Artificial Intelligence and Machine Learning (Unofficial) Notes

Gudfit

Contents

Chapter 1

Introduction

1.1 AI versus ML

Definition 1.1.1. ***Artificial Intelligence***. Artificial Intelligence (AI) is the field dedicated to creating machines or computers capable of performing tasks that typically require human intelligence.

The pursuit of AI mirrors early attempts at human flight. One approach was to mimic nature by designing machines that flapped like birds. The successful alternative focused on understanding aerodynamics to design flying objects that did not resemble birds. Similarly, in AI, one path attempts to replicate human thought processes, while another focuses on developing systems that achieve intelligent outcomes, regardless of the underlying mechanism.

To measure machine intelligence, Alan Turing proposed the Turing Test in 1950 [1]. It posits that if an AI system can communicate with a human interrogator and convince them they are communicating with another human, the system exhibits intelligent behaviour. A modern application of this concept is the CAPTCHA, a task designed to differentiate human and machine users online.

While early AI research focused on logic and rule-based systems to encode knowledge, a dominant modern approach is Machine Learning (ML), a subfield of computer science. Traditionally, AI was viewed as applied logic, using knowledge bases and rule-based systems. In contrast, ML is rooted in statistical theory for prediction, while the related field of Data Mining focuses on uncovering hidden patterns within data. Although the broader field of AI maintains the goal of achieving Artificial General Intelligence (AGI), ML is predominantly concerned with developing systems that excel at a single, specific task.

Two formal definitions help to characterise Machine Learning precisely:

Definition 1.1.2. ***Machine Learning***. ML is the subfield of computer science that gives computers the ability to learn without being explicitly programmed [2].

Definition 1.1.3. ***Learning***. A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E [3].

The process involves training, or fitting, a model on historical data. This trained model is then used to make predictions on new, unseen data. This data-driven approach contrasts with the explicit,

knowledge-based programming of what is now known as Good Old-Fashioned AI (GOFAI), the prevailing paradigm before the dominance of Machine Learning.

Chapter 2

Artificial Intelligence

2.1 Paradigms of Artificial Intelligence

2.1.1 Good Old-Fashioned AI (GOFAI)

GOFAI, prevalent from the 1950s to the 1980s, is an approach to AI based on symbolic reasoning [4], positing that cognition is a computational process of symbol manipulation. This paradigm assumes that intelligence can be achieved by using human-readable symbolic representations of problems and knowledge, then applying mathematical logic or search to infer solutions. Key aspects include:

- **Knowledge Representation.** Information is encoded explicitly using symbols, such as logical statements or if-then rules. Researchers create logic representations with stored declarations about the world, allowing a computer to perform logical reasoning.
- **Logic and Rule-Based Systems.** Reasoning is performed by manipulating these symbols via predefined rules. Expert systems are a prime example, where specialists define facts and rules to create a knowledge base that users can query.
- **Human-Crafted Models.** GOFAI systems rely heavily on human-designed structures, such as decision trees, ontologies, and rule sets, to guide their behaviour.

A rule-based email spam filter exemplifies this approach. Such a system uses predefined rules based on spam characteristics, such as keywords (`free`, `limited-time offer`), excessive punctuation, or suspicious URLs. If an email meets a certain threshold of these characteristics, it is flagged as spam.

GOFAI has notable limitations. Rule-based systems struggle to scale in complex environments and handle the ambiguous, noisy, or incomplete information common in the real world. Crucially, they do not learn from data; every new rule or adaptation requires direct human input. In contrast, Machine Learning models, such as neural networks, learn patterns and representations directly from data. While Symbolic AI is flexible and interpretable, it can struggle to capture complex patterns. An integrated approach, such as in neuro-symbolic methods, combines the strengths of both paradigms by using neural networks to learn from data and symbolic systems to perform logical inference [5].

2.1.2 Rational Agents

Whether the underlying methodology is symbolic like GOFAI or statistical like ML, the objective is often to design systems that act correctly. We aim to design objects that can act rationally, termed

rational agents.

An agent is an entity that perceives its environment through sensors and acts upon that environment through actuators. As shown in Figure 2.1, an agent's behaviour is governed by an agent function that maps a history of percepts to an action [6]. [1]

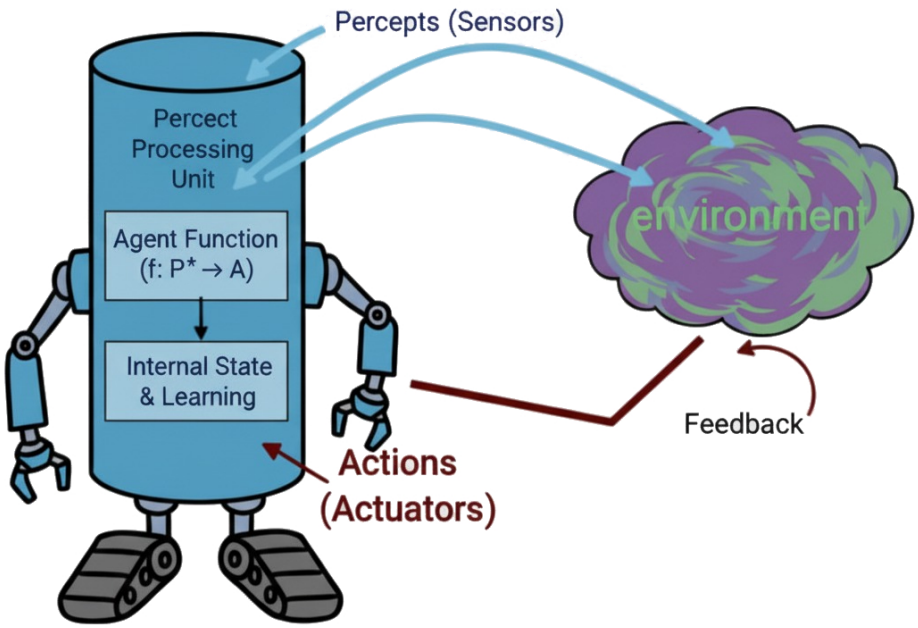


Figure 2.1: The basic setup of an AI agent interacting with its environment.

Definition 2.1.1. ***Agent Function and Agent Program.*** The Agent Function is a map $f : P^* \mapsto A$, where P^* is a sequence of percepts and A is an action. The Agent Program implements this function, which runs on the physical architecture.

A rational agent strives to do the right thing. To formalise this, we must first define the agent's context.

Task Environment To create and evaluate a rational agent, we specify its task environment using the PEAS framework:

- **P**erformance Measure: An objective criterion for success (for a vacuum cleaner, the amount of dirt cleaned, time taken, and electricity consumed).
- **E**nvironment: The world in which the agent operates.
- **A**ctuators: The components of the agent for performing actions (e.g., wheels, brushes).
- **S**ensors: The agent's components for perceiving the environment (e.g., camera, dirt sensor).

Task environments have several key properties that heavily influence agent design:

[1] This image is AI-generated, based on an image from GeeksforGeeks, and may contain inaccuracies.

- **Fully vs Partially Observable:** Can the sensors access the complete state of the environment?
- **Deterministic vs Stochastic:** Is the next state entirely determined by the current state and action?
- **Episodic vs Sequential:** Does the choice of action depend on previous actions?
- **Static vs Dynamic:** Does the environment change while the agent deliberates?
- **Discrete vs Continuous:** Does the environment have a finite number of distinct states and actions?
- **Single vs Multi-agent:** Is the agent operating by itself?

The real world is typically partially observable, stochastic, sequential, dynamic, continuous, and multi-agent.

Types of Agent Programs Implementing an agent function as a lookup table mapping every possible percept sequence to an action is impractical due to the table's enormous size. Instead, agent programs are designed with more efficient structures. Four basic agent programs are detailed in Table 2.1.

Table 2.1: Four basic kinds of agent programs.

Agent Program	Description	Example
Simple Reflex	Maps the current percept directly to an action using condition-action rules.	Thermostat: If temp $>$ setpoint $\rightarrow$ activate cooling.
Model-Based Reflex	Maintains an internal state to handle partial observability and model the world's evolution.	Autonomous vehicle: Tracks other cars' positions and velocities, even when momentarily obscured.
Goal-Based	Considers future actions and their outcomes to find a sequence that achieves a specific goal.	Route planner: Searches for a sequence of roads to travel from an origin to a destination.
Utility-Based	Acts to maximise a utility function, enabling trade-offs between conflicting objectives.	Mars Rover: Balances maximising scientific data collection against minimising energy use and risk.

A goal-based agent tries to achieve a specific goal, whereas a utility-based agent seeks to maximise its utility, which is ideally aligned with the external performance measure. Furthermore, a learning agent can improve its performance over time, and an autonomous agent can update its program based on experience.

Remark. The distinction between goal-based agents and learning agents is often blurred; a learning agent is typically designed to achieve a goal, using its experience to improve its performance. In these notes, we focus on the design of goal-based agents and explore the two primary methodologies for implementing their agent functions. We will begin with classical AI techniques, such as search algorithms, which find explicit paths to a goal. Subsequently, we will delve into how modern machine learning allows an agent to learn how to achieve its goals from data.

2.2 Formal Logic in Artificial Intelligence (FLAI)

The symbolic approach to AI, or GOFAI, posits that intelligence can be achieved by manipulating symbolic representations of knowledge. Rather than enumerating an intractably large state space, logic provides a compact language for describing sets of states. For instance, the sentence "It is raining" is a short description for the vast set of all world states in which it is raining. Logic provides tools to implicitly manipulate these short descriptions to reason about the world. Systems built on this paradigm, such as expert systems, rely on a logical internal representation comprising variables and rules, the foundation of which is a formal logic.[2]

2.2.1 Logical Agents and Knowledge Bases

A logical agent combines the paradigms of rational agents and formal logic, using a knowledge base (KB).

Definition 2.2.1. ***Knowledge Base***. A Knowledge Base is a set of sentences in a formal language that represents knowledge of the world.

The agent can `TELL` the KB new facts derived from percepts and `ASK` the KB what action to take. The answers are derived through inference. For example, an agent might use percepts to infer hidden properties of the world or deduce the outcome of a potential action.

Consider a simple automated weather agent whose knowledge base contains the rule:

$$WetUmbrella \rightarrow IsRaining$$

If the agent perceives a person with a wet umbrella, it can `TELL` its KB the new fact, *WetUmbrella*. Through a simple inference step, the agent can conclude that *IsRaining* is true, updating its understanding of the world without direct observation. This ability to derive new knowledge from existing facts is the core of a logical agent.

Definition 2.2.2. ***Logic***. A formal language defined by three components:

- **Syntax:** Rules specifying which expressions are legally formed sentences.
- **Semantics:** Rules determining the meaning or truth of sentences in some world or model.
- **Proof System:** Rules for manipulating syntactic expressions to derive new, valid sentences from existing ones.

The process of deriving conclusions from information is formalised through inference.

Definition 2.2.3. ***Deduction***. A mode of reasoning that demonstrates a proposition logically follows from a set of premises. If the premises are true, the conclusion is guaranteed to be true.

Definition 2.2.4. ***Induction***. A mode of reasoning that infers a general principle from observing particular instances. The conclusion is probable but not guaranteed.

Definition 2.2.5. ***Inference***. The process of deriving new conclusions from existing information by deduction and induction through applying rules.

[2]For a more in-depth view, see my notes on Informal Logic, and If, Then, Set, Go (Introduction to Formal Logic and Set Theory Part 1). It might not be complete as of 25th Aug 2025; I am only one man.

Remark. Euclidean geometry is a powerful example of a formal deductive system, where all theorems are derived from a small set of axioms. The validity of Euclid's reasoning is independent of physical reality, highlighting that logic operates on form. For instance, in "All humans are mortal. All Britons are humans. Therefore, all Britons are mortal," the conclusion follows from the structure "All B are C. All A are B. Therefore, all A are C," regardless of the meaning of "human" or "mortal".

Definition 2.2.6. ***Axiom***. A statement accepted without proof, serving as a starting point for deducing other truths.

Definition 2.2.7. ***Premise***. A statement in an argument presumed to be true, from which a conclusion is drawn.

Definition 2.2.8. ***Proposition***. A declarative statement that is either true or false.

The premise that thinking is a form of logical reasoning that can be produced mechanically underpins the ambitions of Strong AI.

Definition 2.2.9. ***Weak AI***. The subfield of AI focused on creating systems that can perform a specific task which would otherwise require human intelligence. These systems simulate cognitive processes but do not possess genuine understanding or consciousness.

Definition 2.2.10. ***Strong AI (AGI)***. Artificial General Intelligence refers to a hypothetical form of AI possessing the ability to understand, learn, and apply its intelligence to solve any problem a human being can. It implies consciousness and genuine cognitive abilities.

Definition 2.2.11. ***Artificial Superintelligence (ASI)***. A hypothetical agent that would possess intelligence far surpassing that of the brightest human minds. The development of ASI is often linked to the concept of a technological singularity.

Remark. While the Turing Test remains a conceptual benchmark for AGI, it is challenged by thought experiments like the Chinese Room argument, which questions whether syntactic manipulation alone constitutes belief. A 2023 study reported that GPT-4 outperforms 99% of humans on the Torrance tests of creative thinking, suggesting movement towards what some define as emerging AGI [7].

Formal logic is not monolithic; several systems exist with varying expressive power. The choice of logic involves a trade-off between expressivity and the computational efficiency of reasoning.

Note. PL satisfiability is NP-complete; validity is coNP-complete. FOL validity is semi-decidable (recursively enumerable) and undecidable in general.

2.2.2 Propositional Logic

Propositional logic (PL) is the most fundamental system, where every proposition is true or false.

PL: Syntax and Semantics The syntax of PL defines well-formed sentences recursively. The constants `true` and `false` are sentences, as are propositional variables (e.g., P, Q). If ϕ and ψ are sentences, then so are (ϕ), $\neg\phi$ (negation), $\phi \wedge \psi$ (conjunction), $\phi \vee \psi$ (disjunction), $\phi \to \psi$ (implication), and $\phi \leftrightarrow \psi$ (biconditional). From highest to lowest, operator precedence is $\neg$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$. The implication $\phi \to \psi$ is shorthand for $\neg\phi \vee \psi$, and the biconditional $\phi \leftrightarrow \psi$ for $(\phi \to \psi) \wedge (\psi \to \phi)$. A list of key logical equivalences is provided in Table 2.2.

Table 2.2: A table of common logical equivalences.

Equivalence	**Name**
$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	Commutativity of $\wedge$
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	Commutativity of $\vee$
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	Associativity of $\wedge$
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	Associativity of $\vee$
$\neg(\neg\alpha) \equiv \alpha$	Double-negation elimination
$(\alpha \rightarrow \beta) \equiv (\neg\beta \rightarrow \neg\alpha)$	Contraposition
$(\alpha \rightarrow \beta) \equiv (\neg\alpha \vee \beta)$	Implication elimination
$(\alpha \leftrightarrow \beta) \equiv ((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha))$	Biconditional elimination
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$	De Morgan's Law
$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$	De Morgan's Law
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	Distributivity of $\wedge$ over $\vee$
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	Distributivity of $\vee$ over $\wedge$

Definition 2.2.12. ***PL: Interpretation***. An assignment of a truth value, either true (T) or false (F), to each propositional variable.

According to the standard connectives, summarised in Table 2.3, the values of its components determine the truth value of any complex sentence.

Table 2.3: Truth tables for the logical connectives.

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Definition 2.2.13. ***Valid (Tautology)***. A sentence that is true in all possible interpretations (e.g., $p \vee \neg p$). The sentence $((P \wedge Q) \rightarrow R) \leftrightarrow ((P \rightarrow R) \vee (Q \rightarrow R))$ is another example of a valid sentence, as can be verified with a truth table.

