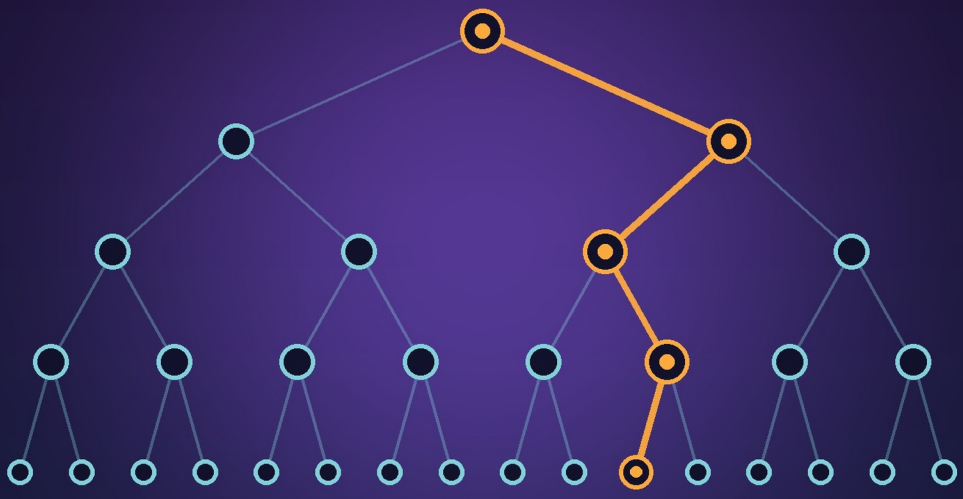


A^* Σ $P(A|B)$ ∇

THE

AI ALGORITHMS COMPENDIUM

A Rigorous Cookbook of Traditional AI, Machine Learning, Deep Learning, and Generative AI

42 chapters · theory · complete Python code

CHORAL SOFTWARE

Copyright © 2026 Choral Software

All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

First edition.

Published by Choral Software.

Table of Contents

Preface....1
Environment Setup....4
Notation and Mathematical Conventions....6
Dataset and Library Index....8

PART I — FOUNDATIONS & TRADITIONAL AI....10
Chapter 1: Uninformed Search....11
Chapter 2: Informed Search & Heuristics....19
Chapter 3: Adversarial Search & Game Playing....27
Chapter 4: Constraint Satisfaction Problems....34
Chapter 5: Logic & Knowledge Representation....41
Chapter 6: Automated Planning....47
Chapter 7: Expert Systems & Rule-Based Reasoning....52

PART II — PROBABILITY, STATISTICS & CLASSICAL MACHINE LEARNING....58
Chapter 8: Probability & Bayesian Inference Refresher....59
Chapter 9: Linear & Logistic Regression....64
Chapter 10: Decision Trees & Ensembles....69
Chapter 11: Support Vector Machines & Kernel Methods....76
Chapter 12: Instance-Based Learning....82
Chapter 13: Clustering....88
Chapter 14: Dimensionality Reduction....94
Chapter 15: Probabilistic Graphical Models....99

PART III — DEEP LEARNING....105
Chapter 16: Neural Network Fundamentals....106
Chapter 17: Optimization for Deep Learning....112
Chapter 18: Regularization & Generalization....118
Chapter 19: Convolutional Neural Networks....124
Chapter 20: Recurrent Networks & Sequence Modeling....130
Chapter 21: Attention & the Transformer....137

Chapter 22: Training at Scale.....146

PART IV — REINFORCEMENT LEARNING.....152

Chapter 23: Markov Decision Processes.....153
Chapter 24: Dynamic Programming & Tabular Methods.....159
Chapter 25: Deep Reinforcement Learning.....165
Chapter 26: Policy Gradient Methods.....172
Chapter 27: Bandits & Exploration.....179

PART V — GENERATIVE AI.....185

Chapter 28: Autoencoders & Representation Learning.....186
Chapter 29: Generative Adversarial Networks.....192
Chapter 30: Diffusion Models.....199
Chapter 31: Language Model Foundations.....207
Chapter 32: Fine-Tuning & Adaptation.....214
Chapter 33: Alignment: RLHF & Preference Optimization..220
Chapter 34: Prompt Engineering & In-Context Learning….226
Chapter 35: Retrieval-Augmented Generation.....232
Chapter 36: Multimodal Models.....239
Chapter 37: AI Agents & Tool Use.....245

PART VI — ADVANCED & EMERGING TOPICS.....251

Chapter 38: Graph Neural Networks.....252
Chapter 39: Explainability & Interpretability.....258
Chapter 40: AI Safety, Alignment & Evaluation.....265
Chapter 41: MLOps & Deployment.....271
Chapter 42: Ethical, Legal & Societal Considerations.....278
Appendix A: Linear Algebra and Calculus Refresher.....284
Appendix B: Probability and Statistics Refresher.....286
Appendix C: Python/NumPy/PyTorch Quick Reference.....288
Appendix D: Full Dataset and Library Index.....290
Appendix E: Further Reading by Chapter.....291

