represented as a string (chromosome). New individuals are produced by selection, crossover, and mutation.

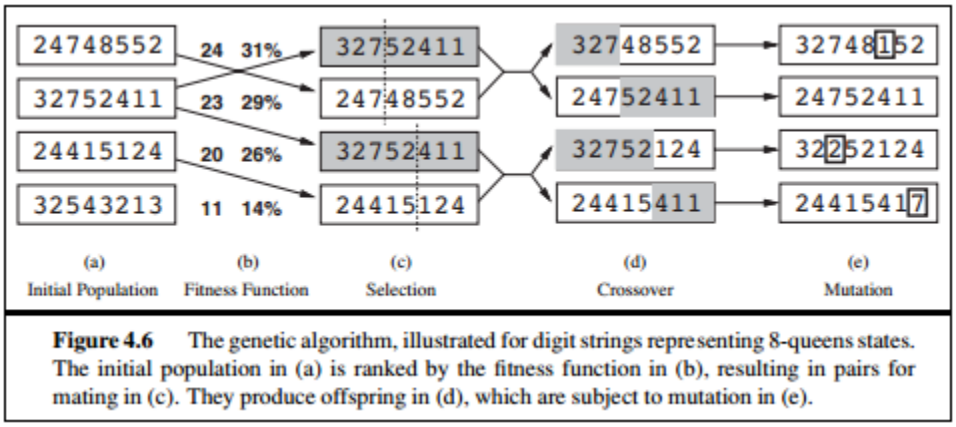


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
      new_population ← empty set
      for i = 1 to SIZE(population) do
          x ← RANDOM-SELECTION(population, FITNESS-FN)
          y ← RANDOM-SELECTION(population, FITNESS-FN)
          child ← REPRODUCE(x, y)
          if (small random probability) then child ← MUTATE(child)
          add child to new_population
      population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Performance Summary

Algorithm	Complete?	Memory	Best Use Case
Hill Climbing	No (local max)	O(1)	Fast, simple, dense solution spaces
Random Restart HC	Yes (prob→1)	O(1)	When p(success) per run is reasonable
Simulated Annealing	Yes (slow sched)	O(1)	Hard landscapes, NP-hard optimisation
Local Beam (k)	No	O(k)	Parallel search with communication
Genetic Algorithm	No	O(k)	Problems with combinatorial structure

SECTION 7 | Adversarial Search — Game Playing (Chapter 5)

Overview: Why Games Are Different

In single-agent search the environment is cooperative: it simply waits for the agent to find a path. In adversarial search — games — a second agent (the opponent) actively works against us, choosing moves that minimise our payoff. This transforms search into a joint optimisation problem: we choose the move that maximises our utility assuming the opponent minimises it.

Russell & Norvig focus on deterministic, fully observable, two-player, zero-sum games of perfect information — the classical setting for combinatorial game theory. Examples: chess, checkers, Othello, Go, tic-tac-toe.

> **Formal Game Definition — a search problem with these components:**
> S_0: Initial state — the starting board configuration.
> PLAYER(s): whose turn it is in state s.
> ACTIONS(s): legal moves available in state s.
> RESULT(s, a): transition model — the state produced by move a in state s.
> TERMINAL-TEST(s): true when the game is over (win/loss/draw).
> UTILITY(s, p): numerical payoff to player p in terminal state s.

The Game Tree and Minimax Values

A game tree is the full expansion of all play sequences. Because chess has ~10^154 nodes in its game tree, we cannot search it completely. Regardless, the minimax value is the key concept: it is the utility MAX can guarantee by playing optimally, assuming MIN also plays optimally.

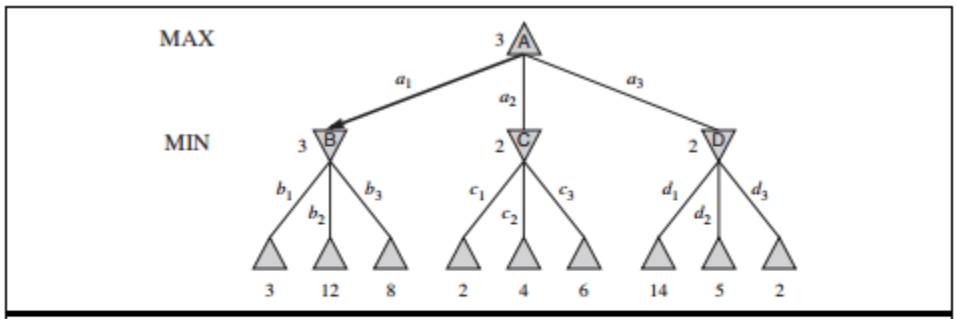


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Minimax Value	MINIMAX(s) = UTILITY(s) if terminal; max_a MINIMAX(RESULT(s,a)) if MAX's turn; min_a MINIMAX(RESULT(s,a)) if MIN's turn.