Definition 2.2.14. ***Unsatisfiable (Contradiction)***. A sentence that is false in all possible interpretations (e.g., $p \wedge \neg p$).

Definition 2.2.15. ***Satisfiable***. A true sentence under at least one interpretation. For example, $(P \wedge Q) \vee \neg Q$ is satisfiable because it is true when P and Q are both true, but false when P is false and Q is true. The problem of determining satisfiability is a classic constraint satisfaction problem.

Entailment and Proof A key task of a logical agent is to determine what follows from its knowledge base.

Definition 2.2.16. ***Model***. An interpretation that makes every sentence in a set Γ true is a model of Γ.

Definition 2.2.17. ***Entailment***. A KB entails a sentence ϕ, written KB $\vDash \phi$, if and only if ϕ is true in every model of the KB. This semantic notion ($\vDash$) is distinct from syntactic derivability ($\vdash$) within a proof system.

Entailment can be checked by enumerating all models. For instance, given a KB with sentences $\{\text{poor} \rightarrow \neg\text{worried}, \text{rich} \rightarrow \text{scared}, \neg\text{rich} \rightarrow \text{poor}\}$, one can show $\text{KB} \vDash (\text{worried} \rightarrow \text{scared})$ by constructing a truth table for all variables and verifying that the conclusion is true in every row where the KB sentences are all true. This method, called model checking, is sound and complete but intractable for many variables. The deduction theorem connects semantics to proof: $\text{KB} \vDash \phi$ if and only if the sentence $(\text{KB} \rightarrow \phi)$ is valid. A proof system offers a syntactic alternative to model checking.

Definition 2.2.18. ***Proof***. A sequence of sentences, where each sentence is either a premise from a knowledge base or is derived from previous sentences by an inference rule.

Definition 2.2.19. ***Soundness and Completeness***. A proof system is sound if it only derives entailed sentences. It is **complete** if it can derive any sentence entailed.

2.3 Uninformed Search

Goal-based rational agents employ search algorithms to find a sequence of actions, or a plan, leading from an initial state to a goal state. The process involves formulating a goal and a problem, then searching for a solution that optimises a performance measure. The agent's core execution loop is outlined by the `SIMPLE-PROBLEM-SOLVING-AGENT` procedure.

Algorithm 1: `Simple-Problem-Solving-Agent(`*percept*`)`

Persistent: *seq*, an action sequence, initially empty
Persistent: *state*, a description of the current world state
Persistent: *goal*, a goal, initially null
Persistent: *problem*, a problem formulation

```
state ← UPDATE-STATE(state, percept)
if seq is empty then
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
    if seq = failure then
        return a null action
action ← FIRST(seq)
seq ← REST(seq)
return action
```

A problem is defined by its initial state, operator set, goal test, and path cost function. For instance, finding a path from Arad to Bucharest requires formulating the goal, defining states and actions, and finding a sequence of cities that meets the goal.

2.3.1 Goal-Based Agents

In a fully observable and deterministic environment, a goal-based agent's problem is abstracted by:

- **State**: A description of the environment, e.g., `<Location,Status(A),Status(B)>`.
- **Actions**: The set of possible actions, e.g., `Left`, `Right`, `Clean`.
- **Transition Model**: A function $s' = \text{RESULT}(s, a)$.
- **Step Cost**: A function $c(s, a, s')$. The **path cost**, $g(n)$, is the sum of step costs from the start node to node n.
- **Goal State(s)**: A set of desirable states.
- **Start State**: The agent's initial state, e.g., `<RoomA,Clean,Dirty>`.

Consider a static MopBot environment with two locations, A and B, and two statuses, Clean (C) and Dirty (D). Its transition model can be represented as a state graph, as shown in Figure 2.2. Such graphs are often specified implicitly, as their explicit form can be large. The search space complexity is characterised by:

- b: the branching factor, the maximum number of successors of any node.
- d: the depth of the shallowest goal state.
- m: the maximum length of any path in the state space.

The problem can be abstracted as finding a minimum-cost path in this graph. Consider the state

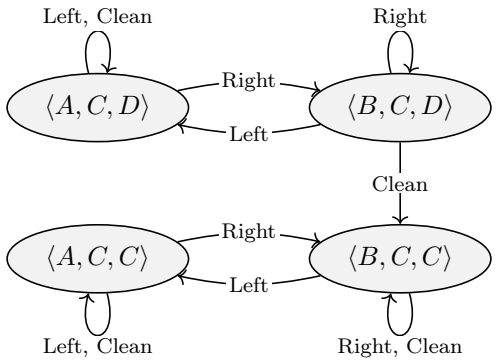


Figure 2.2: Transition model for the MopBot. A, B are locations; C, D denote Clean/Dirty.

graph in Figure 2.3, with start state S_0 and goal state S_2. Table 2.4 lists representative paths and

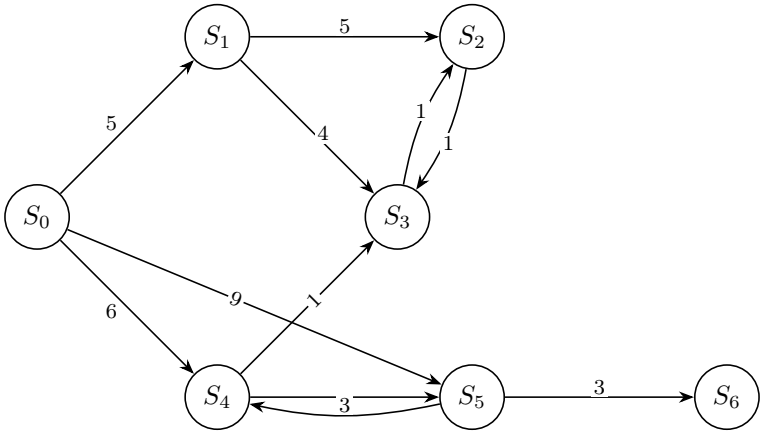


Figure 2.3: A state transition graph where vertices represent states and edges represent actions with costs.

their costs.

Table 2.4: Representative simple paths from S_0 to the goal S_2 in Figure 2.3.

Path π	# **edges**	**Total cost** $g(\pi)$	**Notes**
$\langle S_0, S_1, S_2 \rangle$	2	$5 + 5 = 10$	Shallowest; by BFS (unequal costs).
$\langle S_0, S_4, S_3, S_2 \rangle$	3	$6 + 1 + 1 = 8$	Optimal-cost path; found by UCS.
$\langle S_0, S_1, S_3, S_2 \rangle$	3	$5 + 4 + 1 = 10$	Alternative via S_3.
$\langle S_0, S_5, S_4, S_3, S_2 \rangle$	4	$9 + 3 + 1 + 1 = 14$	Longer detour via S_5.

Note. Only simple paths (no repeated states) are listed.

An action sequence to the goal can be found by converting the graph into a search tree rooted at the initial state, as shown in Figure 2.4. Each node in the tree represents a path from the start, storing data such as `(state, parent, action, g)`. A naive tree search is inefficient as it may explore the same state multiple times via different paths. Storing explored nodes in a graph search avoids redundant computations.

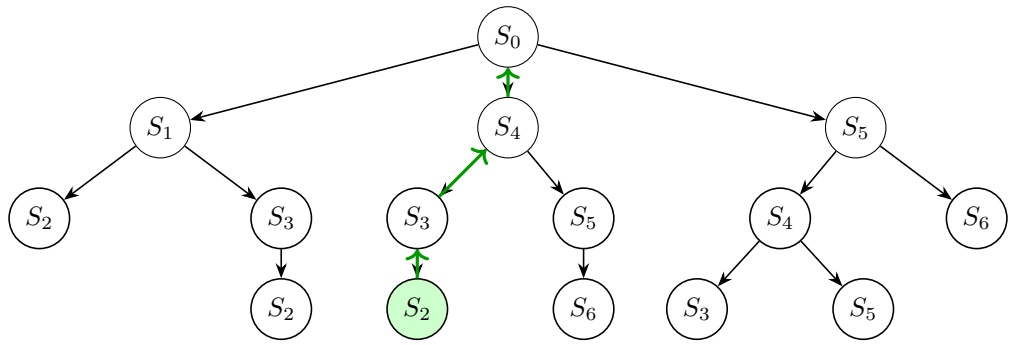


Figure 2.4: Search tree corresponding to the graph in Figure 2.3. The green arrows indicate the optimal solution path $\langle S_0, S_4, S_3, S_2 \rangle$ with cost 8.

2.3.2 Breadth-First Search

Breadth-First Search (BFS) explores all nodes at a given depth before moving to the next level. It uses a First-In-First-Out (FIFO) queue for the frontier and adds child nodes to the explored set upon generation to prevent duplicates.

Algorithm 2: Breadth-First Search

```
1  F ← Queue(start_node) // FIFO frontier
2  E ← {start_node} // Explored set (nodes discovered/enqueued)
3  while F is not empty do
4      u ← F.pop()
5      for all children v of u do
6          if GoalTest(v) then
7              return path(v)
8          if v ∉ E then
9              E.add(v)
10             F.push(v)
11 return Failure
```

Applying BFS to find a path to S_2 in Figure 2.3 results in the states shown in Table 2.5. The algorithm finds the suboptimal path $\langle S_0, S_1, S_2 \rangle$. The properties of BFS are analysed below:

Table 2.5: States of the frontier F and explored set E during BFS.

Iteration	Internal State of Data Structures
$i = 0$	$F = [S_0], E = \{S_0\}$
$i = 1$	$F = [S_1, S_4, S_5], E = \{S_0, S_1, S_4, S_5\}$
$i = 2$	Children of S_1 are explored. GoalTest(S_2) is true, returns path to S_2.

- **Completeness**: BFS is complete. If a path of length d exists, BFS will find it.
- **Optimality**: BFS is optimal only if all step costs are equal.
- **Time Complexity**: In the worst case, BFS generates all nodes up to depth d. The number of nodes is proportional to $1 + b + b^2 + \cdots + b^d$, which is $O(b^d)$, as illustrated in Figure 2.5.
- **Space Complexity**: The frontier can store all nodes at depth d, so space complexity is $O(b^d)$.

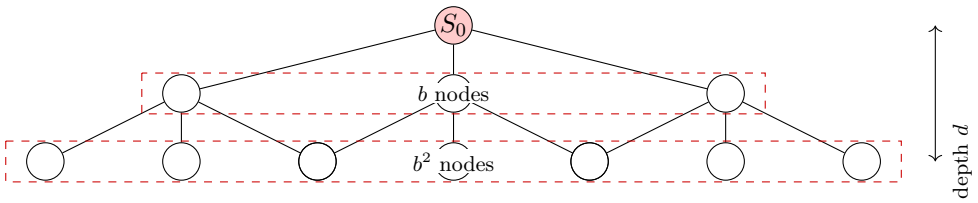


Figure 2.5: Number of nodes in a search tree with depth d and branching factor $b = 3$.

2.3.3 Uniform Cost Search

Uniform Cost Search (UCS) resolves the sub-optimality of BFS by expanding the node with the lowest path cost from the frontier. It uses a priority queue and adds a node to the explored set after it has been selected for expansion. This ensures that when a goal node is selected, the optimal path to it has been found. Tie-breaking between nodes of equal cost does not affect optimality. When all step costs are equal, UCS is equivalent to BFS.

Algorithm 3: Uniform Cost Search (UCS)

```
F ← PriorityQueue(start_node) // Lowest cost node out first
E ← ∅ // Explored set
g[start_node] ← 0
while F is not empty do
    u ← F.pop()
    if GoalTest(u) then
        return path(u)
    E.add(u)
    for all children v of u do
        if v ∉ E and v ∉ F then
            g[v] ← g[u] + c(u, v)
            F.push(v, g[v])
        else if v ∈ F and g[u] + c(u, v) < g[v] then
            g[v] ← g[u] + c(u, v)
            F.update_priority(v, g[v])
return Failure
```

The properties of UCS are as follows:

- **Completeness**: UCS is complete provided that every edge cost exceeds some small positive constant, $\epsilon > 0$.
- **Optimality**: UCS is optimal. When a node u is popped from the frontier, the algorithm has found the optimal path to u.
- **Time and Space Complexity**: Let C^* be the cost of the optimal solution and ϵ be the minimum edge cost. The complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$. When all step costs are equal, this is equivalent to $O(b^{d+1})$.

2.3.4 Depth-First Search

Depth-First Search (DFS) offers a space-efficient alternative by expanding the deepest node in the current frontier. It uses a Last-In-First-Out (LIFO) stack, exploring each branch to its conclusion before backtracking. The properties of DFS highlight a clear trade-off:

Algorithm 4: Depth-First Search (DFS)

```
1  F ← Stack(start_node) // LIFO frontier
2  E ← ∅
3  while F is not empty do
4      u ← F.pop()
5      if GoalTest(u) then
6          return path(u)
7      if u ∉ E then
8          E.add(u)
9          for all children v of u in reverse order do
10             F.push(v)
11 return Failure
```

- **Completeness**: Incomplete in infinite state spaces. Complete only for finite state spaces with no cycles.
- **Optimality**: Not optimal. It returns the first solution it finds.
- **Time Complexity**: For a graph search, $O(|V|+|E|)$. For a tree search, $O(b^m)$.
- **Space Complexity**: DFS has a significant space advantage. It only needs to store a single path from root to leaf, plus unexpanded siblings, giving a space complexity of $O(bm)$.

2.3.5 Depth-Limited and Iterative Deepening Search

The primary drawback of DFS is its failure to terminate in infinite spaces or graphs with cycles. Imposing a depth constraint leads to two related algorithms that combine the space efficiency of DFS with the completeness of BFS.

Depth-Limited Search Depth-Limited Search (DLS) imposes a hard limit on the search depth. Nodes at a specified depth limit l are treated as having no successors. This prevents DFS from exploring infinitely deep paths. The choice of l is critical: if $l < d$, DLS is incomplete; if $l \gg d$, the search is inefficient.

- **Completeness**: Complete only if $l \geq d$.
- **Optimality**: Not optimal.
- **Time Complexity**: $O(b^l)$.
- **Space Complexity**: $O(bl)$.

Iterative Deepening Search Iterative Deepening Search (IDS) resolves the problem of choosing a depth limit by trying all possible limits in increasing order. It performs a DLS for depth 0, then 1, 2, and so on, until a goal is found.

Algorithm 5: Iterative Deepening Search (IDS)

```
for depth ← 0, 1, 2, ... do
    result ← DLS(start_node, goal, depth)
    if result ≠ cutoff then
        return result
```

Although states are generated multiple times, the overhead is not costly because most nodes are in the bottom level of the tree. The exploration process is visualised in Figure 2.6.

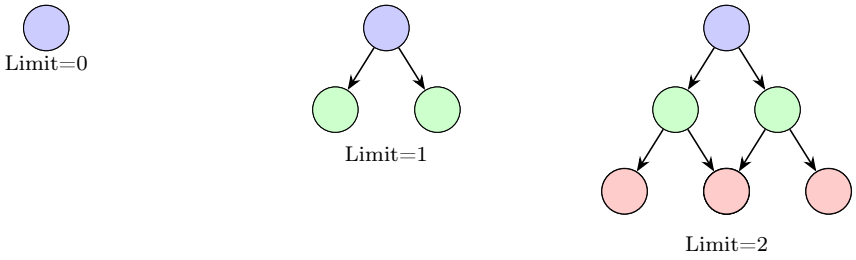


Figure 2.6: The first three iterations of an IDS. For the search to depth $d = 2$, nodes at depth zero are visited three times, nodes at depth one twice, and nodes at depth two once.

The properties of IDS combine the advantages of both BFS and DFS.