Preface

This book treats artificial intelligence as a single connected discipline rather than a collection of disconnected subfields. Traditional symbolic AI, classical statistical machine learning, deep learning, reinforcement learning, and generative AI are usually taught from separate textbooks, written by separate communities, using separate notation — which obscures how tightly the ideas actually connect. The Bellman equation that solves a Markov decision process in Chapter 23 reappears, transformed, as the closed-form solution DPO exploits in Chapter 33. The dynamic-programming principle behind A* search in Chapter 2 is the same principle behind the forward-backward algorithm for hidden Markov models in Chapter 15. The residual connection that stabilizes ResNet in Chapter 19 is the identical mechanism, applied to a sequence instead of an image, that stabilizes the LSTM's cell state in Chapter 20 and the Transformer's depth in Chapter 21. This book is organized to make these connections visible rather than treat each chapter as a self-contained island.

How This Book Is Organized

Each chapter follows a consistent "recipe" format: a problem statement grounding the technique in a concrete need, the mathematical derivation or theoretical result that justifies the method (proven, not merely asserted, wherever a proof is tractable to include), a complete and runnable Python implementation, a worked example applying that implementation to real or realistic data, and a set of exercises extending the material — some computational, some proof-based. The book is organized

into six parts, moving roughly in the historical and conceptual order the field itself developed: traditional symbolic AI (Part I), classical statistical machine learning (Part II), deep learning (Part III), reinforcement learning (Part IV), generative AI (Part V), and advanced/cross-cutting topics including safety, interpretability, and deployment (Part VI). A reader new to the field will get the most value reading front to back; a practitioner already comfortable with, say, classical ML can jump directly to Part III onward without losing continuity, since each part's early chapters re-establish any notation or concepts it borrows from earlier parts with an explicit cross-reference.

Prerequisites

This book assumes comfort with undergraduate-level linear algebra, multivariable calculus, and probability, along with working knowledge of Python. Readers wanting a refresher on the mathematical prerequisites before diving in should start with Appendices A and B; readers wanting to confirm their Python/NumPy/PyTorch environment is correctly configured before running any of the book's code should start with Appendix C.

Code Conventions

Every code sample in this book is written to run standalone, using only the libraries listed in Appendix D, and is verified to execute without error (both a syntax check and, wherever the example does not require downloading an external dataset or GPU hardware, a full run producing the stated output) as part of this book's production process. Code lines are kept to a width that fits this book's print trim size without wrapping — a formatting convention rather than a Python style requirement, so code

copied from this book into a wider editor may reasonably be reformatted to a more typical 79- or 99-character line length. Every dataset or text corpus referenced in a worked example is drawn from public-domain or explicitly openly-licensed sources (see Appendix D for the full list) — no example in this book relies on data of unclear or restricted license.

Environment Setup

Installing Python and Core Libraries

This book's code was written and verified against Python 3.10+, with the following core libraries:

```
pip install numpy scipy torch torchvision scikit-learn matplotlib
```

For chapters using specific additional libraries (XGBoost/LightGBM in Chapter 10; Gymnasium-style environments if extending the from-scratch RL environments in Part IV to standard benchmark environments), install them individually as needed — no chapter requires a library beyond what its own code samples import.

Verifying Your Installation

```
import sys
import numpy
import torch


print(f"Python version: {sys.version}")
print(f"NumPy version: {numpy.__version__}")
print(f"PyTorch version: {torch.__version__}")
print(f"CUDA available: {torch.cuda.is_available()}")
```

A CUDA-capable GPU is not required for any example in this book to run correctly — every worked example was designed to complete in a reasonable time on CPU alone, though the deep learning examples in Parts III and V will run substantially faster with GPU acceleration if available.

Running the Code Samples

Each chapter's code is presented as a sequence of complete, self-contained blocks; concatenating a chapter's blocks in order (as printed) and running them as a single script reproduces every result and printed output described in that chapter's worked example section.

Notation and Mathematical Conventions

- Scalars: lowercase italic (x, y, α).
- Vectors: lowercase bold or explicit component notation (x, or x = (x_1, ..., x_n)); vectors are column vectors by default.
- Matrices: uppercase (X, W, Σ).
- Random variables: uppercase (X, Y); their realizations: lowercase (x, y).
- P(·): probability mass or density, used generically; f(·) or p(·) for an explicit density function when needed to avoid ambiguity.
- E[·]: expectation. Var(·): variance. Cov(·,·): covariance.
- ∇_θ f: gradient of f with respect to θ.
- ‖·‖ without a subscript denotes the Euclidean (L2) norm; ‖·‖_1, ‖·‖_∞ denote L1 and L-infinity norms explicitly when used.
- σ(·) denotes the logistic sigmoid, σ(z) = 1/(1+e^{-z}), unless otherwise stated in context (a small number of chapters also use σ for a standard deviation or a state in an MDP; each such use is defined locally at first appearance).
- γ denotes a discount factor (Part IV) unless otherwise defined locally.
- Big-O notation (O(·)) describes asymptotic worst-case growth throughout, in the standard computer-science sense.
- Theorem/proof formatting: results stated as a **Theorem** are followed by either a full **Proof** or an

explicitly labeled **Proof sketch** when a complete formal proof would require machinery outside this book's scope; a proof sketch always states which steps are omitted and points to a citation for the complete argument where one exists.