Ply	One half-move by one player. A "depth of 2" means one full move by each player (2 plies).

The Minimax Algorithm (Figure 5.3)

Pseudocode

```
function MINIMAX-DECISION(state) returns an action
    return argmax_{a ∈ ACTIONS(state)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← -∞
    for each a in ACTIONS(state):
        v ← MAX(v, MIN-VALUE(RESULT(state, a)))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for each a in ACTIONS(state):
        v ← MIN(v, MAX-VALUE(RESULT(state, a)))
    return v
```

Python Implementation

```
def minimax_decision(state, game):
    """Return the optimal action for MAX."""
    best_action = max(
        game.actions(state),
        key=lambda a: min_value(game.result(state, a), game)
    )
    return best_action

def max_value(state, game):
    if game.terminal_test(state):
        return game.utility(state, player="MAX")
    return max(min_value(game.result(state, a), game)
               for a in game.actions(state))

def min_value(state, game):
    if game.terminal_test(state):
        return game.utility(state, player="MAX")  # always from MAX's view
    return min(max_value(game.result(state, a), game)
               for a in game.actions(state))

# ── Tic-tac-toe example ──────────────────────────────────────────
class TicTacToe:
    def __init__(self):
        self.initial = ('.'*9, 'X')  # (board string, current player)
```

```
    def actions(self, state):
        board, _ = state
        return [i for i, c in enumerate(board) if c == '.']

    def result(self, state, action):
        board, player = state
        new_board = board[:action] + player + board[action+1:]
        return (new_board, 'O' if player == 'X' else 'X')

    def terminal_test(self, state):
        board, _ = state
        wins = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]
        for p in ['X', 'O']:
            if any(all(board[i]==p for i in w) for w in wins): return True
        return '.' not in board

    def utility(self, state, player="MAX"):
        board, _ = state
        wins = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]
        for p, val in [('X',1),('O',-1)]:
            if any(all(board[i]==p for i in w) for w in wins): return val
        return 0
```

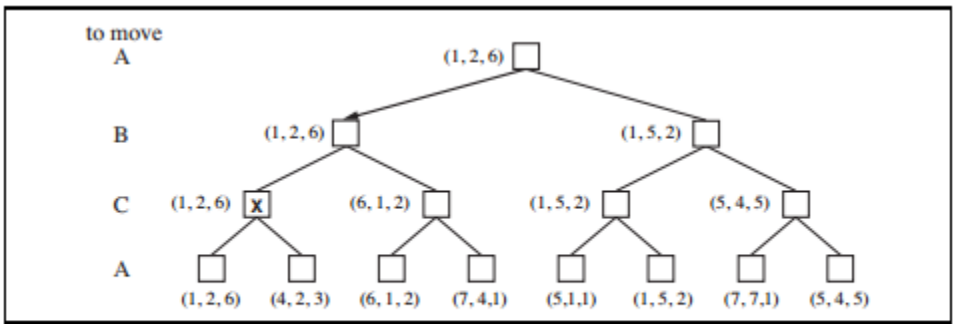


Figure 5.4 The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Alpha–Beta Pruning

Minimax is exponential in tree depth. Alpha–beta pruning achieves the same result as full minimax but eliminates branches that cannot possibly influence the root decision, cutting the effective branching factor from b to approximately √b. With perfect move ordering, time reduces from O(b^m) to O(b^(m/2)) — effectively doubling the searchable depth.

α (alpha): best (highest) value MAX can guarantee so far along the current path.
β (beta): best (lowest) value MIN can guarantee so far along the current path.
Prune at MAX node if v ≥ β: MIN already has a better option elsewhere; will never reach here.
Prune at MIN node if v ≤ α: MAX already has a better option elsewhere; will never reach here.
The pruned nodes are invisible to the opponent and do not change the optimal move.

Pseudocode

```
function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state, α=-∞, β=+∞)
    return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state): return UTILITY(state)
    v ← -∞
    for each a in ACTIONS(state):
        v ← MAX(v, MIN-VALUE(RESULT(state,a), α, β))
        if v ≥ β: return v          -- β prune: MIN will not allow this
        α ← MAX(α, v)
    return v

function MIN-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state): return UTILITY(state)
    v ← +∞
    for each a in ACTIONS(state):
        v ← MIN(v, MAX-VALUE(RESULT(state,a), α, β))
        if v ≤ α: return v          -- α prune: MAX will not allow this
        β ← MIN(β, v)
    return v
```