***Proposition* 2.3.1.** *IDS Completeness.* Iterative Deepening Search is complete.

Proof. If a goal state exists at a finite depth d, IDS is guaranteed to find it. The DLS call with limit $l = d$ will explore all nodes up to that depth, ensuring the shallowest goal is eventually reached. ■

***Proposition* 2.3.2.** *IDS Optimality.* IDS is optimal when all step costs are equal.

Proof. IDS explores the search tree level by level, similar to BFS. It is guaranteed to find the shallowest goal state first, which is optimal if all step costs are equal. ■

***Proposition* 2.3.3.** *IDS Complexity.* The time complexity of IDS is $O(b^d)$ and its space complexity is $O(bd)$.

Proof. The space complexity is determined by the deepest DLS call, which explores to depth d, resulting in $O(bd)$ space. For time, the total nodes generated is $N(\text{IDS}) = \sum_{i=0}^{d}(d+1-i)b^i = (d+1)b^0 + db^1 + \cdots + 1 \cdot b^d$. For a large branching factor b, this sum is dominated by the last term, $O(b^d)$. The cost of re-generating upper levels is asymptotically insignificant. ■

Due to this combination of completeness, optimality for uniform costs, and low memory requirements, IDS is often the preferred uninformed search method when the search space is large and the solution depth is unknown.

2.3.6 Advanced Topics in Uninformed Search

Bidirectional Search When the initial and goal states are known and actions are reversible, search can be performed simultaneously from both ends. This strategy runs two searches (one

forward from the start and one backward from the goal), terminating when their frontiers intersect, as shown in Figure 2.7. By meeting in the middle, each search explores to a depth of approximately $d/2$, reducing the time and space complexity exponentially from $O(b^d)$ to $O(b^{d/2})$.

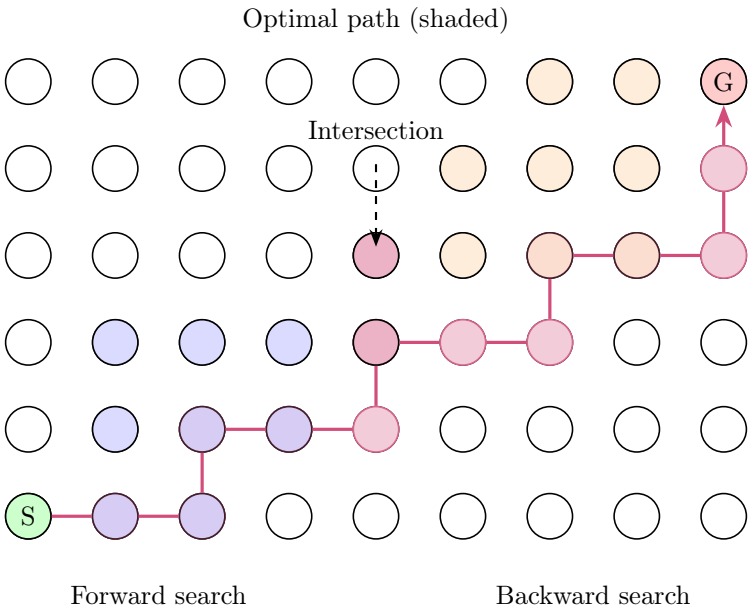


Figure 2.7: Conceptual view of bidirectional search. Two frontiers expand from the start (S) and goal (G), stopping when they meet (purple cells). The shortest path is highlighted.

Effective Branching Factor To compare the difficulty of problems empirically, the effective branching factor, b^*, provides a useful measure. It is the branching factor of a uniform tree that would contain the same number of nodes as the actual search.

Definition 2.3.1. ***Effective Branching Factor***. If a search expands N nodes to find a solution at depth d, the effective branching factor b^* is the value that solves:

$$N + 1 = \sum_{i=0}^{d} (b^*)^i = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d.$$

This concept is visualised in Figure 2.8. A value of b^* close to 1 indicates an efficient search that explores nodes mostly along a single path.

Table 2.6: Properties of uninformed search algorithms (asymptotics for tree search).

Algorithm	Complete?	Optimal?	Time	Space	Notes
BFS	Yes (finite b)	Yes if unit costs	$O(b^d)$	$O(b^d)$	FIFO queue
UCS	Yes if costs $\geq \epsilon$	Yes	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	same	PQ on g; goal on pop
DFS	No (infinite spaces)	No	$O(b^m)$	$O(bm)$	LIFO stack
DLS	If $l \geq d$	No	$O(b^l)$	$O(bl)$	depth limit l
IDS	Yes	Yes if unit costs	$O(b^d)$	$O(bd)$	DFS to increasing limits
Bidirectional	Yes (if BFS used)	Yes if unit costs	$O(b^{d/2})$	$O(b^{d/2})$	Meet in the middle

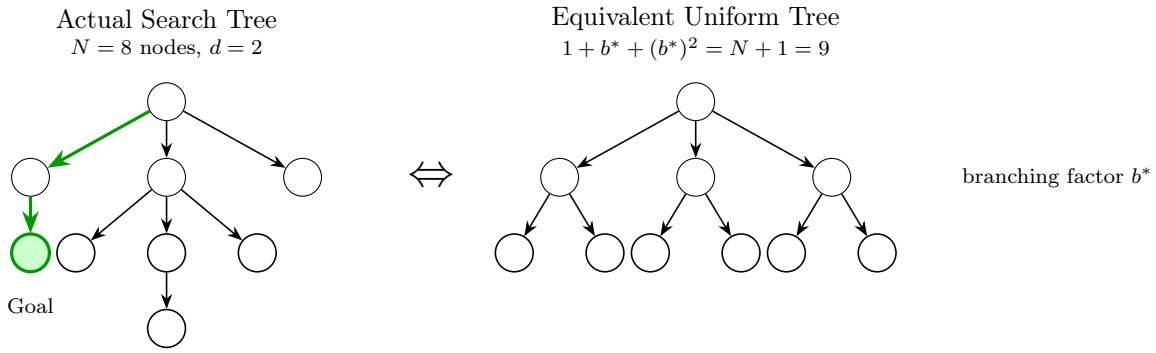


Figure 2.8: Visualisation of the effective branching factor b^*. The irregular tree on the left ($N = 8$, solution at $d = 2$) is conceptualised as the uniform tree on the right. Solving $1 + b^* + (b^*)^2 = 9$ gives $b^* \approx 2.37$.

2.4 Informed Search

Informed search algorithms use problem-specific knowledge to guide the search process, significantly reducing the number of nodes that must be expanded to find a solution. A heuristic function provides this guidance.

Definition 2.4.1. ***Heuristic Function***. A heuristic function, denoted $h(n)$, estimates the cost of the cheapest path from the state at node n to a goal state.

Heuristics are domain-specific and rely on simplified models or auxiliary information about the problem. For instance, in a route-finding problem, the straight-line distance between two cities is a simple and effective heuristic for the actual road distance.

2.4.1 Best-First Search

Best-first search is a general approach that uses an evaluation function, $f(n)$, to determine the order of node expansion. It maintains a frontier of unexpanded nodes in a priority queue, ordered by their f-values, and always selects the node with the best value to expand next. The choice of the evaluation function $f(n)$ defines the specific search strategy.

Greedy Best-First Search Greedy Best-First Search (GBFS) expands the node that appears to be closest to the goal. It evaluates nodes using only the heuristic function, meaning $f(n) = h(n)$. This strategy is "greedy" because it makes the locally optimal choice at each step, hoping this will lead to a globally optimal solution.

Algorithm 6: Greedy Best-First Search (GBFS)

```
F ← PriorityQueue(start_node) // Lowest h(n) out first
E ← ∅ // Explored set of states
while F is not empty do
    u ← F.pop()
    if GoalTest(u) then
        return path(u)
    E.add(u.state)
    for all children v of u do
        if v.state ∉ E and v ∉ F then
            F.push(v, h(v))
return Failure
```

The properties of GBFS are as follows:

- **Completeness**: GBFS is not complete. It can get stuck in loops or follow infinitely long paths that never reach the goal. A graph search version is complete in finite spaces.
- **Optimality**: It is not optimal. The first path found may be far more costly than an alternative path that initially seemed less promising.
- **Time and Space Complexity**: The worst-case complexity is $O(b^m)$, as the entire tree might be explored. With a good heuristic, however, the complexity can be dramatically reduced. It must keep all nodes in memory, so space complexity is also $O(b^m)$.

2.4.2 A* Search

A* search is the most widely used informed search algorithm. It combines the strengths of Uniform Cost Search, which prioritises low-cost paths from the start, and Greedy Best-First Search, which prioritises paths that seem close to the goal. A* evaluates nodes by combining the cost to reach the node, $g(n)$, with the estimated cost from the node to the goal, $h(n)$.

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

Here, $f(n)$ represents the estimated cost of the cheapest solution path that passes through node n. Like UCS, A* uses a priority queue for the frontier and expands the node with the lowest f-value. The algorithm is almost identical to Uniform Cost Search, with the priority queue ordered by $f(n)$ instead of $g(n)$.

Algorithm 7: A* Search

```
F ← PriorityQueue(start_node) // Lowest f(n) out first
E ← ∅ // Explored set of states
g[start_node] ← 0
while F is not empty do
    u ← F.pop()
    if GoalTest(u) then
        return path(u)
    E.add(u.state)
    for all children v of u do
        if v.state ∉ E and v ∉ F then
            g[v] ← g[u] + c(u, v)
            F.push(v, g[v] + h(v))
        else if v ∈ F and g[u] + c(u, v) < g[v] then
            g[v] ← g[u] + c(u, v)
            F.update_priority(v, g[v] + h(v))
        else if v.state ∈ E and g[u] + c(u, v) < g[v] then
            E.remove(v.state)
            // Needed for inconsistent h
            g[v] ← g[u] + c(u, v)
            F.push(v, g[v] + h(v))
return Failure
```

The behaviour and guarantees of A* search depend critically on the quality of the heuristic function $h(n)$.

Admissibility A heuristic must be optimistic to guarantee optimality.

Definition 2.4.2. ***Admissible Heuristic***. A heuristic $h(n)$ is admissible if, for every node n, it never overestimates the true cost to reach the goal. That is, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost of the optimal path from n to the goal.

For example, the straight-line distance is an admissible heuristic for road travel because the shortest path between two points is a straight line.

***Proposition* 2.4.1.** *A* Tree Search Optimality.* If $h(n)$ is an admissible heuristic, A* search using a tree-search formulation is optimal.

Consistency For graph search, a stronger condition is required to ensure optimality.

Definition 2.4.3. ***Consistent Heuristic***. A heuristic $h(n)$ is consistent (or monotonic) if, for every node n and every successor n' generated by an action a, the estimated cost of reaching the goal from n is no greater than the step cost to n' plus the estimated cost from n'. That is,

$$h(n) \leq c(n, a, n') + h(n').$$

This is a form of the triangle inequality. A consistent heuristic implies that the f-values along any path are non-decreasing: $f(n') \geq f(n)$. If a heuristic is consistent, it is also admissible, assuming $h(\text{goal}) = 0$.

***Proposition* 2.4.2.** *A* Graph Search Optimality.* If $h(n)$ is a consistent heuristic, A* search using a graph-search formulation is optimal.

Proof. With a consistent heuristic, the sequence of f-values of the nodes expanded by A* is non-decreasing. Consequently, the first time A* selects a goal node for expansion, it is guaranteed to have found an optimal path. Any other path to that goal must pass through a frontier node with an f-value at least as high. If the heuristic is only admissible but not consistent, optimality requires that A* re-opens nodes from the explored set if a cheaper path to them is found, as included in A* Search. ■

The properties of A* search can be summarised as:

- **Completeness**: A* is complete, provided there are finitely many nodes with $f(n) \leq f(G)$, where G is a goal node.
- **Optimality**: A* is optimal if the heuristic is admissible (for tree search) or consistent (for graph search).
- **Time and Space Complexity**: The complexity is exponential in the worst case, but depends on the heuristic's quality. A* must store all generated nodes, so its space complexity is a significant drawback.

A* is optimally efficient, meaning no other optimal algorithm using the same heuristic is guaranteed to expand fewer nodes. The search process of A* can be visualised as expanding "f-contours" of nodes with equal f-cost, as shown in Figure 2.9. Compared to the circular contours of UCS, A*'s contours are elongated towards the goal, focusing the search more effectively.

2.4.3 Heuristic Design

The performance of A* depends heavily on the chosen heuristic. An ideal heuristic would be the optimal cost, $h^*(n)$, but this is unavailable. The goal is to design a heuristic $h(n)$ that is as close to $h^*(n)$ as possible without ever exceeding it.

Relaxed Problems A robust method for generating admissible heuristics is to consider a relaxed version of the problem—a problem with fewer restrictions on actions. The cost of an optimal solution

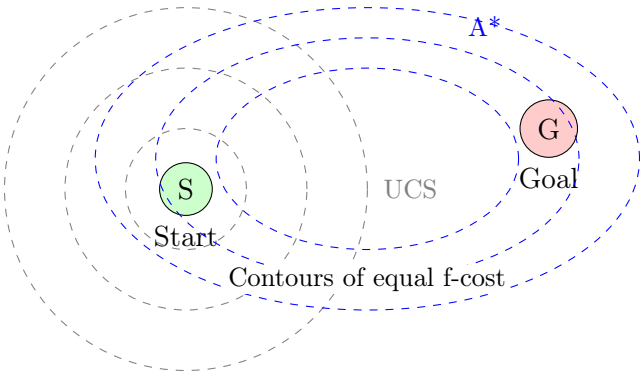


Figure 2.9: Conceptual illustration of search contours. UCS expands nodes in uniform circles of path cost $g(n)$ from the start. A* expands nodes in elliptical contours of estimated total cost $f(n) = g(n) + h(n)$, focusing the search towards the goal.

in the relaxed problem is an admissible heuristic for the original problem because any solution in the original problem is also a solution in the relaxed problem and is therefore at least as expensive.

For example, consider the 8-puzzle, a sliding tile puzzle aiming to arrange tiles numerically.

- A relaxed rule could be "a tile can move from any square to any other square". The number of misplaced tiles is the number of moves to solve this relaxed problem. This gives the heuristic $h_1(n) =$ number of misplaced tiles.
- A slightly less relaxed rule could be "a tile can move to any adjacent square, even if occupied". The number of moves to solve this is the sum of the Manhattan distances of each tile from its goal position. This gives $h_2(n) = \sum_{i=1}^{8} \text{ManhattanDistance}(i)$.

Both h_1 and h_2 are admissible.

Dominance When multiple admissible heuristics are available, one is often better than another.

Definition 2.4.4. ***Heuristic Dominance***. If $h_2(n)$ and $h_1(n)$ are two admissible heuristics, h_2 dominates h_1 if $h_2(n) \geq h_1(n)$ for all nodes n.

An algorithm using a dominant heuristic will expand no more nodes than one that does not. For the 8-puzzle, illustrated in Figure 2.10, the Manhattan distance heuristic, h_2, dominates the misplaced tiles heuristic, h_1. Using h_2 results in A* expanding significantly fewer nodes than h_1. The trade-off is that a more accurate heuristic may be more expensive to compute.

2.4.4 Memory-Bounded Heuristic Search

Memory-bounded algorithms address the space complexity drawback of A* by adapting heuristic search to use only a limited amount of memory.

Iterative-Deepening A* (IDA*) IDA* is a direct adaptation of IDS to heuristic search. Instead of a depth limit, IDA* uses an f-cost limit. It begins with a limit equal to the heuristic of the start state, $h(S_0)$. Each iteration performs a depth-first search that prunes any path as soon as its f-cost, $g(n) + h(n)$, exceeds the current limit. If no solution is found, the new limit is set to the lowest

Start State

1	2	3
8		4
7	6	5

Goal State

1	2	3
4	5	6
7	8	

$h_1(n) = 4$ (misplaced: 4, 5, 6, 8)
$h_2(n) = 2 + 2 + 2 + 2 = 8$ (Manhattan)

Figure 2.10: Heuristics for the 8-puzzle. The number of misplaced tiles (h_1) and the sum of Manhattan distances (h_2) are admissible. Since $h_2(n) \geq h_1(n)$ for all n, h_2 dominates h_1.

f-cost that exceeded the old limit. This process, outlined in and visualised in Figure 2.11, repeats until a goal is found. IDA* is complete and optimal (with an admissible heuristic) but has the low space complexity of DFS, $O(bd)$.