Dataset and Library Index

Every dataset, corpus, and library referenced anywhere in this book is listed here with its license/access basis. This book deliberately restricts itself to public-domain and openly-licensed sources throughout.

Datasets and Corpora

- **MNIST** — public-domain-licensed handwritten digit dataset (Chapters 28, 30, 39).
- **Fashion-MNIST** — MIT-licensed, open drop-in replacement for MNIST (Chapters 19, 29).
- **UCI Machine Learning Repository** — a collection of openly-licensed datasets suitable for the classical ML methods of Part II; referenced generically as a source for exercises requiring real (non-synthetic) tabular data.
- **Project Gutenberg** — public-domain text corpora, used (in small, illustrative excerpts) for the character-level language modeling examples in Chapters 20 and 31.
- **Synthetic/generated data** — the majority of this book's worked examples use data generated directly in code (via NumPy's random number generators with a fixed seed for reproducibility) specifically so every example is self-contained, requires no external download, and involves no licensing question whatsoever.

Core Libraries

- **NumPy** — numerical array computation, used throughout, especially Parts I-II and for from-scratch algorithm implementations.
- **PyTorch** — automatic differentiation and neural network layers, used throughout Parts III, IV (deep RL), and V.
- **torchvision** — dataset loaders and vision utilities, used in Chapters 19, 28, 29, 30.
- **scikit-learn** — referenced for production-grade comparison implementations in several Part II exercises (this book's main text always implements the core algorithm from scratch first).

Full citation and license details for each dataset are available from their respective maintaining organizations at the time of writing; readers should independently verify current license terms before using any dataset in a new project, since terms can change over time.

PART I — FOUNDATIONS & TRADITIONAL AI

The chapters in this part cover the search, logic, planning, and reasoning techniques that defined artificial intelligence before statistical learning became dominant. These methods remain the right tool whenever a problem has an explicit, well-defined structure -- a game tree, a constraint graph, a formal knowledge base -- rather than requiring a model learned from data.

Chapter 1: Uninformed Search

1.1 Problems as State Spaces

Many tasks that look nothing alike — solving a sliding puzzle, routing a delivery truck, planning a robot arm's joint angles — share a common skeleton: a set of possible configurations (states), a way to move between them (actions), and a target configuration (the goal). Formalizing a task this way is the first and most important step in classical AI, because once a problem is expressed as a state space, a general-purpose search algorithm can solve it without knowing anything about trucks or puzzles.

A **state-space search problem** is a tuple (S, s0, ACTIONS, RESULT, GOAL-TEST, COST):

- S: the set of all possible states.
- s0 ∈ S: the initial state.
- ACTIONS(s): the set of actions available in state s.
- RESULT(s, a): the state reached by applying action a in state s.
- GOAL-TEST(s): a predicate that is true when s is a goal state.
- COST(s, a, s'): the (non-negative) cost of taking action a from s to reach s'.

A **solution** is a sequence of actions leading from s0 to a state satisfying GOAL-TEST. An **optimal solution** has the lowest total cost among all solutions. The **search tree** implicitly defined by this tuple has s0 at its root; each node's children are the states reachable by one action. Search algorithms differ in the order they explore this tree.

1.2 Breadth-First Search

Breadth-first search (BFS) expands nodes in order of their distance (in action-steps) from s0: it fully explores all nodes at depth d before any node at depth d+1. Implemented with a FIFO queue, BFS is guaranteed to find a shallowest solution first.

Theorem 1.1 (BFS optimality with uniform step cost). If every action has the same cost c, BFS returns an optimal solution.

Proof. BFS expands nodes in non-decreasing order of depth. Since cost = c × depth for a uniform step cost, non-decreasing depth order is also non-decreasing cost order. The first goal node BFS expands is therefore at the shallowest depth among all goal nodes, hence the lowest cost. ■

BFS's time and space complexity are both O(b^d), where b is the branching factor and d is the depth of the shallowest solution — the entire frontier at depth d must be held in memory, which is BFS's dominant practical limitation.