Python Implementation

```
def alpha_beta_search(state, game):
    """Return the optimal action using alpha-beta pruning."""
    def max_value(state, alpha, beta):
        if game.terminal_test(state):
            return game.utility(state)
        v = float('-inf')
        for a in game.actions(state):
            v = max(v, min_value(game.result(state, a), alpha, beta))
            if v >= beta:
                return v      # β-prune
            alpha = max(alpha, v)
        return v

    def min_value(state, alpha, beta):
        if game.terminal_test(state):
            return game.utility(state)
        v = float('inf')
        for a in game.actions(state):
            v = min(v, max_value(game.result(state, a), alpha, beta))
            if v <= alpha:
                return v      # α-prune
            beta = min(beta, v)
        return v

    return max(
        game.actions(state),
        key=lambda a: min_value(game.result(state, a), float('-inf'),
float('inf'))
    )
```

Alpha–Beta Trace on Figure 5.2 Example

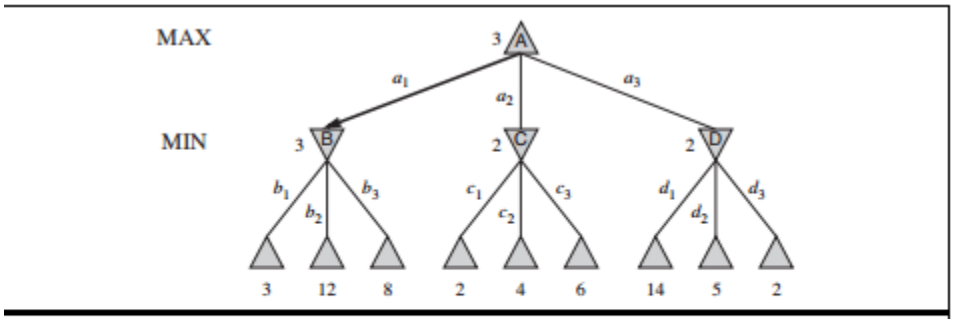


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Heuristic Minimax — Cutting Off Search

For real games like chess, searching to terminal states is impossible. Claude Shannon (1950) proposed cutting off the search at depth d and applying a heuristic evaluation function EVAL(s) to non-terminal leaf nodes, converting them into effective terminal nodes.

> Replace TERMINAL-TEST with CUTOFF-TEST(state, depth) — returns true at depth d or true terminal.
>
> Replace UTILITY with EVAL(state) — a heuristic estimate of the position's value.
>
> In chess: EVAL uses material balance (pawn=1, bishop/knight=3, rook=5, queen=9) plus positional bonuses.
>
> Quiescence search: extend search at "noisy" positions (captures/checks) beyond depth d to avoid horizon effects.
>
> Transposition table: a hash table caching evaluated positions to avoid re-evaluating transpositions.

Complexity Comparison

Algorithm	Time	Pruning	Optimal?	Notes
Minimax	O(b^m)	None	Yes	Impractical for real games
Alpha–Beta	O(b^m/2)	α and β cuts	Yes	Same result, ~√b effective branching
H-Minimax	O(b^d)	α–β + depth	No	Practical; quality depends on EVAL
Expectiminimax	O(b^m · n^m)	Limited	Yes	For stochastic games (chance nodes)

SECTION 8 | Constraint Satisfaction Problems — Concepts & Inference (Chapter 6)

What Is a CSP?

A constraint satisfaction problem (CSP) represents the state as a set of variables, each with a domain of possible values, and a set of constraints that restrict which combinations of values are allowed. Rather than treating states as atomic "black boxes," the CSP framework exposes internal structure that algorithms can exploit to prune enormous swathes of the search space.

Formal definition — a CSP is a triple (X, D, C):

$X = \{X_1, X_2, \ldots, X_n\}$: a set of n variables.

$D = \{D_1, D_2, \ldots, D_n\}$: domains — D_i is the set of allowable values for X_i.

$C = \{C_1, C_2, \ldots, C_m\}$: constraints — each $C_i = \langle \text{scope, relation} \rangle$ specifies which assignments are legal.

Solution: a complete, consistent assignment — every variable assigned a value that satisfies all constraints.

Consistent Assignment	An assignment that violates no constraint. Also called "legal."
Complete Assignment	An assignment in which every variable has been assigned a value.
Partial Assignment	An assignment covering only some variables — the state during search.

Example 1 — Map Colouring (Australia)

Colour each Australian state/territory with one of {red, green, blue} so that no two adjacent regions share a colour. This classic example from the book illustrates all key CSP concepts.