Algorithm 8: Iterative-Deepening A* (IDA*)

```
limit ← h(start_node)
while true do
    result, limit ← SEARCH(start_node, 0, limit)
    if result is a solution then
        return result
    if limit = ∞ then
        return Failure

Function SEARCH(node, g, limit):
    f ← g + h(node)
    if f > limit then
        return null, f
    if GoalTest(node) then
        return node, limit
    min_cutoff ← ∞
    for all children v of node do
        res, new_limit ← SEARCH(v, g + c(node, v), limit)
        if res is a solution then
            return res, limit
        min_cutoff ← min(min_cutoff, new_limit)
    return null, min_cutoff
```

Recursive Best-First Search (RBFS) RBFS, shown in , is a recursive algorithm that aims to simulate standard best-first search using only linear space. As it expands nodes down a path, it keeps track of the f-value of the best alternative path available from any ancestor of the current node (the f_{limit}). If the current node's f-value exceeds this limit, the recursion unwinds to the alternative path. As it unwinds, RBFS replaces the f-value of each node with the best f-value of its children. This "backed-up" value allows the search to remember the quality of paths in subtrees

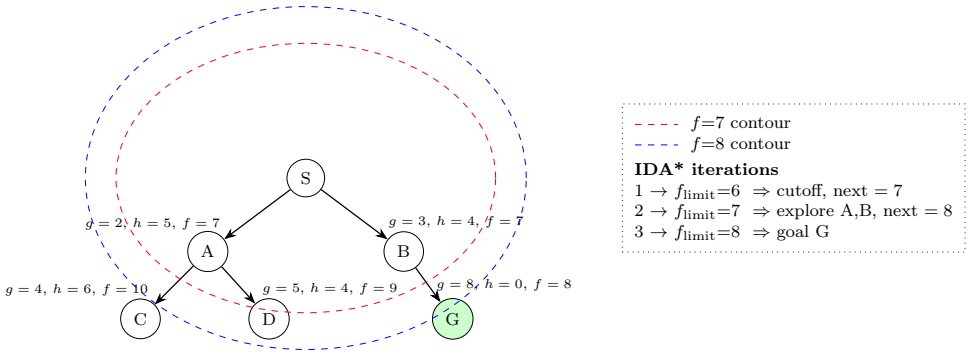


Figure 2.11: IDA* search expanding successive f-contours. The search deepens to the current f-limit; pruned branches set the next limit.

it has explored and potentially re-explore them later if they become promising again.

Algorithm 9: Recursive Best-First Search (RBFS)

```
Function RBFS(problem, node, f_limit):
    if GoalTest(node) then
        return node, 0
    successors ← Expand(node)
    if successors is empty then
        return Failure, ∞
    for each s in successors do
        s.f ← max(g(s) + h(s), node.f)
    while true do
        Sort successors by f-value
        best ← first element of successors
        if best.f > f_limit then
            return Failure, best.f
        alternative ← second element's f-value
        result, best.f ← RBFS(problem, best, min(f_limit, alternative))
        if result ≠ Failure then
            return result, 0
```

RBFS is more efficient than IDA* as it suffers less from redundant node generations, but it can involve excessive node re-generation if f-values are unstable. The properties of the discussed informed search algorithms are summarised in Table 2.7.

Table 2.7: Properties of informed search algorithms (asymptotics for tree search).

Algorithm	Complete?	Optimal?	Time	Space	Notes
GBFS	No (tree); Yes (graph)	No	$O(b^m)$	$O(b^m)$	PQ on $h(n)$
A*	Yes	Yes (with conds.)	Exponential	Exponential	PQ on $f(n) = g(n) + h(n)$
IDA*	Yes	Yes (with conds.)	Exponential	$O(bd)$	f-cost limited DFS
RBFS	Yes	Yes (with conds.)	Exponential	$O(bd)$	Simulates A* in linear space

2.5 Local Search

For many optimisation problems, such as the n-queens problem or circuit layout, the path to the solution is irrelevant; only the final configuration, or state, matters. In these cases, local search algorithms offer an effective alternative to the path-finding methods previously discussed. Local search operates on a single current state, iteratively moving to a neighbouring state to improve an objective function. These algorithms are not systematic and may not find a solution, but they are often highly efficient in space, typically using $O(1)$ memory.

2.5.1 Hill-Climbing Search

Hill-climbing is a simple greedy local search algorithm, detailed in Hill-Climbing Search. It starts with a random initial state and continuously moves toward increasing the objective function's value. It terminates when it reaches a "peak" where no neighbour has a higher value. Hill-climbing is simple and fast, but its major drawback is that it can get stuck in local maxima, failing to find the global maximum, as illustrated in Figure 2.12. Variants like random-restart hill-climbing can mitigate this issue by conducting multiple searches from different random initial states.

Algorithm 10: `Hill-Climbing`(*problem*)

Input: *problem*, a problem description

```
1 current ← problem.InitialState
2 while true do
3     neighbour ← Highest-Valued-Successor(current)
4     if Value(neighbour) ≤ Value(current) then
5         return current
6     current ← neighbour
```

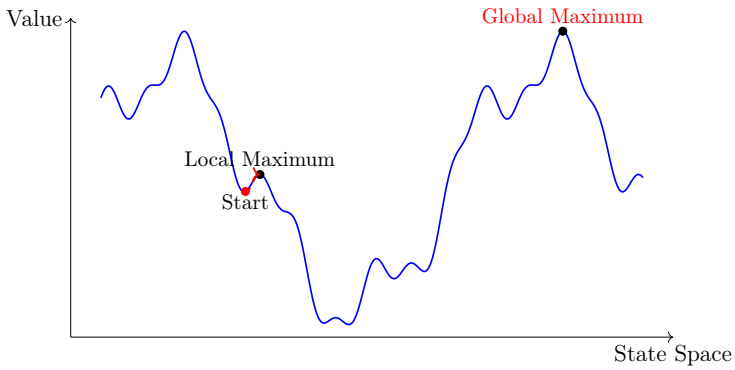


Figure 2.12: The problem of local maxima in hill-climbing. A search starting at the indicated point will follow the gradient upwards and become trapped at the local maximum, failing to discover the global maximum.

2.5.2 Simulated Annealing

Simulated annealing is a more sophisticated local search algorithm designed to escape local maxima. It borrows its central idea from the metallurgical process of annealing, where a material is heated and then slowly cooled to settle into a low-energy crystalline state. The algorithm, shown in Simulated Annealing, allows for "bad" moves (moves to states with lower values), with a probability that

decreases over time. This probability is governed by a temperature parameter, T, which starts high and gradually lowers according to a schedule. When T is high, the algorithm explores the state space widely. As T approaches zero, the algorithm becomes increasingly greedy, converging towards a local, and hopefully global, optimum. Simulated annealing is guaranteed to find the global optimum if the temperature is lowered sufficiently slowly.

Algorithm 11: `Simulated-Annealing`(*problem*, *schedule*)

Input: *problem*, a problem description
Input: *schedule*, a mapping from time to temperature

```
current ← problem.InitialState
for t ← 1 to ∞ do
    T ← schedule(t)
    if T = 0 then
        return current
    next ← Randomly-Selected-Successor(current)
    ΔE ← Value(next) − Value(current)
    if ΔE > 0 then
        current ← next
    else
        current ← next with probability e^(ΔE/T)
```

2.5.3 Local Beam Search

Local beam search keeps track of k states rather than just one. It begins with k randomly generated states. At each step, it generates all successors of all k states. If any successor is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats. This approach, outlined in Local Beam Search, differs from running k independent hill-climbing searches because information is shared among the parallel search threads. A successful search can discover a promising region and colonise the other searches to focus on that area. A stochastic variant adds random successors to maintain diversity.

Algorithm 12: `Local-Beam-Search`(*problem*, k)

Input: *problem*, a problem description
Input: k, the number of states to maintain

```
beam ← Generate k initial states
while true do
    successors ← ∅
    for each state in beam do
        if GoalTest(state) then
            return state
        successors ← successors ∪ Successors(state)
    beam ← Select k best from successors
    if beam has no better states than previous beam then
        return Best state seen
```

2.5.4 Genetic Algorithms

A genetic algorithm (GA) is a variant of local beam search that models the process of natural evolution. A GA maintains a population of candidate states, called individuals. An objective function evaluates the fitness of each individual. The population evolves over successive generations through three main operators:

- **Selection**: Fitter individuals are more likely to be chosen to reproduce.
- **Crossover**: Offspring are created by combining two parents' genetic material (state representation).
- **Mutation**: Random modifications are introduced to the offspring to maintain genetic diversity.

The process, outlined in Genetic Algorithms, is well-suited for large or complex state spaces where gradient information is unavailable.

Algorithm 13: `Genetic-Algorithm`($population, fitness\_fn$)

Input: *population*, a set of individuals
Input: $fitness\_fn$, a function that measures an individual's fitness

```
1  while true do
2      new_population ← ∅
3      for i ← 1 to size(population) do
4          x ← Selection(population, fitness_fn)
5          y ← Selection(population, fitness_fn)
6          child ← Crossover(x, y)
7          if small random probability then
8              child ← Mutate(child)
9          new_population.add(child)
10     population ← new_population
11     if an individual is fit enough or max generations reached then
12         return Best individual
```

The properties of these local search algorithms are summarised in Table 2.8.

Table 2.8: Properties of local search algorithms.

Algorithm	Memory	Key Idea / Notes
Hill-Climbing	$O(1)$	Greedy ascent; gets stuck in local optima.
Simulated Annealing	$O(1)$	Allows downward moves to escape local optima via a temperature schedule.
Local Beam Search	$O(k)$	Keeps k states; parallel search threads share information.
Genetic Algorithm	$O(k)$	Population-based search using selection, crossover, and mutation.

2.6 Constraint Satisfaction Problems

Continuing with problems where the solution is a state rather than a path, Constraint Satisfaction Problems (CSPs) provide a robust and standardised framework. A CSP is defined by a set of variables, each with a domain of possible values, and a set of constraints that specify allowable combinations of values for subsets of variables [5].

Formally, a CSP consists of three components:

- A set of variables, $X = \{X_1, X_2, \ldots, X_n\}$.
- A set of domains, $D = \{D_1, D_2, \ldots, D_n\}$, where D_i is the domain of variable X_i.
- A set of constraints, $C = \{C_1, C_2, \ldots, C_m\}$, where each constraint C_j involves some subset of the variables and specifies the allowable combinations of values for that subset.

A solution to a CSP is a complete and consistent assignment of values to all variables. An example is the map-colouring problem shown in Figure 2.13, where the task is to assign a colour to each region such that no two adjacent regions have the same colour.

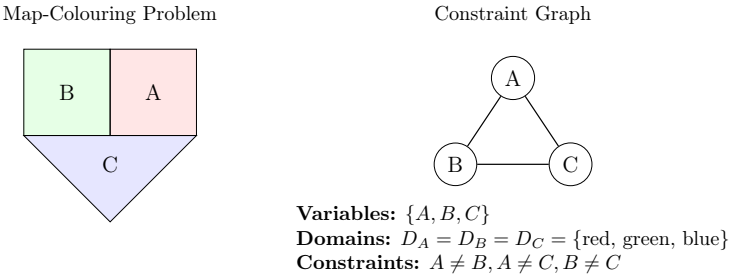


Figure 2.13: A map-colouring problem represented as a Constraint Satisfaction Problem. Regions become variables, and adjacency relationships become inequality constraints.

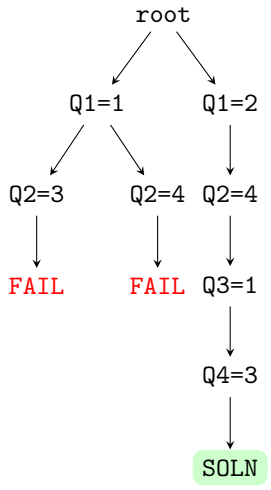
2.6.1 Backtracking Search for CSPs

CSPs can be solved using a specialised depth-first search called backtracking search.

Algorithm 14: `Backtracking-Search`(*csp*)

```
1  Function Recursive-Backtrack(assignment, csp):
2      if assignment is complete then
3          return assignment
4      var ← Select-Unassigned-Variable(csp)
5      for each value in Order-Domain-Values(var, assignment, csp) do
6          if value is consistent with assignment then
7              Add {var = value} to assignment
8              result ← Recursive-Backtrack(assignment, csp)
9              if result ≠ failure then
10                 return result
11             Remove {var = value} from assignment
12     return failure
13 return Recursive-Backtrack (∅, csp)
```

The algorithm, shown in Backtracking Search for CSPs, incrementally assigns values to variables individually. If an assignment violates a constraint, the algorithm backtracks to the preceding variable and tries a different value. This avoids the inefficiency of generating complete assignments and then testing them. A trace of backtracking search with forward checking for the 4-queens problem, a classic CSP, is shown in Figure 2.14.



Trace of assignments and domain changes:
1. **Q1=1**. FC prunes Q2:{1,2}, Q3:{1,3}, Q4:{1,4}.
Remaining: $D_{Q2} = \{3, 4\}, D_{Q3} = \{2, 4\}, D_{Q4} = \{2, 3\}$.
2. **Q2=3**. FC prunes Q3:{3,4}, Q4:{2,4}.
Remaining: $D_{Q3} = \{2\}, D_{Q4} = \{\}$. Empty domain. **FAIL**.
3. Backtrack. **Q2=4**. FC prunes Q3:{2,3}, Q4:{3,4}.
Remaining: $D_{Q3} = \{2\}, D_{Q4} = \{3\}$.
4. **Q3=2**. FC prunes Q4:{1,3}.
Remaining: $D_{Q4} = \{\}$. Empty domain. **FAIL**.
5. Backtrack to Q1. **Q1=2**. FC prunes domains.
Remaining: $D_{Q2} = \{4\}, D_{Q3} = \{1, 3\}, D_{Q4} = \{1, 3, 4\}$.
6. **Q2=4**. FC prunes domains.
Remaining: $D_{Q3} = \{1\}, D_{Q4} = \{3\}$.
7. **Q3=1**. FC prunes domains.
Remaining: $D_{Q4} = \{3\}$.
8. **Q4=3**. Assignment is complete. **SOLUTION**.

Figure 2.14: A partial search tree and trace for the 4-queens problem using backtracking with forward checking. The search finds the solution $\langle 2, 4, 1, 3 \rangle$.

2.6.2 Improving Backtracking Efficiency

The performance of standard backtracking can be dramatically improved with several techniques that prune the search space more effectively.

Variable and Value Ordering The order in which variables are chosen and values are tried can substantially impact performance.

- **Minimum Remaining Values (MRV):** This heuristic selects the variable with the fewest legal values remaining in its domain. This "fail-fast" strategy quickly prunes large parts of the search tree by identifying inevitable failures early.
- **Degree Heuristic:** As a tie-breaker for MRV, this heuristic selects the variable involved in the largest number of constraints on other unassigned variables.
- **Least Constraining Value (LCV):** When selecting a value for the current variable, this heuristic prefers the value that rules out the fewest choices for the neighbouring variables in the constraint graph. It leaves maximal flexibility for subsequent assignments.

Inference and Constraint Propagation Beyond simple checking, we can infer the consequences of an assignment. Constraint propagation repeatedly enforces local consistency conditions to prune the domains of variables.