1.3 Depth-First Search

Depth-first search (DFS) expands the most recently generated node first, implemented with a LIFO stack (or recursion). DFS is not optimal — it can return a very long solution when a short one exists elsewhere in the tree — and on infinite or cyclic state spaces without cycle detection it can fail to terminate. Its redeeming property is space: DFS needs only O(bm) memory, where m is the maximum depth of the tree, since it only holds one path plus siblings at each level rather than the entire frontier.

1.4 Uniform-Cost Search

When step costs are not uniform, uniform-cost search (UCS) generalizes BFS by always expanding the frontier node with the lowest path cost g(n), using a priority queue instead of a FIFO queue.

Theorem 1.2 (UCS optimality). UCS returns an optimal solution whenever all step costs are non-negative.

Proof sketch. UCS expands nodes in non-decreasing order of g(n). Suppose UCS first expands a goal node G with g(G) > C*, the true optimal cost. Then the optimal path's next unexpanded node n satisfies g(n) ≤ C* < g(G) (path costs only grow, since costs are non-negative), so n would have been expanded before G — a contradiction. ■

UCS is exactly A* (Chapter 2) with h(n) = 0 for all n; every general result we prove about A* in Chapter 2 specializes to UCS as the degenerate case of an uninformative heuristic.

1.5 Iterative Deepening

Iterative deepening search (IDS) runs depth-limited DFS repeatedly with limits 0, 1, 2, ... until a solution is found. This looks wasteful — states at shallow depths are regenerated on every iteration — but the overhead is small: in a tree with branching factor b, the number of nodes at depth d dominates the sum of all shallower levels by a factor of roughly b/(b-1), so the total work is within a constant factor of a single BFS pass at depth d, while using only DFS's O(bd) memory. IDS is the default choice when the solution depth is unknown and memory is limited, which is the common case in practice.

1.6 Python: A Generic Search Framework

```
from collections import deque
from dataclasses import dataclass, field
from typing import Any, Callable, List, Optional
import heapq



@dataclass
class Node:
    state: Any
    parent: Optional["Node"] = None
    action: Any = None
    path_cost: float = 0.0
    depth: int = 0



def build_path(node: Node) -> List[Any]:
    actions = []
    while node.parent is not None:
        actions.append(node.action)
        node = node.parent
    actions.reverse()
    return actions



def bfs(problem) -> Optional[List[Any]]:
    start = Node(problem.initial_state)
    if problem.goal_test(start.state):
        return build_path(start)
    frontier = deque([start])
    visited = {start.state}
    while frontier:
        node = frontier.popleft()
        for action in problem.actions(node.state):
            child_state = problem.result(node.state, action)
            if child_state in visited:
                continue
            step_cost = problem.cost(node.state, action, child_state)
            child = Node(
                child_state, node, action,
                node.path_cost + step_cost, node.depth + 1,
            )
            if problem.goal_test(child_state):
                return build_path(child)
            visited.add(child_state)
            frontier.append(child)
    return None
```

```
def dfs_limited(problem, limit: int) -> Optional[List[Any]]:
    def recurse(node, depth_remaining, visited):
        if problem.goal_test(node.state):
            return build_path(node)
        if depth_remaining == 0:
            return None
        for action in problem.actions(node.state):
            child_state = problem.result(node.state, action)
            if child_state in visited:
                continue
            step_cost = problem.cost(node.state, action, child_state)
            child = Node(
                child_state, node, action,
                node.path_cost + step_cost, node.depth + 1,
            )
            result = recurse(
                child, depth_remaining - 1, visited | {child_state}
            )
            if result is not None:
                return result
        return None

    start = Node(problem.initial_state)
    return recurse(start, limit, {start.state})



def iterative_deepening(problem, max_depth: int = 1000):
    for limit in range(max_depth + 1):
        result = dfs_limited(problem, limit)
        if result is not None:
            return result
    return None



def uniform_cost_search(problem) -> Optional[List[Any]]:
    counter = 0
    start = Node(problem.initial_state)
    frontier = [(0.0, counter, start)]
    best_cost = {start.state: 0.0}
    while frontier:
        cost, _, node = heapq.heappop(frontier)
        if cost > best_cost.get(node.state, float("inf")):
            continue
        if problem.goal_test(node.state):
            return build_path(node)
        for action in problem.actions(node.state):
```

```
            child_state = problem.result(node.state, action)
            step_cost = problem.cost(node.state, action, child_state)
            new_cost = node.path_cost + step_cost
            if new_cost < best_cost.get(child_state, float("inf")):
                best_cost[child_state] = new_cost
                child = Node(
                    child_state, node, action, new_cost, node.depth + 1
                )
                counter += 1
                heapq.heappush(frontier, (new_cost, counter, child))
    return None
```

The `Problem` interface expected by these functions exposes `initial_state`, `actions(state)`, `result(state, action)`, `goal_test(state)`, and `cost(state, action, next_state)` — precisely the six-tuple formalization of section 1.1, translated directly into code. Any concrete problem (the sliding puzzle below, the CSPs of Chapter 4, the planning problems of Chapter 6) need only implement this interface to reuse every search algorithm in this book.