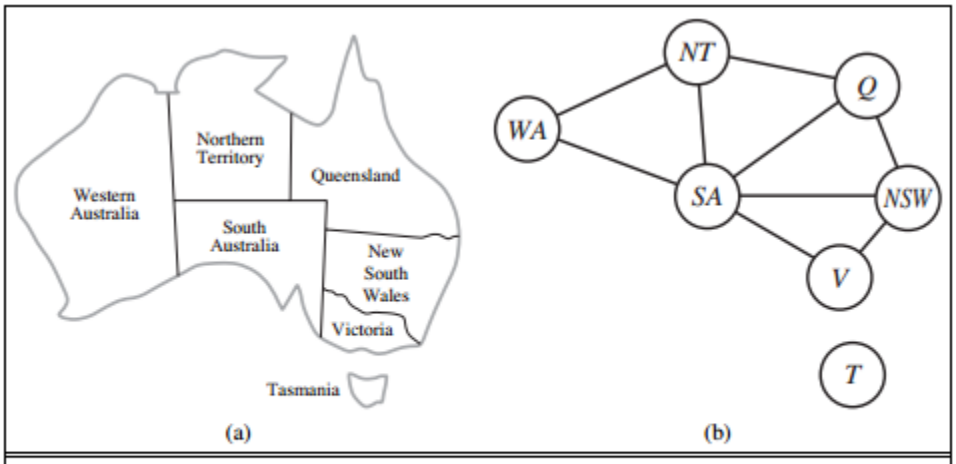


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Example 2 — Sudoku as a CSP

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

Types of Constraints

Type	Arity	Example	Representation
Unary	1 var	SA ≠ green	Restrict domain directly
Binary	2 vars	SA ≠ WA	Constraint graph edge
Ternary	3 vars	Between(X, Y, Z)	Hyperedge
Global	n vars	Alldiff($X_1 \ldots X_n$)	Hypergraph; special algorithms

Constraint Propagation and Local Consistency

The central idea in CSP is constraint propagation: use constraints to eliminate values from domains, which may trigger further eliminations, and so on. This is called enforcing local consistency. Unlike search, constraint propagation is a form of inference — it derives facts rather than guessing.

Node Consistency	Variable X_i is node-consistent if all values in D_i satisfy X_i's unary constraints. Eliminates trivially illegal values.
Arc Consistency	X_i is arc-consistent w.r.t. X_j if for every value in D_i, there exists some value in D_j satisfying the constraint between them. The most widely used form.
Path Consistency	A pair $\{X_i, X_j\}$ is path-consistent w.r.t. X_m if every consistent assignment to $\{X_i, X_j\}$ has some extension to X_m. Stronger than arc consistency.
k-Consistency	For any k−1 assigned variables, any consistent kth variable can be assigned. Arc consistency = 2-consistency; path consistency = 3-consistency (binary CSPs).

The AC-3 Algorithm (Figure 6.3)

AC-3 is the standard arc-consistency algorithm, used as a preprocessing step and interleaved with backtracking search. It maintains a queue of arcs (X_i, X_j) to check. When X_i's domain is reduced, all arcs pointing to X_i are re-added to the queue. Worst-case time: $O(cd^3)$ where c = number of arcs, d = max domain size.

Pseudocode

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  inputs: *csp*, a binary CSP with components (X, D, C)
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to *queue*
  return *true*

---

function REVISE(*csp*, X_i, X_j) **returns** true iff we revise the domain of X_i
  revised $\leftarrow$ *false*
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      revised $\leftarrow$ *true*
  return *revised*

Figure 6.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

Python Implementation

```
from collections import deque

def ac3(csp):
    """
    csp: dict with keys:
      'variables': list of variable names
      'domains':   dict var → set of values
      'constraints': dict (Xi, Xj) → callable(xi_val, xj_val) → bool
    Returns True if arc-consistent solution possible; False if contradiction.
    Modifies csp['domains'] in place.
    """
    domains = csp['domains']
    constraints = csp['constraints']

    # Build neighbour map for efficiency
    neighbours = {v: set() for v in csp['variables']}
    for (xi, xj) in constraints:
        neighbours[xi].add(xj)
        neighbours[xj].add(xi)

    queue = deque(constraints.keys())   # start with all arcs

    while queue:
        xi, xj = queue.popleft()
        if revise(domains, constraints, xi, xj):
            if len(domains[xi]) == 0:
                return False           # domain empty: contradiction
            for xk in neighbours[xi] - {xj}:
                queue.append((xk, xi))
    return True
```

```
def revise(domains, constraints, xi, xj):
    """Remove values from domains[xi] that have no support in domains[xj]."""
    revised = False
    constraint = constraints.get((xi, xj)) or constraints.get((xj, xi))
    if constraint is None:
        return False
    to_remove = []
    for x in domains[xi]:
        if not any(constraint(x, y) for y in domains[xj])  \
               and not any(constraint(y, x) for y in domains[xj]):
            to_remove.append(x)
    for x in to_remove:
        domains[xi].discard(x)
        revised = True
    return revised

# ── Australia Map Colouring Example ──────────────────────────────────
colors = {'red', 'green', 'blue'}
australia_csp = {
    'variables': ['WA','NT','Q','NSW','V','SA','T'],
    'domains': {v: set(colors) for v in ['WA','NT','Q','NSW','V','SA','T']},
    'constraints': {
        ('SA','WA'): lambda a,b: a!=b, ('SA','NT'): lambda a,b: a!=b,
        ('SA','Q'): lambda a,b: a!=b,  ('SA','NSW'): lambda a,b: a!=b,
        ('SA','V'): lambda a,b: a!=b,  ('WA','NT'): lambda a,b: a!=b,
        ('NT','Q'): lambda a,b: a!=b,  ('Q','NSW'): lambda a,b: a!=b,
        ('NSW','V'): lambda a,b: a!=b,
    }
}
result = ac3(australia_csp)
print("Arc-consistent:", result)
print("Domains after AC-3:", australia_csp['domains'])
```