- **Forward Checking:** When a value is assigned to a variable X_i, forward checking examines each unassigned variable X_j that is connected to X_i by a constraint. It deletes any value from X_j's domain inconsistent with the new assignment to X_i.
- **Arc Consistency:** A more powerful form of propagation. An arc from X_i to X_j is arc-consistent if for every value in the domain of X_i, there is some value in the domain of X_j that satisfies the binary constraint between them. The AC-3 algorithm is commonly used to enforce arc consistency across the entire CSP, either as a pre-processing step or interleaved with the search process. A trace of AC-3 reducing domains is shown in Table 2.9.

Table 2.9: Trace of the AC-3 algorithm for a simple CSP. Variables A, B, C have initial domain $\{0, 1, 2, 3, 4\}$. Constraints are $A = B + 1$ and $B = 2C$. The initial queue is $\{(A, B), (B, A), (B, C), (C, B)\}$.

Arc Processed	Domains After Processing	Queue	Notes
-	$D_A = \{0..4\}, D_B = \{0..4\}, D_C = \{0..4\}$	$\langle (A,B), (B,A), (B,C), (C,B) \rangle$	Initial state
(A, B)	$D_A = \{1, 2, 3, 4\}$	$\langle (B,A), (B,C), (C,B) \rangle$	$A = 0$ removed (no $B = -1$)
(B, A)	$D_B = \{0, 1, 2, 3\}$	$\langle (B,C), (C,B), (C,B) \rangle$	$B = 4$ removed (no $A = 5$)
(B, C)	$D_B = \{0, 2\}$	$\langle (C,B), (A,B) \rangle$	$B = 1, 3$ removed (not even). Add (A, B).
(C, B)	$D_C = \{0, 1\}$	$\langle (A,B) \rangle$	$C = 2, 3, 4$ removed (no $B = 4, 6, 8$).
(A, B)	$D_A = \{1, 3\}$	$\langle (C,A) \rangle$	$A = 2, 4$ removed (no $B = 1, 3$). Add (C, A).

Note. The final queue is empty after (C, A) is processed with no domain changes. Final domains: $D_A = \{1, 3\}, D_B = \{0, 2\}, D_C = \{0, 1\}$.

The effect of these heuristics is demonstrated in Figure 2.15, which traces the initial steps of solving the map-colouring problem for Australia. Initially, all states have a domain of $\{R, G, B\}$. SA is chosen first based on the degree of heuristic. After assigning SA=Red, forward checking reduces the domains of its neighbours—the MRV heuristic guides subsequent choices, leading to a solution without backtracking.

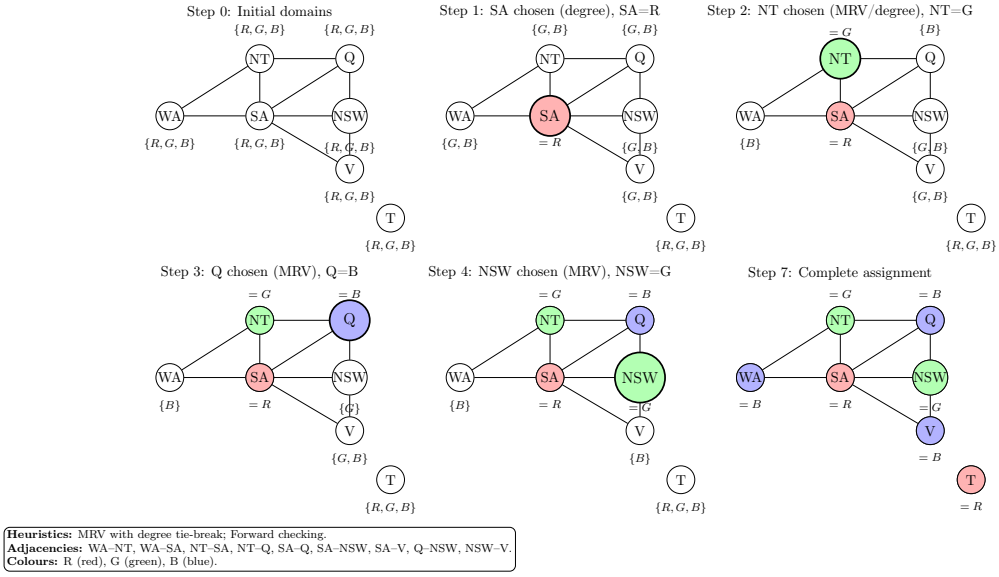


Figure 2.15: Trace of backtracking search with forward checking and MRV (degree tie-break) on the Australia map-colouring CSP with colours $\{R, G, B\}$. The search proceeds SA→NT→Q→NSW, then finishes without backtracking: WA= B, NT= G, SA= R, Q= B, NSW= G, V= B, T= R.

2.6.3 The Structure of Problems

The complexity of solving a CSP is often related to the structure of its constraint graph. For instance, if the constraint graph is a tree (i.e., contains no cycles), the CSP can be solved efficiently in a linear time manner in the number of variables. Problems with nearly tree-structured graphs can be solved using conditioning, where a small set of variables (a cycle cutset) is instantiated, and the remaining problem is solved as a tree.

The key techniques for enhancing backtracking search are summarised in Table 2.10.

Table 2.10: Summary of techniques for improving CSP backtracking search.

Technique	**Purpose**	**Description**
MRV Heuristic	Variable Ordering	Choose variable with the fewest remaining legal values.
Degree Heuristic	Variable Ordering	Tie-breaker: choose the variable with the most constraints on other variables.
LCV Heuristic	Value Ordering	Choose value that prunes the fewest values from neighbours' domains.
Forward Checking	Inference	After assigning to X_i, prune inconsistent values from its neighbours.
Arc Consistency	Inference	Enforce that for each value of X_i, a consistent value exists for X_j.

2.7 Adversarial Search

Whereas previous search methods operate in single-agent environments, adversarial search addresses multi-agent environments with conflicting goals. Such scenarios are modelled as games where an agent's success depends on the actions of its opponents.

A game can be formally defined by the following components:

- **Initial State**: The starting configuration of the game, S_0.
- **Players**: A function that defines which player has the move in a given state.
- **Actions**: A function that returns the set of legal moves in a state, ACTIONS(S).
- **Transition Model**: A function that defines the resulting state after a move, RESULT(S, a).
- **Terminal Test**: A function that determines if the game is over, TERMINAL(S).
- **Utility Function**: A function that assigns a numerical value to a terminal state for a given player, UTILITY(S).

This section focuses on two-player, deterministic, zero-sum games, such as chess or Go, where one player's gain is exactly the other player's loss.

2.7.1 The Minimax Algorithm

The minimax algorithm provides a strategy for finding the optimal move in a game by assuming the opponent will also play optimally. The two players are designated MAX, who seeks to maximise the utility, and MIN, who seeks to minimise it. The algorithm performs a complete depth-first exploration of the game tree. At each level, values propagate up from the terminal states: MAX nodes choose the maximum value from their children, and MIN nodes choose the minimum. This process is illustrated in Figure 2.16.

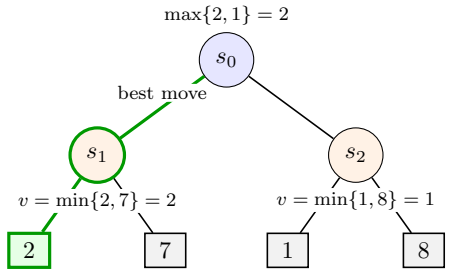


Figure 2.16: A simple two-player game tree. MAX chooses a path (s_1 or s_2), then MIN chooses a terminal state. The values are propagated up the tree: MIN nodes take the minimum of their children's values, and MAX nodes take the maximum. MAX's optimal move at s_0 is to move to s_1, guaranteeing a utility of at least 2.

The minimax value for a state s can be defined recursively:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL}(s) \\ \max_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if player is MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if player is MIN} \end{cases}$$

The The Minimax Algorithm procedure formalises this strategy. Its properties are:

- **Completeness**: If the game tree is finite.
- **Optimality**: Yes, against an optimal opponent.

- **Time Complexity**: $O(b^m)$, where b is the branching factor and m is the maximum depth.
- **Space Complexity**: $O(bm)$ for a depth-first implementation.

The exponential time complexity makes minimax impractical for non-trivial games.

Algorithm 15: `Minimax-Decision(`*state*`)`

```
return arg max_{a∈ACTIONS(state)} Min-Value(RESULT(state, a))
Function Max-Value(state):
    if TERMINAL(state) then
        return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← max(v, Min-Value(RESULT(state, a)))
    return v
Function Min-Value(state):
    if TERMINAL(state) then
        return UTILITY(state)
    v ← ∞
    for each a in ACTIONS(state) do
        v ← min(v, Max-Value(RESULT(state, a)))
    return v
```

2.7.2 Alpha-Beta Pruning

The performance of minimax can be drastically improved by alpha-beta pruning, a technique that eliminates large parts of the search tree that cannot influence the final decision. It maintains two values during the search:

- α: The best value (highest score) found for MAX along the path to the root.
- β: The best value (lowest score) found so far for MIN along the path to the root.

The search can be pruned below any MIN node once a value is found that is less than or equal to α, and below any MAX node once a value is found that is greater than or equal to β. An example of this pruning process is shown in Figure 2.17, and the algorithm is formalised in Alpha-Beta Pruning.

With optimal move ordering, where the best moves are explored first, alpha-beta pruning reduces the effective branching factor from b to approximately $\sqrt{b}$. This allows the search to reach about twice as deep in the same amount of time. The time complexity in the best case is $O(b^{m/2})$, while in the worst case it is the same as minimax, $O(b^m)$. If moves are ordered randomly, the average complexity is about $O(b^{3m/4})$. The space complexity remains $O(bm)$.

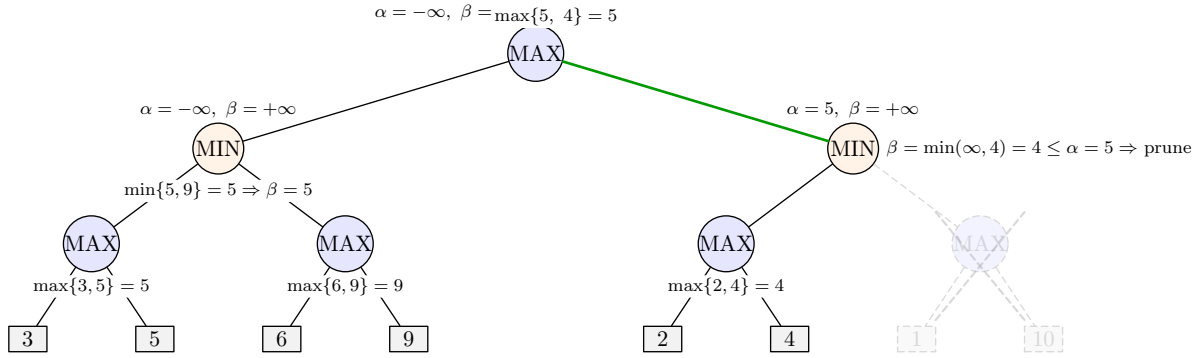


Figure 2.17: Alpha–beta pruning with left-to-right evaluation. After the left subtree returns 5, the right MIN node inherits $\alpha = 5$. Its first child returns 4, setting $\beta = 4$. Since $\beta \leq \alpha$, the remaining sibling subtree is pruned without evaluation.

Algorithm 16: Alpha-Beta Search

Function `Max-Value`$(state, \alpha, \beta)$**:**
  if *TERMINAL(state)* **then**
    return *UTILITY(state)*
  $v \leftarrow -\infty$
  for *each a in ACTIONS(state)* **do**
    $v \leftarrow \max(v,$ `Min-Value`$(\text{RESULT}(state, a), \alpha, \beta))$
    if $v \geq \beta$ **then**
      return v
    // Prune branch
    $\alpha \leftarrow \max(\alpha, v)$
  return v

Function `Min-Value`$(state, \alpha, \beta)$**:**
  if *TERMINAL(state)* **then**
    return *UTILITY(state)*
  $v \leftarrow \infty$
  for *each a in ACTIONS(state)* **do**
    $v \leftarrow \min(v,$ `Max-Value`$(\text{RESULT}(state, a), \alpha, \beta))$
    if $v \leq \alpha$ **then**
      return v
    // Prune branch
    $\beta \leftarrow \min(\beta, v)$
  return v

2.7.3 Imperfect Real-Time Decisions

The search space for complex games like chess is too ample to explore fully, even with alpha-beta pruning. In practice, algorithms must make decisions under time constraints by cutting off the search at a predetermined depth limit, d. At this cutoff, the utility function is replaced by a heuristic evaluation function, EVAL(S), which estimates the position's utility. Non-terminal nodes at the depth limit are treated as leaves.

A good heuristic evaluation function should have the following properties:

- It should order terminal states similarly to the true utility function.
- Its computation should be efficient.
- For non-terminal states, it should be highly correlated with the chances of winning.

For example, a common evaluation function in chess is a weighted linear function of features:

$$\text{EVAL}(S) = w_1 f_1(S) + w_2 f_2(S) + \cdots + w_n f_n(S)$$

where w_i are weights and $f_i(S)$ are features of the game state, such as material advantage or piece mobility. The alpha-beta algorithm is modified to use a cutoff test and the evaluation function, as shown in Imperfect Real-Time Decisions. With a fixed depth limit d, the time complexity becomes $O(b^{d/2})$ in the best case.

Algorithm 17: Heuristic Alpha-Beta Search

```
Function H-Max-Value(state, α, β, depth):
    if CUTOFF-TEST(state, depth) then
        return EVAL(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← max(v, H-Min-Value(RESULT(state, a), α, β, depth + 1))
        if v ≥ β then
            return v
        α ← max(α, v)
    return v
Function H-Min-Value(state, α, β, depth):
    if CUTOFF-TEST(state, depth) then
        return EVAL(state)
    v ← ∞
    for each a in ACTIONS(state) do
        v ← min(v, H-Max-Value(RESULT(state, a), α, β, depth + 1))
        if v ≤ α then
            return v
        β ← min(β, v)
    return v
```

The properties of adversarial search algorithms are summarised in Table 2.11.

Table 2.11: Properties of adversarial search algorithms.

Algorithm	Time	Space	Notes
Minimax	$O(b^m)$	$O(bm)$	Optimal but impractical for large games.
Alpha-Beta Pruning	$O(b^{m/2})$	$O(bm)$	Optimal, same result as minimax. Time is the best-case.
Heuristic Alpha-Beta	$O(b^{d/2})$	$O(bd)$	Suboptimal; uses depth limit d and heuristics.

2.8 FLAI 2 - Inference in Propositional Logic

Many reasoning tasks can be cast as determining the satisfiability of a logical sentence, driving the development of highly optimised algorithms known as SAT solvers. These algorithms typically operate on sentences in Conjunctive Normal Form (CNF): a conjunction of clauses, where each clause is a disjunction of literals. Any sentence can be converted to CNF using the logical equivalences in Table 2.2. For example, the sentence $(\text{sat} \vee \text{sun}) \rightarrow (\text{free} \rightarrow \text{concert})$ can be converted as follows:

$$
\begin{aligned}
(\text{sat} \vee \text{sun}) &\rightarrow (\neg\text{free} \vee \text{concert}) && \text{Implication elimination} \\
\neg(\text{sat} \vee \text{sun}) &\vee (\neg\text{free} \vee \text{concert}) && \text{Implication elimination} \\
(\neg\text{sat} \wedge \neg\text{sun}) &\vee (\neg\text{free} \vee \text{concert}) && \text{De Morgan's Law} \\
(\neg\text{sat} \vee \neg\text{free} \vee \text{concert}) &\wedge (\neg\text{sun} \vee \neg\text{free} \vee \text{concert}) && \text{Distributivity}
\end{aligned}
$$

The final sentence is in CNF. Since each clause has at most one positive literal, it is also in Horn form.

The DPLL Algorithm The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is a complete, backtracking-based search for deciding the satisfiability of PL formulae in CNF. It improves upon naive truth-table enumeration through several heuristics that prune the search space, as detailed in .