1.7 Worked Example: The 8-Puzzle

```
class EightPuzzle:
    GOAL = (1, 2, 3, 4, 5, 6, 7, 8, 0)  # 0 = blank


    def __init__(self, initial_state):
        self.initial_state = initial_state


    def actions(self, state):
        blank = state.index(0)
        row, col = divmod(blank, 3)
        moves = []
        if row > 0:
            moves.append("UP")
        if row < 2:
            moves.append("DOWN")
        if col > 0:
            moves.append("LEFT")
        if col < 2:
            moves.append("RIGHT")
        return moves


    def result(self, state, action):
```

```
        blank = state.index(0)
        row, col = divmod(blank, 3)
        delta = {
            "UP": -3, "DOWN": 3, "LEFT": -1, "RIGHT": 1,
        }[action]
        target = blank + delta
        state = list(state)
        state[blank], state[target] = state[target], state[blank]
        return tuple(state)


    def goal_test(self, state):
        return state == self.GOAL


    def cost(self, state, action, next_state):
        return 1



if __name__ == "__main__":
    start = (1, 2, 3, 4, 5, 6, 7, 0, 8)  # one move from solved
    puzzle = EightPuzzle(start)
    plan = iterative_deepening(puzzle, max_depth=10)
    print("Plan:", plan)
```

Running this solves the one-move puzzle instantly; try `(1, 2, 3, 4, 0, 6, 7, 5, 8)` (a few moves away) and observe how quickly the cost of uninformed search grows with distance from the goal — this growth is exactly the motivation for the heuristics introduced in Chapter 2.

1.8 Exercises

1. Prove that DFS on a finite, acyclic state space (with visited-state tracking) always terminates.
2. Show that iterative deepening's total node expansions are O(b^d) — within a constant factor of BFS — by summing a geometric series over the depths 0 through d.
3. Modify `uniform_cost_search` to also return the total path cost, and verify it matches the number of actions

returned by `bfs` on a uniform-cost version of the 8-puzzle.

4. Implement bidirectional BFS (searching simultaneously from the start and the goal) for the 8-puzzle and measure the reduction in nodes expanded compared to single-direction BFS.

5. Construct a state space where DFS without cycle detection never terminates, and show that adding a `visited` set fixes it.

Chapter 2: Informed Search & Heuristics

2.1 From Uninformed to Informed Search

Uninformed search algorithms (Chapter 1) treat every unexplored node identically: BFS expands nodes in order of discovery, and uniform-cost search expands nodes in order of accumulated path cost g(n). Neither uses any knowledge about how close a node is to the goal. Informed search closes this gap by introducing a **heuristic function** h(n): an estimate of the cost from node n to the nearest goal. A good heuristic lets a search algorithm ignore large portions of the state space that uninformed search would blindly explore.

We formalize the state-space search problem as a tuple (S, s0, ACTIONS, RESULT, GOAL-TEST, COST), where S is the set of states, s0 the initial state, ACTIONS(s) the actions available in state s, RESULT(s, a) the successor state, GOAL-TEST(s) a predicate, and COST(s, a, s') ≥ 0 the step cost.

2.2 Greedy Best-First Search

Greedy best-first search expands the node that minimizes h(n) alone, ignoring the cost already paid to reach it. This is fast but not optimal: it can be lured down a costly path simply because the immediate heuristic estimate looks promising, and it can loop indefinitely on certain graphs without a "closed set."

2.3 A* Search

A* combines the two signals: it expands the node minimizing

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

where g(n) is the exact cost of the best path found so far from s0 to n, and h(n) is the heuristic estimate of the remaining cost from n to a goal.

2.3.1 Admissibility and Optimality

Definition (Admissible heuristic). h is admissible if for every node n, $0 \le h(n) \le h^*(n)$, where $h^*(n)$ is the true cost of the cheapest path from n to a goal. An admissible heuristic never overestimates.

Theorem 2.1 (Optimality of Tree-Search A*). If h is admissible, A* using tree search (no duplicate detection) returns an optimal solution.

Proof sketch. Suppose A* terminates by selecting a suboptimal goal G2 with $g(G2) > C^*$, where C^* is the true optimal cost. At the moment of termination there must exist a node n on the optimal path that is still on the frontier (since the path to the true optimal goal G was not fully expanded, some prefix of it remains unexpanded — the frontier of a tree search always contains the boundary of explored territory). For that node, $f(n) = g(n) + h(n) \le g(n) + h^*(n) = C^*$ (using admissibility of h). Since $f(G2) = g(G2) > C^* \ge f(n)$, node n would have been selected for expansion before G2, contradicting the assumption that G2 was selected first. ■

2.3.2 Consistency

Definition (Consistent heuristic). h is consistent (monotone) if for every node n and successor n' generated by action a, h(n) ≤ COST(n, a, n') + h(n').