AC-3 Trace on Australia (Partial)

```
  Initial domains: all variables have {red, green, blue}

  Suppose we manually assign WA = red, then apply AC-3:
  WA domain → {red}

  Arc (NT, WA): for each value in NT, check if WA has a compatible value.
    NT=red: WA=red violates SA≠WA? Here WA≠NT constraint:
            red conflicts with red → remove "red" from NT.
    NT=green: WA=red OK → keep.
    NT=blue:  WA=red OK → keep.
    NT domain → {green, blue}

  Arc (SA, WA): SA=red conflicts with WA=red → remove red from SA.
    SA domain → {green, blue}

  This cascades: other arcs involving NT and SA are re-queued.
  Eventual result: domains shrink, eliminating much of the search space.
```

SECTION 9 | Backtracking Search for CSPs (Chapter 6, Sections 6.3–6.5)

Backtracking Search Overview

When constraint propagation cannot solve a CSP alone, we must search. The backtracking search algorithm assigns variables one at a time, checking constraints as it goes, and backtracks when no legal value remains for the current variable. Unlike naive depth-first search, it exploits CSP commutativity: the order of variable assignments does not change the solution, so we assign one variable per search level.

> Key insight — commutativity: {WA=red, NT=green} = {NT=green, WA=red}. We fix the variable order, reducing $n! \cdot d^n$ leaves to d^n.
>
> Backtracking = depth-first search with early constraint checking (detect failure as soon as a constraint is violated).
>
> Three enhancement dimensions: variable ordering, value ordering, inference.

Pseudocode

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or *k*-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

The term **backtracking search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in Figure 6.5. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value. Part of the search tree for the Australia problem is shown in Figure 6.6, where we have assigned variables in the order WA, NT, Q, ... Because the representation of CSPs is standardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test. Notice that BACKTRACKING-SEARCH keeps only a single representation of a state and alters that representation rather than creating new ones, as described on page 87. In Chapter 3 we improved the poor performance of uninformed search algorithms by supplying them with domain-specific heuristic functions derived from our knowledge of the problem. It turns out that we can solve CSPs efficiently *without* such domain-specific knowledge. Instead, we can add some sophistication to the unspecified functions in Figure 6.5, using them to address the following questions:

1. Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?
2. What inferences should be performed at each step in the search (INFERENCE)?
3. When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