Algorithm 18: The Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

```
1  Function DPLL(φ):
2      if φ is empty (all clauses satisfied) then
3          return true
4      if φ contains an empty clause then
5          return false
6      if a pure literal l exists in φ then
7          return DPLL (Simplify(φ, l))
8      else if a unit clause {l} exists in φ then
9          return DPLL (Simplify(φ, l))
10     else
11         l ← Choose-Literal(φ)
12         if DPLL (Simplify(φ, l)) then
13             return true
14         else
15             return DPLL (Simplify(φ, ¬l))
```

- **Early Termination.** The search terminates if a partial assignment makes a clause false (failure) or if all clauses are satisfied (success).
- **Pure Symbol Heuristic.** A propositional variable that appears with only one polarity (always positive or always negative) can be assigned a value that satisfies all clauses containing it.
- **Unit Clause Heuristic.** A clause with only one unassigned literal forces an assignment. This must be satisfied, so the variable is assigned accordingly.

These heuristics have direct parallels in solving general CSPs. The unit clause heuristic is a form

of forward checking related to the MRV heuristic.

Stochastic Local Search For huge problems, incomplete local search algorithms like WalkSAT can be more practical. WalkSAT, detailed in , begins with a random assignment of variables and iteratively flips the assignment of a variable within an unsatisfied clause. Choosing which variable to flip mixes a greedy strategy (minimising the remaining unsatisfied clauses) with a random choice to escape local minima.

Algorithm 19: `WalkSAT`($clauses, p, max\_flips$)

Input: *clauses*, a set of clauses in propositional logic
Input: p, the probability of choosing to do a "random walk" move
Input: max_flips, number of flips allowed before giving up

```
model ← a random assignment of true/false to the symbols in clauses
for i ← 1 to max_flips do
    if model satisfies clauses then
        return model
    clause ← a randomly selected clause from clauses that is false in model
    if with probability p then
        flip the value in model of a randomly selected symbol from clause
    else
        flip whichever symbol in clause maximizes the number of satisfied clauses
return failure
```

While sound, WalkSAT is incomplete because its stochastic nature means it cannot prove unsatisfiability; it may fail to find a satisfying assignment even if one exists. The performance of WalkSAT can be enhanced by incorporating heuristics from DPLL. A pre-processing step can apply unit propagation and pure literal assignment to simplify the problem and reduce the search space before the local search begins. During the walk, the greedy choice can be modified to disallow flips that would violate a satisfied unit clause, thereby guiding the search more effectively. These modifications do not change the worst-case exponential complexity but often reduce the number of flips required in practice.

Interestingly, for randomly generated 3-CNF sentences with m clauses and n variables, the difficulty of satisfiability problems peaks around a critical ratio of $m/n \approx 4.3$. Problems in this phase transition region are computationally challenging for most SAT solvers.

2.8.1 Resolution Theorem Proving

Resolution is a powerful proof strategy that forms the basis of many modern automated theorem provers. It relies on a single inference rule, which, when applied systematically to a set of sentences in Conjunctive Normal Form (CNF), provides a sound and refutation-complete proof system. The core technique is proof by refutation.

Propositional Resolution The resolution inference rule for PL states that from two clauses containing complementary literals, a new clause can be inferred containing the disjunction of their

remaining literals.

$$\frac{(\alpha \vee \beta) \quad (\neg\beta \vee \gamma)}{(\alpha \vee \gamma)}$$

The process, known as resolution refutation, involves three steps:

1. Convert all sentences in the knowledge base to CNF.
2. Negate the desired conclusion (the query ϕ), convert it to CNF, and add its clauses to the KB.
3. Apply the resolution rule repeatedly to the set of clauses. If this process derives an empty clause ($\square$), which represents a contradiction, then the original query ϕ is proven to be entailed by the KB.

If the process terminates because no further resolution steps can be made and the empty clause has not been derived, then the query is not entailed. Consider a KB with premises $\{P \vee Q, P \rightarrow R, Q \rightarrow R\}$ and a query R. The proof is shown in Figure 2.18.

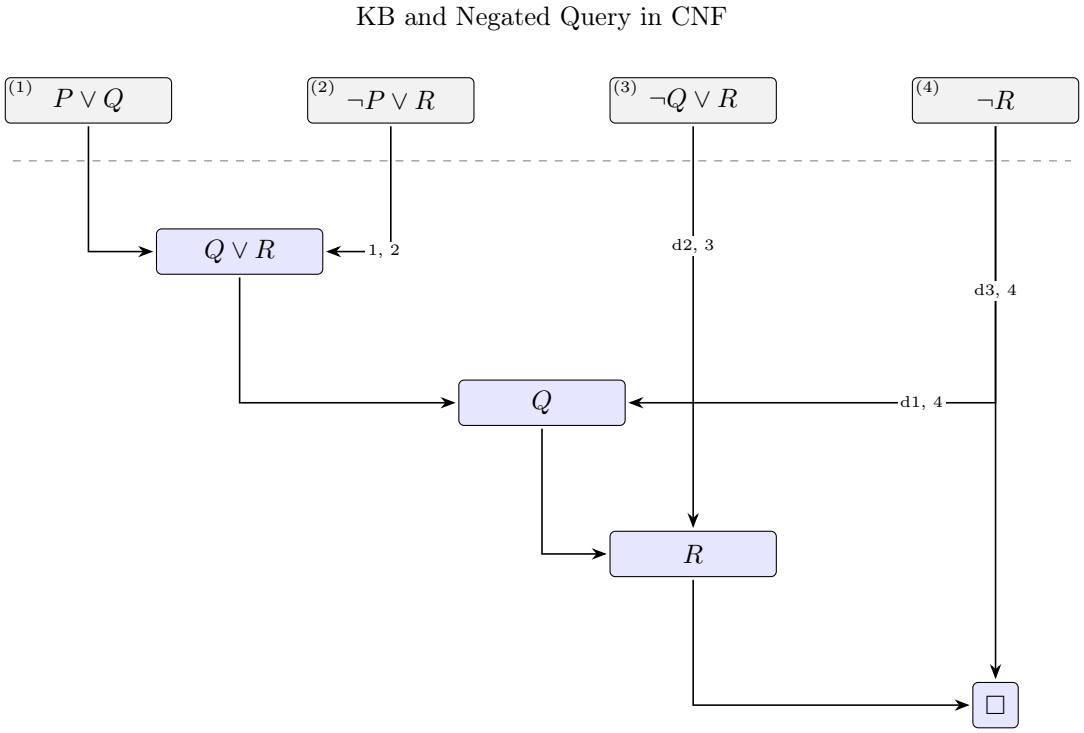


Figure 2.18: Resolution refutation proving KB $\models R$. Resolve (1) and (2) on P to get $Q \vee R$; with (4) on R to get Q; with (3) on Q to get R; with (4) to derive $\square$.

To improve efficiency, heuristics are often employed. The **unit preference** strategy prioritises resolution steps involving a unit clause, which always produces a shorter clause. The **set-of-support** strategy restricts resolutions to only those involving the negated query or clauses derived from it, focusing the search on relevant parts of the KB.

First-Order Resolution Extending resolution to FOL requires incorporating unification to handle variables. First, all sentences in the KB and the negated query must be converted to clausal form. This multi-step process standardises sentences for the resolution algorithm:

1. **Eliminate Implications:** Replace all instances of $\alpha \rightarrow \beta$ with $\neg\alpha \vee \beta$.

2. **Move Negation Inwards:** Use De Morgan's laws and quantifier duality rules ($\neg\forall \equiv \exists\neg$, $\neg\exists \equiv \forall\neg$) to push $\neg$ symbols so they only apply to atomic sentences.
3. **Standardise Variables:** Rename variables to ensure each quantifier binds a unique variable, preventing scope confusion (e.g., $\forall x.P(x) \wedge \exists x.Q(x)$ becomes $\forall x.P(x) \wedge \exists y.Q(y)$).
4. **Skolemise:** Remove existential quantifiers. An existentially quantified variable not within the scope of any universal quantifier is replaced by a unique Skolem constant (e.g., $\exists x.P(x)$ becomes $P(\texttt{C1})$). If it is within the scope of universal quantifiers, it is replaced by a unique Skolem function of the universally quantified variables (e.g., $\forall x.\exists y.P(x,y)$ becomes $\forall x.P(x,F(x))$).
5. **Drop Universal Quantifiers:** All remaining variables are assumed to be universally quantified.
6. **Distribute $\vee$ over $\wedge$:** Convert the quantifier-free sentences into a conjunction of clauses.

With sentences in clausal form, the first-order resolution rule can be applied. It is a "lifted" version of its propositional counterpart that incorporates unification.

$$\frac{(\alpha \vee l_1) \quad (\beta \vee \neg l_2)}{(\alpha\theta \vee \beta\theta)} \quad \text{where } \theta = \text{Unify}(l_1, l_2)$$

This rule applies to two clauses (with variables standardised apart) containing literals l_1 and l_2 that can be unified with an MGU θ. A new clause is inferred by taking the disjunction of the remaining literals and applying the substitution θ.

The entire process is demonstrated in Figure 2.19 with the "Curiosity Killed the Cat" problem.

1. Knowledge Base in First-Order Logic

(a) John owns a dog: $\exists x.D(x) \wedge O(J,x)$
(b) Anyone who owns a dog is an animal lover: $\forall x.(\exists y.D(y) \wedge O(x,y)) \rightarrow L(x)$
(c) Animal lovers do not kill animals: $\forall x.L(x) \rightarrow (\forall y.A(y) \rightarrow \neg K(x,y))$
(d) Either Jack killed Tuna or curiosity killed Tuna: $K(J,T) \vee K(C,T)$
(e) Tuna is a cat: $C(T)$
(f) All cats are animals: $\forall x.C(x) \rightarrow A(x)$

2. Conversion to Clausal Form

(a) From (a), after Skolemisation with constant `Fido`:
  (a)1. $D(\texttt{Fido})$
  (a)2. $O(J,\texttt{Fido})$
(b) From (b): $\neg D(y) \vee \neg O(x,y) \vee L(x)$ (3)
(c) From (c): $\neg L(x) \vee \neg A(y) \vee \neg K(x,y)$ (4)
(d) From (d): $K(J,T) \vee K(C,T)$ (5)
(e) From (e): $C(T)$ (6)
(f) From (f): $\neg C(x) \vee A(x)$ (7)

3. Resolution Refutation for Query: K(C,T)

8. Negated Query: $\neg K(C,T)$
9. (5, 8): $K(J,T)$
10. (7, 6), $\theta = \{x/T\}$: $A(T)$
11. (4, 9), $\theta = \{x/J, y/T\}$: $\neg L(J) \vee \neg A(T)$
12. (11, 10): $\neg L(J)$
13. (3, 12), $\theta = \{x/J\}$: $\neg D(y) \vee \neg O(J,y)$
14. (13, 2), $\theta = \{y/\texttt{Fido}\}$: $\neg D(\texttt{Fido})$
15. (14, 1): $\square$

Figure 2.19: A complete resolution refutation proof. The derivation of the empty clause ($\square$) from the KB and the negated query proves that curiosity did kill the cat.

2.8.2 Proving Validity

Resolution refutation can also be used to prove that a sentence is valid. A valid sentence is one that is true in all interpretations, which means it is entailed by the empty set of axioms.

$$\text{is\_valid}(\phi) \iff \{\} \vDash \phi$$

To prove validity using resolution refutation, one simply negates the sentence ϕ, converts $\neg\phi$ to clausal form, and attempts to derive a contradiction. If the empty clause is derived, the original sentence ϕ is proven to be valid.

Forward and Backward Chaining For knowledge bases restricted to Horn clauses (with at most one positive literal), inference can be performed in linear time.

- **Forward chaining** is data-driven. It applies Modus Ponens, firing any rule whose premises are satisfied in the KB to add new facts until the query is derived or a fixed point is reached.
- **Backward chaining** is goal-driven. It starts from the query and works backwards, finding rules that conclude it and attempting to prove their premises recursively. This is the fundamental mechanism used in logic programming systems like Prolog.

Both algorithms are sound and complete for Horn KBs.

2.8.3 FOL: Syntax and Semantics

The syntax of FOL introduces elements to denote objects and their relationships.

Definition 2.8.1. ***Term***. An expression denoting an object. A term can be a **Constant Symbol** (e.g., `Fred`), a **Variable** (e.g., x, y), or a **Function Symbol** applied to terms (e.g., `mother_of(John)`).

Sentences are built from terms. An **atomic sentence** (or atom) is the simplest formula, formed by applying a predicate symbol to a tuple of terms (e.g., `On(BlockA, BlockB)`) or stating equality ($t_1 = t_2$). A **propositional variable** is a 0-arity predicate. A **complex sentence** is constructed by combining atoms using logical connectives ($\wedge, \vee, \neg, \rightarrow, \leftrightarrow$).

If ϕ is a sentence and v is a variable, then $\forall v.\phi$ (universal quantification) and $\exists v.\phi$ (existential quantification) are sentences. Quantifiers bind variables; only closed formulas (with no free variables) have a truth value under an interpretation. The order of quantifiers is critical; $\forall x \exists y$ is not equivalent to $\exists y \forall x$. There is a duality between quantifiers: $\forall x.P(x) \equiv \neg\exists x.\neg P(x)$. Conventionally, implication ($\rightarrow$) is the main connective with $\forall$ and conjunction ($\wedge$) with $\exists$. A list of common FOL representations is provided in Table 2.12.

Table 2.12: Examples of natural language sentences represented in First-Order Logic.

Sentence	First-Order Logic Representation
Anything that glitters is gold.	$\forall x.\ \text{Glitters}(x) \rightarrow \text{Gold}(x)$
Some students took French in Spring 2001.	$\exists x.\ \text{Student}(x) \wedge \text{Took}(x, \text{French}, \text{S2001})$
Every student who takes French passes it.	$\forall x, t.\ (\text{Student}(x) \wedge \text{Took}(x, \text{French}, t)) \rightarrow \text{Passes}(x, \text{French}, t)$
Only one student took Greek.	$\exists x.\ \text{Student}(x) \wedge \text{Took}(x, \text{Greek}) \wedge (\forall y.\ (\text{Student}(y) \wedge \text{Took}(y, \text{Greek})) \rightarrow y = x)$
No person buys an expensive policy.	$\forall x, p.\ (\text{Person}(x) \wedge \text{Policy}(p) \wedge \text{Expensive}(p)) \rightarrow \neg\text{Buys}(x, p)$
All Germans speak the same languages.	$\forall x, y, l.\ (\text{German}(x) \wedge \text{German}(y)) \rightarrow (\text{Speaks}(x, l) \leftrightarrow \text{Speaks}(y, l))$

Definition 2.8.2. ***FOL: Interpretation***. An interpretation specifies what symbols refer to. It consists of:

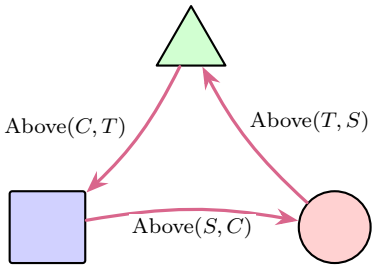
- A non-empty set of objects called the **universe** or **domain of discourse**, U.
- A mapping from constant symbols to objects in U.
- A mapping from predicate symbols to relations on U.
- A mapping from function symbols to functions on U.

The truth of a sentence is evaluated within an interpretation. An atomic sentence $P(t_1, \ldots, t_n)$ is true in interpretation I if the tuple of objects denoted by $\langle I(t_1), \ldots, I(t_n)\rangle$ is in the relation denoted by $I(P)$. For example, the sentence $\exists x, y.\ x = y$ is valid because in any interpretation with a non-empty domain, we can choose an element $d \in U$ and assign $x = d$ and $y = d$, making the equality true.