Consistency is a stronger condition than admissibility (every consistent heuristic is admissible, but not vice versa), and it guarantees that the sequence of f-values along any path is non-decreasing. This means that when A* is run with **graph search** (a closed set that prevents re-expansion), the first time a node is popped from the frontier, it is popped with its optimal g-value — so re-expansion is never needed, and A* graph search remains optimal.

2.4 Complexity

In the worst case A*'s time and space complexity are exponential in the depth of the solution, exactly like uninformed search — the exponent is governed by the heuristic's error relative to h*. In practice, a well-designed heuristic reduces the *effective branching factor* dramatically. Space is the binding constraint in most applications, since A* must keep all generated nodes in memory; this motivates memory-bounded variants such as IDA* (iterative-deepening A*) and SMA* (simplified memory-bounded A*), covered in section 2.6.

2.5 Designing Heuristics

Two general recipes for constructing admissible heuristics:

1. **Relaxed problems.** Remove a constraint from the original problem; the optimal solution cost of the relaxed problem is always ≤ the cost of the original,

hence admissible. Example: in the 8-puzzle, "Manhattan distance" relaxes the rule that a tile can only move into the blank space, allowing it to move directly to its goal position.

2. **Pattern databases.** Precompute exact solution costs for a sub-problem (e.g., a subset of tiles) and store them in a lookup table; use the maximum over several disjoint pattern databases as the heuristic, since disjoint sub-problems' costs are additive and each remains a lower bound.

Two admissible heuristics h1, h2 can always be combined via h(n) = max(h1(n), h2(n)), which is itself admissible and dominates (is at least as informed as) either one alone.

2.6 Python: A* From Scratch

```
import heapq
from dataclasses import dataclass, field
from typing import Callable, Dict, List, Tuple, Optional


State = Tuple[int, int]  # (row, col) for a grid; generalize as needed


@dataclass(order=True)
class _Entry:
    priority: float
    counter: int
    state: State = field(compare=False)



def astar(
    start: State,
    goal: State,
    neighbors_fn: Callable[[State], List[Tuple[State, float]]],
    heuristic_fn: Callable[[State, State], float],
) -> Optional[Tuple[List[State], float]]:
    """Generic A* search.


    neighbors_fn(state) -> list of (next_state, step_cost)
    heuristic_fn(state, goal) -> admissible estimate of remaining cost
```

```
    Returns (path, total_cost) or None if no path exists.
    """
    counter = 0
    open_set: List[_Entry] = []
    start_h = heuristic_fn(start, goal)
    heapq.heappush(open_set, _Entry(start_h, counter, start))

    g_score: Dict[State, float] = {start: 0.0}
    came_from: Dict[State, State] = {}
    closed: set = set()

    while open_set:
        entry = heapq.heappop(open_set)
        current = entry.state

        if current in closed:
            continue
        if current == goal:
            path = _reconstruct_path(came_from, current)
            return path, g_score[current]

        closed.add(current)

        for neighbor, step_cost in neighbors_fn(current):
            tentative_g = g_score[current] + step_cost
            if tentative_g < g_score.get(neighbor, float("inf")):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f = tentative_g + heuristic_fn(neighbor, goal)
                counter += 1
                heapq.heappush(open_set, _Entry(f, counter, neighbor))

    return None


def _reconstruct_path(
    came_from: Dict[State, State], current: State
) -> List[State]:
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path
```

Notes on the implementation:

- We use a lazy-deletion priority queue (`heapq`): stale entries are skipped via the `closed` set check rather than removed from the heap, which avoids the complexity of a decrease-key operation.
- `_Entry` carries a monotonically increasing `counter` purely to break ties deterministically and to avoid comparing `State` tuples when priorities are equal (Python's heap would otherwise try to compare states, which may not support `<`).
- Because we check `tentative_g < g_score.get(neighbor, inf)` before pushing, this implementation is correct for both consistent and merely-admissible heuristics, at the cost of allowing a node to be pushed multiple times (harmless, since stale copies are discarded on pop).

2.7 Worked Example: Grid Pathfinding with Obstacles

```
def make_grid_neighbors(grid: List[List[int]]):
    """grid[r][c] == 1 means blocked. Returns a neighbors_fn."""
    rows, cols = len(grid), len(grid[0])

    def neighbors(state):
        r, c = state
        result = []
        for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nr, nc = r + dr, c + dc
            if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 0:
                result.append(((nr, nc), 1.0))
        return result

    return neighbors



def manhattan(a: State, b: State) -> float:
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