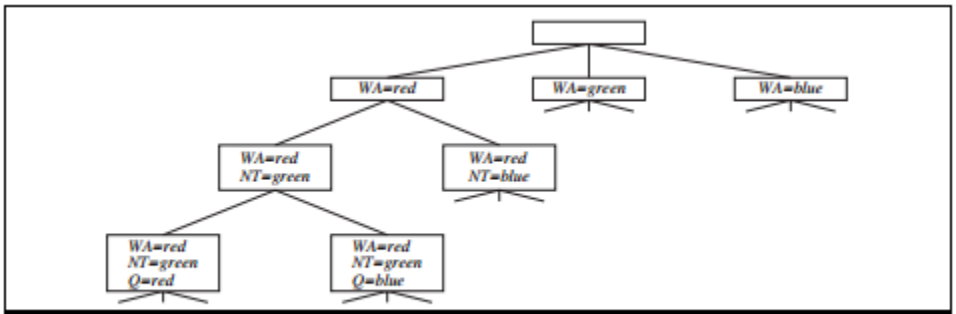


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

Full Python Implementation with MRV, LCV, and Forward Checking

```
from collections import defaultdict

class CSP:
    def __init__(self, variables, domains, constraints):
        self.variables   = variables
        self.domains     = {v: list(d) for v, d in domains.items()}
        self.constraints = constraints  # dict: (vi,vj) → callable(a,b)→bool
        # Build adjacency for efficiency
        self.neighbours = defaultdict(set)
        for (vi, vj) in constraints:
            self.neighbours[vi].add(vj)
            self.neighbours[vj].add(vi)

    def consistent(self, var, value, assignment):
        """Check var=value against all currently assigned neighbours."""
        for nbr in self.neighbours[var]:
            if nbr not in assignment:
                continue
            arc = (var, nbr)
            con = self.constraints.get(arc) or self.constraints.get((nbr, var))
            if con:
                if not con(value, assignment[nbr]):
                    return False
        return True

# ── Variable ordering ── MRV (Minimum Remaining Values) ──────────────────
def select_unassigned_variable(csp, assignment, domains):
    """MRV: choose variable with fewest legal values (fail-first)."""
    unassigned = [v for v in csp.variables if v not in assignment]
    return min(unassigned, key=lambda v: len(domains[v]))

# ── Value ordering ── LCV (Least Constraining Value) ─────────────────────
def order_domain_values(var, assignment, csp, domains):
    """LCV: prefer values that rule out fewest choices for neighbours."""
    def count_conflicts(value):
        total = 0
        for nbr in csp.neighbours[var]:
            if nbr not in assignment:
                for nbr_val in domains[nbr]:
                    arc = (var, nbr)
                    con = csp.constraints.get(arc) or
csp.constraints.get((nbr,var))
                    if con and not con(value, nbr_val):
                        total += 1
        return total
    return sorted(domains[var], key=count_conflicts)

# ── Inference ── Forward Checking ────────────────────────────────────────
def forward_checking(csp, var, value, assignment, domains):
    """
    After assigning var=value, delete any values from unassigned
    neighbours that are now inconsistent. Returns False if any
    domain becomes empty (contradiction detected early).
    """
    pruned = []   # remember what we removed for backtracking
    for nbr in csp.neighbours[var]:
        if nbr not in assignment:
```

```
            arc = (var, nbr)
            con = csp.constraints.get(arc) or csp.constraints.get((nbr, var))
            if con:
                for nbr_val in list(domains[nbr]):
                    if not con(value, nbr_val):
                        domains[nbr].remove(nbr_val)
                        pruned.append((nbr, nbr_val))
                if not domains[nbr]:
                    return None, pruned   # contradiction!
    return pruned, pruned

# ── Main backtracking ──────────────────────────────────────────────
def backtracking_search(csp):
    domains = {v: list(d) for v, d in csp.domains.items()}  # working copy

    def backtrack(assignment):
        if len(assignment) == len(csp.variables):
            return assignment
        var = select_unassigned_variable(csp, assignment, domains)
        for value in order_domain_values(var, assignment, csp, domains)  \
                     if domains[var] else []:
            if csp.consistent(var, value, assignment):
                assignment[var] = value
                old_domains = {v: list(d) for v, d in domains.items()}
                pruned, ok = forward_checking(csp, var, value, assignment,
domains)
                if ok is not None:
                    result = backtrack(assignment)
                    if result is not None:
                        return result
                # Undo: restore domains
                for v, val in (pruned or []):
                    domains[v].append(val)
                del assignment[var]
                domains.update(old_domains)
        return None

    return backtrack({})

# ── Test on Australia ──────────────────────────────────────────────
colors = ['red', 'green', 'blue']
problem = CSP(
    variables=['WA','NT','Q','NSW','V','SA','T'],
    domains={v: colors for v in ['WA','NT','Q','NSW','V','SA','T']},
    constraints={
        ('SA','WA'): lambda a,b: a!=b, ('SA','NT'): lambda a,b: a!=b,
        ('SA','Q'): lambda a,b: a!=b,  ('SA','NSW'): lambda a,b: a!=b,
        ('SA','V'): lambda a,b: a!=b,  ('WA','NT'): lambda a,b: a!=b,
        ('NT','Q'): lambda a,b: a!=b,  ('Q','NSW'): lambda a,b: a!=b,
        ('NSW','V'): lambda a,b: a!=b,
    }
)
sol = backtracking_search(problem)
print("Solution:", sol)
```

Heuristics for Variable and Value Ordering

Variable Ordering — MRV (Most Constrained Variable)

The Minimum Remaining Values (MRV) heuristic — also called fail-first — selects the variable with the fewest legal values remaining in its domain. If a variable has only one value left, assigning it costs nothing and may propagate useful information. If it has zero values, we detect failure immediately.

Variable Ordering — Degree Heuristic

As a tiebreaker when several variables have the same domain size, the degree heuristic picks the variable involved in the most constraints on other unassigned variables. In Australia, SA has degree 5 — the highest — and choosing it first propagates the most information.

Value Ordering — Least Constraining Value (LCV)

Once a variable is chosen, the Least Constraining Value heuristic prefers the value that rules out the fewest choices for neighbouring variables. The intuition: we want to leave the most flexibility for future assignments. Unlike MRV (which is fail-first for variables), LCV is "succeed-first" for values: try the value most likely to lead to a solution.