To handle quantifiers, we consider interpretations extended with a variable binding. Let $I_{x/a}$ be an interpretation identical to I except that variable x is assigned object $a \in U$.

- $\forall x.\phi$ is true in I if ϕ is true in $I_{x/a}$ for *every* object $a \in U$.
- $\exists x.\phi$ is true in I if ϕ is true in $I_{x/a}$ for *at least one* object $a \in U$.

Entailment is determined by models. For example, a KB containing only $P(a)$ and $P(b)$ does not entail $\forall x.P(x)$, because one can construct a model with a domain $\{a, b, c\}$ where $P(c)$ is false. In this model, the KB is true, but the conclusion is false. As illustrated in Figure 2.20, in the given interpretation, $\forall x.\exists y.\text{Above}(y, x)$ is true, as for every object x, we can find a y that is above it. However, $\exists y.\forall x.\text{Above}(y, x)$ is false, as no single object is above everything.



An FOL interpretation I

Universe U: { ● , ■ , ▲ }

Constant: $I(\texttt{Fred})$ = ■

Predicates:

$I(\text{Circle}) = \{$ ● $\}$

$I(\text{Square}) = \{$ ■ $\}$

$I(\text{Triangle}) = \{$ ▲ $\}$

$I(\text{Above}) = \{\ \langle T, S\rangle,\ \langle S, C\rangle,\ \langle C, T\rangle\ \}$

Quantified facts in I:

$\forall x\, \exists y\, \text{Above}(y, x)$ $\Rightarrow$ true (each object has some predecessor in the cycle)

$\exists y\, \forall x\, \text{Above}(y, x)$ $\Rightarrow$ false (no single object points to all)

Figure 2.20: A compact world and an interpretation I for First-Order Logic. Objects (left) are the domain elements; the binary relation $\text{Above}(y, x)$ is shown by arrows $y \to x$. The chosen interpretation makes $\forall x\, \exists y\, \text{Above}(y, x)$ true but $\exists y\, \forall x\, \text{Above}(y, x)$ false.

2.8.4 Inference in First-Order Logic

Inference in FOL is more complex due to variables and quantifiers. While a FOL KB can be propositionalised by instantiating variables with ground terms, this approach is often intractable. Function symbols can lead to infinite ground terms (the Herbrand universe), making the propositionalised

KB infinite. Turing and Church show that entailment in FOL is semi-decidable: an algorithm can confirm any entailed sentence, but no algorithm can refute any non-entailed sentence in finite time.

Unification and Generalised Modus Ponens To avoid the inefficiencies of propositionalisation, modern inference in FOL uses unification.

Definition 2.8.3. ***Unification***. The process of finding a substitution θ that makes two logical expressions identical. A key goal is to find the most general unifier (MGU).

For example, to unify `Knows(John, x)` and `Knows(y, Mother(y))`, the MGU is $\theta = \{y/\texttt{John}, x/\texttt{Mother(John)}\}$.

Unification is the core of the **Generalised Modus Ponens (GMP)** rule, which "lifts" Modus Ponens to FOL. Given atomic sentences $p'_1, \ldots, p'_n$ and a rule $(p_1 \wedge \ldots \wedge p_n \Rightarrow q)$, if a substitution θ exists such that $p'_i\theta = p_i\theta$ for all i, then we can infer $q\theta$.

$$\frac{p'_1, \ldots, p'_n, \quad (p_1 \wedge \ldots \wedge p_n \Rightarrow q)}{q\theta} \quad \text{where } p'_i\theta = p_i\theta$$

GMP allows inference to be performed directly on first-order sentences.

Forward and Backward Chaining in FOL With GMP, forward and backward chaining can be lifted to operate on FOL KBs composed of definite clauses (Horn clauses in FOL).

- **Forward Chaining in FOL** repeatedly applies GMP to add new atomic sentences to the KB until no new inferences can be made. It is sound and complete for definite clauses, but may not terminate if the query is not entailed.
- **Backward Chaining in FOL** uses GMP to work backwards from the query, unifying it with the conclusions of rules and recursively trying to prove the premises. This depth-first search is susceptible to infinite loops, requiring checks to ensure termination.

Consider the following KB:

1. $\forall x, y, z.\ \texttt{American}(x) \wedge \texttt{Weapon}(y) \wedge \texttt{Sells}(x, y, z) \wedge \texttt{Hostile}(z) \Rightarrow \texttt{Criminal}(x)$
2. $\exists x.\ \texttt{Owns}(\texttt{Nono}, x) \wedge \texttt{Missile}(x)$
3. $\forall x.\ \texttt{Missile}(x) \wedge \texttt{Owns}(\texttt{Nono}, x) \Rightarrow \texttt{Sells}(\texttt{West}, x, \texttt{Nono})$
4. $\forall x.\ \texttt{Missile}(x) \Rightarrow \texttt{Weapon}(x)$
5. $\forall x.\ \texttt{Enemy}(x, \texttt{America}) \Rightarrow \texttt{Hostile}(x)$
6. $\texttt{American}(\texttt{West})$
7. $\texttt{Enemy}(\texttt{Nono}, \texttt{America})$

Existential instantiation on (2) introduces a Skolem constant, M_1, giving: $\texttt{Owns}(\texttt{Nono}, M_1)$ and $\texttt{Missile}(M_1)$. A forward chaining algorithm can use these facts and the rules to infer successively that $\texttt{Weapon}(M_1)$, $\texttt{Sells}(\texttt{West}, M_1, \texttt{Nono})$, $\texttt{Hostile}(\texttt{Nono})$, and finally the query $\texttt{Criminal}(\texttt{West})$. A backward chaining system would start with the goal $\texttt{Criminal}(\texttt{West})$ and recursively prove the premises of rule (1).

2.8.5 Logic Programming

I did not take notes on this section; sorry.

2.9 Planning

Planning is the task of finding a sequence of actions to achieve a goal. While this can be framed as a problem-solving search, as discussed in section 2.3, planning algorithms typically use more structured representations of states and actions derived from logic. This allows them to reason about sets of states compactly and exploit the problem structure more effectively than searching through an explicit state space.

2.9.1 Situation Calculus

A formal approach to planning is the situation calculus, which uses first-order logic to model a changing world.

- **Situations are reified:** Situations (states) are treated as objects in the logic. A predicate that changes over time takes an extra situation argument, e.g., $\text{At}(\text{Robot}, \text{RoomB}, S_0)$.
- **Actions map situations to situations:** A function, $\text{Result}(a, s)$, denotes the situation resulting from performing action a in situation s.
- **Axioms describe change:**
  - **Effect axioms** specify how actions change the world. For instance, $\forall s.\text{At}(\text{Lobby}, s) \rightarrow \text{KnowsGate}(\text{Result}(\text{ReadDisplay}, s))$.
  - **Frame axioms** specify what does not change. For example, $\forall s, c.\text{Colour}(\text{Wall}, c, s) \rightarrow \text{Colour}(\text{Wall}, c, \text{Result}(\text{GoToGate}, s))$. The need to state all non-effects is known as the frame problem.

Planning in situation calculus becomes a theorem-proving task. Given a description of the initial state S_0 and a goal, the planner must prove that a situation s exists where the goal holds, e.g., $\exists s.\text{At}(\text{Gate1}, s) \wedge \text{Have}(\text{Ticket}, s)$. The constructive proof would yield a term for s, such as $\text{Result}(\text{GoToGate}, \text{Result}(\ldots, S_0))$, from which the plan can be extracted. While elegant, this approach is often computationally impractical.

2.9.2 The STRIPS Representation

A more efficient and widely-used alternative is the STRIPS representation, which restricts the language for describing states, goals, and actions.

- **States** are conjunctions of ground, positive literals (e.g., $\text{At}(\text{Home}) \wedge \text{Sells}(\text{SM}, \text{Milk})$). Anything not mentioned is unknown or assumed false (the closed-world assumption).
- **Goals** are conjunctions of literals, which may contain variables (e.g., $\text{Have}(\text{Milk}) \wedge \text{Have}(\text{Bananas})$).
- **Actions**, or operators, are defined by three components:
  - **Action Name:** A name with parameters, e.g., $\text{Buy}(x, \text{store})$.
  - **Preconditions:** A conjunction of literals that must be true for the action to be applicable.
  - **Effects:** A conjunction of literals describing the changes to the state. Positive effects are on the add-list, and negative effects are on the delete-list.

For example, a shopping domain might include the operators shown in Table 2.13.

Table 2.13: STRIPS operators for a simple shopping domain.

Action	**Buy(x, store)**
Preconditions	$\text{At}(\text{store}) \wedge \text{Sells}(\text{store}, x)$
Effects	$\text{Have}(x)$
Action	**Go(x, y)**
Preconditions	$\text{At}(x)$
Effects	$\text{At}(y) \wedge \neg\text{At}(x)$

2.9.3 State-Space Planning Algorithms

With the STRIPS representation, planning can be viewed as a search problem.

- **Progression Planning (Forward Search):** This approach searches forward from the initial state to the goal. A node in the search space is a state (a set of literals), and an edge is a ground action. This is often inefficient due to a high branching factor, as many actions are applicable in any given state.
- **Regression Planning (Backward Search):** This searches backward from the goal. A node is a sub-goal (a set of literals). To move from a sub-goal G to a preceding sub-goal, an action A is chosen that has an effect in G. The new sub-goal is formed by the preconditions of A, plus any parts of G not achieved by A. This is more goal-directed but can still involve significant branching.

2.9.4 Partial-Order Planning

A more flexible and often more efficient approach is partial-order planning (POP), which searches through the space of plans rather than the space of states. This method embodies a least-commitment strategy, only adding ordering constraints between actions when necessary.

A partial-order plan consists of four components:

1. A set of **steps** (the actions in the plan).
2. A set of **ordering constraints**, $S_i < S_j$, meaning step S_i must execute before S_j.
3. A set of **variable binding constraints**, e.g., $v = x$.
4. A set of **causal links**, $S_i \xrightarrow{c} S_j$, indicating that step S_i achieves precondition c for step S_j.

The planning process begins with a minimal plan containing only two steps: a `Start` step, whose effects are the initial state literals, and a `Finish` step, whose preconditions are the goal literals. The algorithm, detailed in Partial-Order Planning Algorithms, iteratively adds steps and constraints to resolve open preconditions and threats until a complete and consistent plan is found.

Definition 2.9.1. ***Open Precondition***. An open precondition is a precondition of a step not yet satisfied by a causal link.

Definition 2.9.2. ***Threat***. A threat is a step that might undo a causal link.

Specifically, a step S_k threatens a link $S_i \xrightarrow{c} S_j$ if S_k has an effect $\neg c$ and could potentially be ordered between S_i and S_j. Threats are resolved by adding ordering constraints, either forcing the threat to occur before the link's provider ($S_k < S_i$) or after its consumer ($S_j < S_k$).

Algorithm 20: `POP`(*initial_plan*)

Input: *plan*, a partial plan

```
if no open preconditions in plan then
    return plan
S_need, c ← Select-Open-Precondition(plan) // Select an open precondition
for each step S_add (new or existing) that can add c do
    plan' ← plan
    Add causal link S_add --c--> S_need to plan'
    Add ordering constraint S_add < S_need to plan'
    if S_add is new then
        Add S_add to steps in plan'
        Add constraints Start < S_add < Finish to plan'
    plan' ← Resolve-Threats(plan') // Demote or promote any threats
    if plan' is not a failure then
        result ← POP(plan')
        if result ≠ failure then
            return result
return failure
```

An example of the POP algorithm building a shopping plan is shown in Figure 2.21. The final plan contains all necessary steps, but the ordering between buying milk and buying bananas is left unspecified, demonstrating the least-commitment nature of the algorithm. Any valid topological sort of this partial order, such as ⟨Go(Home,SM), Buy(Milk,SM), Buy(Bananas,SM)⟩, constitutes a solution.

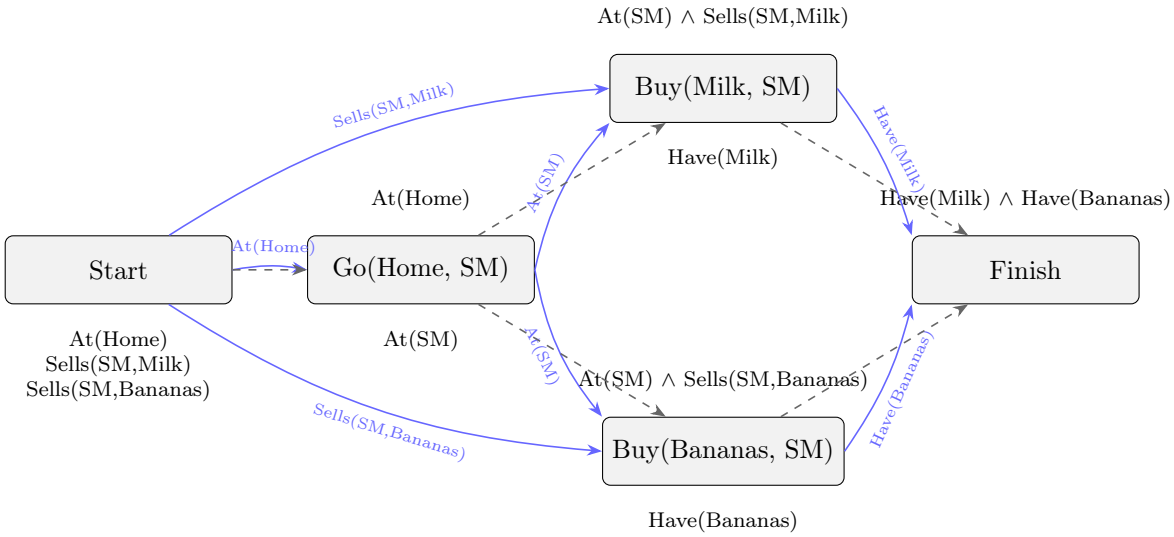


Figure 2.21: A complete partial-order plan for the shopping problem. Causal links (solid blue) show how preconditions are satisfied. Ordering constraints (dashed) enforce temporal sequence. The plan leaves the order of buying milk vs. bananas unconstrained.

2.9.5 Partial-Order Planning Algorithms

An alternative to searching through the state space is to search through the space of plans. This approach, known as plan-space search or partial-order planning (POP), starts with an empty plan and iteratively adds actions and constraints until a valid solution is formed. This method decouples the order of planning from the order of execution and embodies a least-commitment strategy, where decisions about action ordering or variable bindings are deferred until necessary.

A partial-order plan is defined by four components:

- A set of **steps** (the actions in the plan).
- A set of **ordering constraints**, $S_i < S_j$, meaning step S_i must execute before S_j.
- A set of **variable binding constraints**, e.g., $v = x$.
- A set of **causal links**, $S_i \xrightarrow{c} S_j$, indicating that step S_i achieves precondition c for step S_j.

The planning process begins with a minimal plan containing two steps: a `Start` step, whose effects are the initial state literals, and a `Finish` step, whose preconditions are the goal literals, with an initial ordering constraint `Start` $<$ `Finish`. The algorithm, outlined in Partial-Order Planning Algorithms, then iteratively refines this plan.

Algorithm 21: `POP`(*initial_plan*)

Input: *plan*, a partial plan

if *no open preconditions in plan* **then**
  return *plan*
$S_{need}, c \leftarrow$ `Select-Open-Precondition`(*plan*) ▷ Select an open precondition
for *each step* S_{add} *(new or existing) that can add* c **do**
  $plan' \leftarrow plan$
  Add causal link $S_{add} \xrightarrow{c} S_{need}$ to $plan'$
  Add ordering constraint $S_{add} < S_{need}$ to $plan'$
  if S_{add} *is new* **then**
    Add S_{add} to steps in $plan'$
    Add constraints `Start` $< S_{add} <$ `Finish` to $plan'$
  $plan' \leftarrow$ `Resolve-Threats`($plan'$) ▷ Demote or promote any threats
  if $plan'$ *is not a failure* **then**
    $result \leftarrow$ `POP`($plan'$)
    if $result \neq failure$ **then**
      return *result*
return *failure*

The algorithm operates by repeatedly selecting an open precondition.