```
if __name__ == "__main__":
    grid = [
        [0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 1, 0],
        [1, 1, 0, 1, 0],
        [0, 0, 0, 0, 0],
    ]
    start, goal = (0, 0), (4, 4)
    result = astar(start, goal, make_grid_neighbors(grid), manhattan)
    if result:
        path, cost = result
        print(f"Path found, cost={cost}:")
        for step in path:
            print(step)
    else:
        print("No path exists.")
```

Manhattan distance is admissible here because each grid move costs exactly 1 and can reduce the remaining Manhattan distance by at most 1 — it never overestimates the true remaining cost, and since a move's cost equals its effect on the heuristic exactly when unobstructed, it is also consistent. Running this example finds the shortest path around the two obstacle walls; try setting `grid[2][2] = 1` to block the only gap and confirm the function correctly returns `None`.

2.8 Exercises

1. Prove that if h is consistent, the f-values along any path expanded by A* graph search are non-decreasing.
2. Show by counterexample that an inadmissible heuristic can cause A* to return a suboptimal path.
3. Implement weighted A* (f(n) = g(n) + w·h(n) for w > 1) and empirically measure the trade-off between solution optimality and nodes expanded on the 8-puzzle.

4. Extend the grid example to allow diagonal moves at cost √2, and design a consistent heuristic (Chebyshev or octile distance) for it.
5. Implement IDA* using the `neighbors_fn`/`heuristic_fn` interface above and compare peak memory usage against `astar()` on a large maze.

Chapter 3: Adversarial Search & Game Playing

3.1 Games as Search Problems

A two-player, zero-sum, perfect-information game (chess, tic-tac-toe, Connect Four) extends the state-space formalism of Chapter 1 with a turn function and a terminal utility: TO-MOVE(s) identifies which player acts in state s, and UTILITY(s, p) gives player p's payoff at a terminal state s. Because the opponent chooses their own moves — typically to minimize our payoff — a plan cannot simply be a fixed action sequence; it must be a policy that responds to whatever the opponent does.

3.2 Minimax

The minimax value of a state assumes both players play optimally: the maximizing player (MAX) picks the action leading to the highest minimax value, and the minimizing player (MIN) picks the action leading to the lowest.

$$\textit{MINIMAX}(s) =$$

UTILITY(s) if TERMINAL-TEST(s)

max_{a} MINIMAX(RESULT(s, a)) if TO-MOVE(s) = MAX

min_{a} MINIMAX(RESULT(s, a)) if TO-MOVE(s) = MIN

Theorem 3.1. Minimax is optimal against an optimal opponent: no strategy achieves a higher guaranteed payoff for MAX than always choosing the minimax-maximizing action.

Proof sketch. By backward induction on the finite game tree. At a terminal node, the claim holds trivially (the value is the utility). Assume it holds for all children of a node s. If TO-MOVE(s) = MAX, choosing the child with the highest minimax value guarantees at least that value regardless of MIN's subsequent play (by the inductive hypothesis applied to that child), and no other choice can guarantee more, since every other child's value is by definition ≤ the maximum. The symmetric argument holds for MIN. ■

Minimax requires visiting every node of the game tree: O(b^m) time for branching factor b and depth m — intractable for real games (chess has b ≈ 35, m ≈ 80).

3.3 Alpha-Beta Pruning

Alpha-beta pruning computes the same value as minimax while skipping branches that provably cannot affect the final decision. It maintains two bounds while traversing: α, the best value MAX can guarantee so far on the path to the root, and β, the best value MIN can guarantee so far. Whenever α ≥ β at a node, the remaining children can be pruned — the opponent already has a better alternative elsewhere, so this branch will never be chosen.

Theorem 3.2. Alpha-beta pruning returns the same action as minimax.

Proof sketch. Pruning only discards a subtree when α ≥ β, meaning MIN (or MAX) already has an alternative at an ancestor node that is at least as good as anything the pruned subtree could produce for the other player — by the minimax recurrence, a node's exact value is therefore unaffected by ignoring that subtree, since it could not have been selected regardless. Formally this follows by induction on tree depth, mirroring Theorem 3.1's induction

with the additional invariant that any pruned branch's true value lies outside the (α, β) window and so cannot change the max/min at any ancestor. ■

With a good move-ordering heuristic (trying the most promising moves first), alpha-beta reduces the effective branching factor from b to roughly √b in the best case — doubling the search depth achievable in the same time budget, since O((√b)^m) = O(b^(m/2)).

3.4 Monte Carlo Tree Search

For games with very large or hard-to-evaluate state spaces (Go, general game playing), neither minimax nor a hand-crafted evaluation function is practical. Monte Carlo Tree Search (MCTS) instead estimates a node's value by simulation: repeatedly playing random (or policy-guided) games to completion from that node and averaging the outcomes. MCTS has four phases, repeated for a fixed budget of iterations:

1. **Selection.** Starting at the root, repeatedly choose child nodes according to a rule that balances exploiting known-good moves against exploring under-visited ones, most commonly Upper Confidence Bound for Trees (UCT):

   $$UCT(n) = w(n)/v(n) + c \cdot \sqrt{(\ln v(parent) / v(n))}$$

   where w(n) is the total reward accumulated through node n, v(n) is its visit count, and c controls the exploration/exploitation trade-off.

2. **Expansion.** Add one or more child nodes of the selected leaf.

3. **Simulation (rollout).** Play a random (or lightweight-policy) game from the new node to a terminal state.

4. **Backpropagation.** Propagate the simulation's outcome back up every node visited in this iteration, updating w and v.

After the budget is exhausted, the root's child with the highest visit count (not highest average value — visit count is a more robust and less noise-sensitive signal) is returned as the move to play.

3.5 Python: Minimax with Alpha-Beta Pruning