Inference During Search

Forward Checking (FC)

After assigning $X_i = v$, forward checking immediately removes from the domains of each unassigned neighbour X_j any value inconsistent with v. If any neighbour's domain becomes empty, backtrack immediately. This detects imminent failures one step ahead, avoiding useless work.

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	(R)	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	(R)	B	(G)	R B	R G B	B	R G B
After *V=blue*	(R)	B	(G)	R	(B)		R G B

Figure 6.7 The progress of a map-coloring search with forward checking. *WA* = *red* is assigned first; then forward checking deletes *red* from the domains of the neighboring variables *NT* and *SA*. After *Q* = *green* is assigned, *green* is deleted from the domains of *NT*, *SA*, and *NSW*. After *V* = *blue* is assigned, *blue* is deleted from the domains of *NSW* and *SA*, leaving *SA* with no legal values.

Maintaining Arc Consistency (MAC)

MAC is stronger than forward checking. After assigning $X_i = v$, it calls AC-3 starting from the arcs (X_j, X_i) for all unassigned neighbours X_j, then propagates arc consistency recursively. MAC detects inconsistencies further ahead than FC, often preventing more backtracking.

Intelligent Backtracking — Conflict-Directed Backjumping

Standard backtracking upon failure backs up one step (chronological backtracking). This is often wasteful: the failure may be caused by an assignment made several levels earlier, with no causal relationship to the immediately preceding step.

> Conflict set for X_i: the set of previously assigned variables that caused one of X_i's values to be eliminated.
>
> Backjumping: when X_i has no legal values, jump back to the most recent variable in conf(X_i).
>
> Conflict-directed backjumping (CBJ): deeper form — when X_j also has no legal values, its conflict set is absorbed upward: conf(X_i) ← conf(X_i) ∪ conf(X_j) − {X_i}.
>
> Constraint learning / no-goods: record minimal subsets of conflicting assignments so the same failure is never repeated across different parts of the search tree.

Local Search for CSPs — Min-Conflicts (Figure 6.8)

An alternative to backtracking is to start with a complete (but inconsistent) assignment and iteratively repair it. The min-conflicts heuristic selects a conflicted variable and assigns it the value that minimises the total number of constraint violations. Remarkably efficient for many large CSPs.

Pseudocode

function MIN-CONFLICTS(csp, max_steps) **returns** a solution or failure
  inputs: csp, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  $current \leftarrow$ an initial complete assignment for csp
  for i = 1 to max_steps **do**
    if $current$ is a solution for csp **then return** $current$
    $var \leftarrow$ a randomly chosen conflicted variable from csp.VARIABLES
    $value \leftarrow$ the value v for var that minimizes CONFLICTS($var, v, current, csp$)
    set $var = value$ in $current$
  return *failure*

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

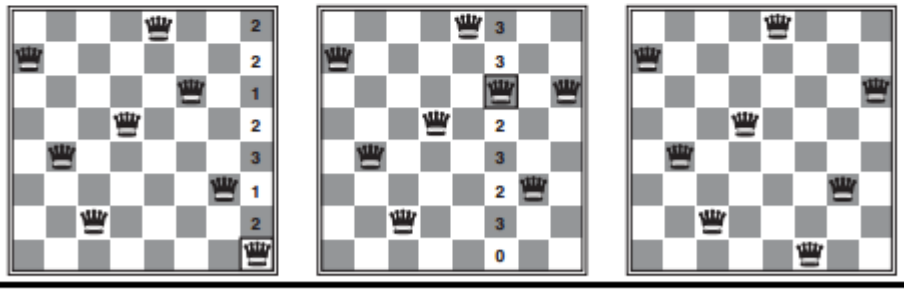


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

Python Implementation

```
import random

def min_conflicts(csp, max_steps=10000):
    """
    Local search for CSPs using the min-conflicts heuristic.
    Returns a solution assignment or None.
    """
    # Start with a random complete assignment
    assignment = {v: random.choice(list(csp.domains[v])) for v in csp.variables}

    for _ in range(max_steps):
        if is_solution(csp, assignment):
            return assignment
        # Pick a random conflicted variable
        conflicted = [v for v in csp.variables
                      if count_conflicts(csp, v, assignment[v], assignment) > 0]
        if not conflicted:
            return assignment
        var = random.choice(conflicted)
        # Assign the value minimising conflicts
        assignment[var] = min(
            csp.domains[var],
            key=lambda val: count_conflicts(csp, var, val, assignment)
        )
    return None

def count_conflicts(csp, var, value, assignment):
    """Count how many constraints var=value violates."""
    conflicts = 0
    for nbr in csp.neighbours[var]:
        if nbr in assignment:
            arc = (var, nbr)
            con = csp.constraints.get(arc) or csp.constraints.get((nbr, var))
            if con and not con(value, assignment[nbr]):
                conflicts += 1
    return conflicts

def is_solution(csp, assignment):
    if len(assignment) < len(csp.variables): return False
    for (vi, vj), con in csp.constraints.items():
        if vi in assignment and vj in assignment:
            if not con(assignment[vi], assignment[vj]): return False
    return True

# Min-conflicts solves the million-queens problem in ~50 steps on average.
# Compare with backtracking which cannot scale beyond ~100 queens.
```