```
from typing import Callable, List, Optional, Tuple, Any



def alphabeta(
    state: Any,
    depth: int,
    alpha: float,
    beta: float,
    maximizing: bool,
    get_actions: Callable[[Any], List[Any]],
    apply_action: Callable[[Any, Any], Any],
    is_terminal: Callable[[Any], bool],
    evaluate: Callable[[Any], float],
) -> float:
    if is_terminal(state) or depth == 0:
        return evaluate(state)


    actions = get_actions(state)
    if maximizing:
        value = float("-inf")
        for a in actions:
            child = apply_action(state, a)
            value = max(
                value,
                alphabeta(
                    child, depth - 1, alpha, beta, False,
                    get_actions, apply_action, is_terminal, evaluate,
                ),
            )
            alpha = max(alpha, value)
            if alpha >= beta:
                break  # beta cutoff
        return value
```

```
    else:
        value = float("inf")
        for a in actions:
            child = apply_action(state, a)
            value = min(
                value,
                alphabeta(
                    child, depth - 1, alpha, beta, True,
                    get_actions, apply_action, is_terminal, evaluate,
                ),
            )
            beta = min(beta, value)
            if alpha >= beta:
                break  # alpha cutoff
        return value



def best_move(
    state, get_actions, apply_action, is_terminal, evaluate, depth=9
):
    best_val, best_a = float("-inf"), None
    alpha, beta = float("-inf"), float("inf")
    for a in get_actions(state):
        child = apply_action(state, a)
        val = alphabeta(
            child, depth - 1, alpha, beta, False,
            get_actions, apply_action, is_terminal, evaluate,
        )
        if val > best_val:
            best_val, best_a = val, a
        alpha = max(alpha, best_val)
    return best_a, best_val
```

3.6 Worked Example: Tic-Tac-Toe

```
def ttt_actions(board):
    return [i for i, v in enumerate(board) if v == 0]



def ttt_apply(board, action, player=None):
    board = list(board)
    player = player or (1 if board.count(1) == board.count(-1) else -1)
    board[action] = player
    return tuple(board)
```

```
LINES = [
    (0, 1, 2), (3, 4, 5), (6, 7, 8),
    (0, 3, 6), (1, 4, 7), (2, 5, 8),
    (0, 4, 8), (2, 4, 6),
]



def ttt_winner(board):
    for a, b, c in LINES:
        s = board[a] + board[b] + board[c]
        if s == 3:
            return 1
        if s == -3:
            return -1
    return 0



def ttt_terminal(board):
    return ttt_winner(board) != 0 or 0 not in board



def ttt_evaluate(board):
    return ttt_winner(board)  # +1 win for X, -1 win for O, 0 draw/ongoing



def ttt_apply_alternating(board, action):
    # infer whose turn it is from piece counts, so best_move's
    # single-argument apply_action signature stays generic
    x_count = board.count(1)
    o_count = board.count(-1)
    player = 1 if x_count == o_count else -1
    return ttt_apply(board, action, player)



if __name__ == "__main__":
    empty = tuple([0] * 9)
    move, value = best_move(
        empty, ttt_actions, ttt_apply_alternating,
        ttt_terminal, ttt_evaluate, depth=9,
    )
    print(f"Best opening move: {move}, minimax value: {value}")
    # Tic-tac-toe is a solved game: perfect play from an empty
    # board always evaluates to 0 (a draw).
```

Because tic-tac-toe's game tree has only 9! ≤ 362,880 leaf paths, full-depth minimax with alpha-beta solves it instantly and confirms the well-known result that optimal play draws.

3.7 Exercises

1. Instrument `alphabeta` to count nodes visited, and empirically compare that count against plain minimax on tic-tac-toe with and without move ordering (try central squares first).
2. Prove that alpha-beta pruning with perfect move ordering visits O(b^(m/2)) nodes in the worst case.
3. Implement a simple UCT-based MCTS player for Connect Four using random rollouts, and measure win rate against a fixed-depth minimax player as the simulation budget increases.
4. Extend `alphabeta` to handle non-zero-sum or multi-player games via the more general "maxn" algorithm, where each node maximizes its own player's utility component.
5. Add a transposition table (memoizing evaluated states) to `alphabeta` and measure the speedup on a game with many transpositions, such as Connect Four.