Structure of CSPs — Tree Decomposition

The structure of the constraint graph has a profound effect on complexity. A key result: if the constraint graph is a tree, the CSP can be solved in O(nd²) time — polynomial! This is achieved by directed arc consistency from root to leaves.

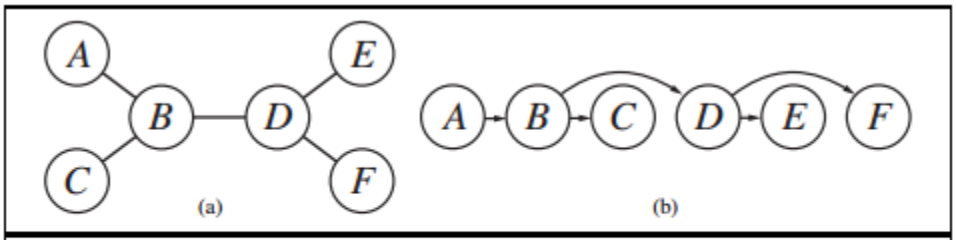


Figure 6.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root. This is known as a **topological sort** of the variables.

function TREE-CSP-SOLVER(*csp*) **returns** a solution, or failure
  inputs: *csp*, a CSP with components X, D, C

  $n \leftarrow$ number of variables in X
  assignment $\leftarrow$ an empty assignment
  root $\leftarrow$ any variable in X
  $X \leftarrow$ TOPOLOGICALSORT(X, *root*)
  for $j = n$ **down to** 2 **do**
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent **then return** *failure*
  for $i = 1$ **to** n **do**
    assignment$[X_i] \leftarrow$ any consistent value from D_i
    if there is no consistent value **then return** *failure*
  return *assignment*

Figure 6.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

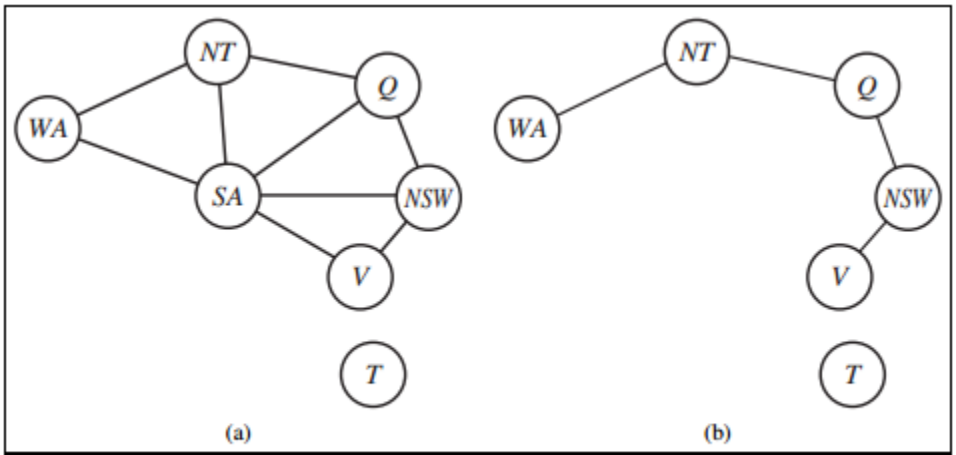


Figure 6.12 (a) The original constraint graph from Figure 6.1. (b) The constraint graph after the removal of SA.

For non-tree graphs, two approaches reduce them to trees: (1) Cutset conditioning — find a cycle cutset S of size c, enumerate all consistent assignments to S (dc), and solve the remaining tree for each. Total: O(dc · (n−c)d²). (2) Tree decomposition — decompose the constraint graph into tree-shaped subproblems, solve each independently, and combine. Time O(n · d^(w+1)) where w = tree width.

Summary — Complete CSP Algorithm Comparison

Algorithm	Type	Key Feature	Best For
Backtracking (basic)	Search	Constraint checking	Small CSPs
FC + MRV + LCV	Search	Forward checking + ordering	Medium CSPs
MAC (AC-3 interleaved)	Search+Inf	Full arc consistency during search	Tight constraint graphs
Min-Conflicts	Local	Random repair heuristic	Large CSPs, online repair
Tree CSP Solver	Special	$O(nd^2)$ — polynomial	Tree-structured graphs