Definition 2.9.3. ***Open Precondition***. Is a precondition of a step not yet satisfied by a causal link—and finding a way to achieve it. This can be done by using an existing step in the plan or by adding a new one. This choice represents a branch in the search space.

Once a causal link $S_i \xrightarrow{c} S_j$ is added, the planner must protect it from threats. A threat is a step S_k that has an effect $\neg c$ and could potentially be ordered between S_i and S_j. Threats are resolved by adding ordering constraints:

- **Demotion:** Force the threatening step to occur after the link ($S_j < S_k$).
- **Promotion:** Force the threatening step to occur before the link ($S_k < S_i$).

If a threat cannot be resolved, that branch of the search fails. The process continues until there are no open preconditions, resulting in a complete and consistent plan.

Example: Shopping Problem Consider the shopping problem with the goal Have(Milk) ∧ Have(Bananas) ∧ Have(Drill). The initial state is At(Home), with knowledge that a supermarket (SM) sells milk and bananas, and a hardware store (HW) sells drills. The POP algorithm might produce the plan shown in Figure 2.22. In this solution, the agent must go to the hardware store before the supermarket. However, the order of buying milk and bananas at the supermarket is left unconstrained, demonstrating the least-commitment nature of the planner.

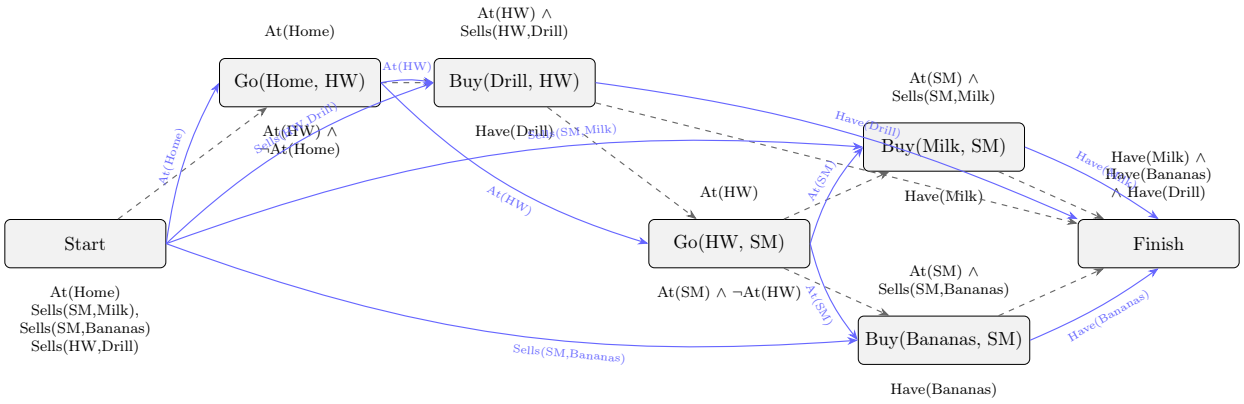


Figure 2.22: Partial-order plan for the shopping problem with an HW → SM route. Causal links (solid blue) satisfy preconditions; dashed constraints enforce only what's necessary. The order of buying milk vs. bananas remains unconstrained.

Example: The Sussman Anomaly A classic problem illustrating the power of POP is the Sussman Anomaly, shown in Figure 2.23. The goal is to create the stack `On(A,B)` ∧ `On(B,C)`. A simple planner that tries to solve sub-goals independently might achieve `On(A,B)` first by moving C to the table, but this makes achieving `On(B,C)` difficult. Similarly, achieving `On(B,C)` first traps block A. The problem requires interleaving the steps of the sub-plans. POP naturally handles such interactions by only adding ordering constraints to resolve threats. It would correctly deduce the optimal plan: move C to the table, then move B onto C, and finally move A onto B.

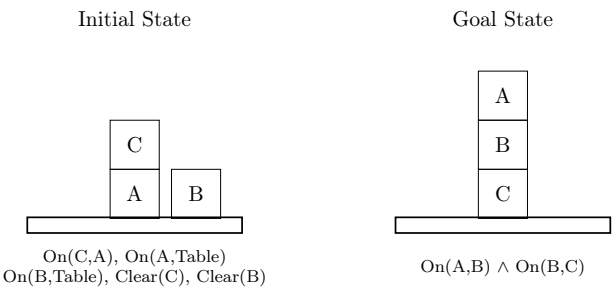


Figure 2.23: The Sussman Anomaly problem. The interdependent sub-goals make it challenging for planners that do not interleave steps.

While POP is sound and complete, its efficiency depends heavily on heuristics for selecting open preconditions and resolving threats. Without good guidance, the search space of partial plans can be vast.

2.9.6 GraphPlan

In contrast to the heuristic, search-based nature of POP, the GraphPlan algorithm offers a more systematic approach. Developed in the mid-1990s, it marked a significant advance in planner performance by transforming the planning problem into a search on a specialised data structure called a plan graph. GraphPlan operates on propositional logic, meaning all actions and states are ground (free of variables). This leads to a larger problem representation but allows for more efficient algorithmic processing.

The core of GraphPlan is an iterative process:

1. **Graph Expansion:** Construct a plan graph forward from the initial state up to a fixed depth, k.
2. **Solution Extraction:** Search backward from the goal propositions at depth k to find a valid sub-graph that constitutes a plan.
3. **Iteration:** If no solution is found, increment k and repeat.

The Plan Graph A plan graph is a layered graph that alternates between levels of propositions and levels of actions.

- **Proposition Levels (S_i):** Even-numbered levels contain all propositions that could possibly be true at that time step. S_0 contains the initial state literals.
- **Action Levels (A_i):** Odd-numbered levels contain all ground actions whose preconditions are present and non-mutex in the preceding proposition level. This also includes maintenance actions (or no-ops), which carry a proposition forward unchanged.

The graph is built layer by layer. For each new layer, GraphPlan also computes mutex (mutually exclusive) links between pairs of nodes (actions or propositions) that cannot be true at the same time.

Mutex Propagation Mutex links are crucial for pruning the search space. They are propagated forward through the graph as it is built.

- Two **actions** at level A_i are mutex if:
  (a) **Inconsistent Effects:** One action's effect is the negation of another's (e.g., one adds P, the other adds $\neg P$).
  (b) **Interference:** One action deletes a precondition of the other.
  (c) **Competing Needs:** A pair of their preconditions is mutex at the previous level, S_{i-1}.
- Two **propositions** at level S_i are mutex if:
  (a) One is the negation of the other.
  (b) **Inconsistent Support:** All pairs of actions at level A_{i-1} that could achieve the two propositions are themselves mutex.

As the graph grows, the number of mutex links tends to decrease or level off, representing that more states become reachable over time.

Solution Extraction Once the graph is expanded to a level k where all goal propositions exist and are not mutually exclusive, a backward search begins. The search, outlined in Solution Extraction, attempts to find a non-mutex set of actions at level A_{k-1} to achieve the goals. The preconditions of this set of actions become the sub-goals for the previous level, S_{k-1}, and the process repeats until the initial state at S_0 is reached. If the search fails at any level, backtracking occurs. If all backward searches from level k fail, the graph is expanded to $k+1$.

Algorithm 22: `GraphPlan`(*problem*)

```
graph ← Initialise-Graph(problem)
for k ← 1, 2, ... do
    if Goal propositions all exist in S_k of graph and are not mutex then
        solution ← Extract-Solution(Goals, graph, k)
        if solution ≠ failure then
            return solution
    if Graph has leveled off then
        return failure
    graph ← Expand-Graph(graph)
```

Algorithm 23: `Extract-Solution`(*goals*, *graph*, *k*)

```
Function Extract-Solution(goals, graph, k)
    if k = 0 then
        return empty plan
    foreach non-mutex action set A in A_{k-1} that achieves goals do
        subgoals ← preconditions of actions in A
        subplan ← Extract-Solution(subgoals, graph, k − 1)
        if subplan ≠ failure then
            return A ∪ subplan
    return failure
```

An example of a plan graph for a simple "birthday dinner" problem is shown in Table 2.14 and Table 2.15. The goal is to have dinner ready, the present wrapped, and no garbage. At depth 1 (Action level A_1, Proposition level S_2), a plan is impossible because achieving all three goals requires actions that are mutually exclusive (e.g., "Cook" and "Carry" interfere). At depth 2, the mutex constraints relax, allowing for a valid two-step plan to be found: in the first step, "Cook" and "Wrap"; in the second step, "Carry".

Table 2.14: Plan graph: nodes per level

Level	Kind	Contents
S_0	Propositions	G, CH, Q
A_1	Actions	Cook, Wrap, Carry, Dolly, noop(G), noop(CH), noop(Q)
S_1	Propositions	D, P, noG, G, CH, noCH, Q, noQ
A_2	Actions	Carry, Cook, Wrap, Dolly, noop(D), noop(P), noop(noG), noop(G), noop(CH), noop(noCH), noop(Q), noop(noQ)
S_2	Propositions	**D**, **P**, **noG**, G, CH, noCH, Q, noQ

Table 2.15: Complete plan-graph summary (preconditions, effects, mutex)

(a) Action preconditions

Level	Action	Requires
A_1	Cook	CH
	Wrap	Q
	Carry	(none)
	Dolly	(none)
	noop(G)	G
	noop(CH)	CH
	noop(Q)	Q
A_2	Cook	CH
	Wrap	Q
	Carry	(none)
	Dolly	(none)
	noop(D)	D
	noop(P)	P
	noop(noG)	noG
	noop(G)	G
	noop(CH)	CH
	noop(noCH)	noCH
	noop(Q)	Q
	noop(noQ)	noQ

(b) Action effects

Action	Effects (add-list)
Cook	D
Wrap	P
Carry	noG, noCH
Dolly	noG, noQ
noop(X)	X

(c) Mutex relations (as drawn)

Level	Type	Pairs
A_1	action–action	(cook1, carry1), (wrap1, dolly1), (carry1, noopCH1), (dolly1, noopQ1)
S_1	prop–prop	(G1, noG1), (CH1, noCH1), (Q1, noQ1), (D1, noCH1), (P1, noQ1)
S_2	prop–prop	(G2, noG2), (CH2, noCH2), (Q2, noQ2)

Chapter 3

Machine Learning

3.1 A Formal Model for Learning

To formalise what it means for an agent to learn to "do the right thing" as described earlier (Learning), we require a mathematical framework. The statistical learning framework provides this by defining the components of a learning problem and the criteria for success [8].

3.1.1 The Statistical Learning Framework

The core task of a supervised learning algorithm is to learn a mapping from inputs to outputs based on a set of examples. This process can be deconstructed into several key components:

- **Instance Space ($\mathcal{X}$):** An arbitrary set containing the objects to be labelled. Each object, or instance, is typically represented by a vector of features. For example, in a medical diagnosis task, $\mathcal{X}$ could be the space of all possible patient measurements.
- **Label Set ($\mathcal{Y}$):** The set of possible outcomes or labels. For binary classification, this is often $\mathcal{Y} = \{0, 1\}$ or $\{-1, +1\}$.
- **Training Data (S):** A finite sequence of labelled examples, $S = ((x_1, y_1), \ldots, (x_m, y_m))$, where each pair (x_i, y_i) is an element of $\mathcal{X} \times \mathcal{Y}$. This data is the sole source of information available to the learner.

The learner's objective is to use the training data S to produce a hypothesis, $h : \mathcal{X} \to \mathcal{Y}$. This hypothesis, also called a predictor or classifier, is the function the agent will use to predict the label of new, unseen instances from $\mathcal{X}$.

This setup assumes the training data is generated from some underlying, unknown probability distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$. This distribution represents the environment. The fundamental assumption of statistical learning is that the training examples are sampled independently and identically distributed (i.i.d.) from $\mathcal{D}$. The learner is blind to $\mathcal{D}$ and must infer its properties solely through the window provided by S.

3.1.2 Risk, Minimisation, and Overfitting

To measure the success of a hypothesis, we need to define its error, or risk. There are two distinct measures of risk.

Definition 3.1.1. ***Generalisation Risk***. The true risk (or generalisation error) of a hypothesis h is its expected error on new data drawn from the underlying distribution $\mathcal{D}$. For classification, this is the probability of misclassification:

$$L_{\mathcal{D}}(h) = \mathbb{P}_{(x,y)\sim\mathcal{D}}[h(x) \neq y]$$

The true risk, $L_{\mathcal{D}}(h)$, is the ultimate measure of a hypothesis's quality, but it cannot be calculated directly because $\mathcal{D}$ is unknown. The learner can only measure performance on the data it has.

Definition 3.1.2. ***Empirical Risk***. The empirical risk (or training error) of a hypothesis h is its average error on the training set S:

$$L_S(h) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{I}(h(x_i) \neq y_i)$$

where $\mathbb{I}(\cdot)$ is the indicator function.

This leads to a natural learning principle: since the true risk is unknowable, the learner should instead aim to minimise the risk on the training data. This strategy is known as **Empirical Risk Minimisation (ERM)**.

However, naively applying ERM can lead to a significant problem. Consider a scenario where a learner has enough flexibility to choose a highly complex hypothesis that perfectly memorises every training example, including any noise. While its empirical risk would be zero, this hypothesis would perform poorly on new data because it has failed to learn the true underlying pattern. This phenomenon is known as overfitting.

3.1.3 Inductive Bias and Hypothesis Classes

To combat overfitting, the ERM principle is constrained. Rather than allowing the learner to choose any possible function, we restrict its search to a predefined set of functions called a **hypothesis class**, denoted $\mathcal{H}$. By choosing a class $\mathcal{H}$ in advance, we introduce an inductive bias, which is a set of assumptions about the form of the solution. The learning algorithm then becomes:

$$\text{ERM}_{\mathcal{H}}(S) \in \underset{h\in H}{arg\,min}\, L_S(h)$$

Choosing a more restricted hypothesis class provides better protection against overfitting but may introduce a stronger bias, potentially preventing the learner from finding a good solution if the true pattern lies outside the class. This trade-off is a fundamental concept in machine learning.

3.1.4 Model Selection and Generalisation Control

Minimising the empirical risk on S can understate the generalisation risk, especially when we overfit the training data. To keep our estimate of out-of-sample performance clean, we split data into `train`, `validation`, and `test` sets. We fit parameters on `train`; we choose hyperparameters (e.g., the capacity of $\mathcal{H}$, regularisation strength, or an early-stopping epoch) by minimising the error on `validation`; and we report final performance once on `test`. This protocol ensures the final risk estimate is not biased by repeated tuning on the test data.

We control model capacity in three primary ways. First, we constrain $\mathcal{H}$ directly through architecture or feature set choices, which sets the inductive bias for $\mathrm{ERM}_{\mathcal{H}}(S)$. Second, we penalise complexity via regularisation:

$$\hat{h} \in \underset{h \in H}{arg\,min}\ L_S(h) + \lambda\Omega(h),$$

where $\Omega(h)$ encodes a simplicity preference and λ is selected on the `validation` set. Third, we use early stopping: while training, we monitor the `validation` error and halt the process when it stops decreasing, retaining the parameters from the epoch with the best performance.

All of these choices are consistent with the PAC framework in PAC Learnability: we are trading estimation error against approximation error through the size of $\mathcal{H}$ and the strength of the bias we impose.

3.1.5 Guarantees for Finite Hypothesis Classes

The PAC framework provides a formal guarantee that ERM can succeed. To understand how such a guarantee is derived, we can analyse the simplest case: a finite hypothesis class $\mathcal{H}$ under the realizability assumption.

Definition 3.1.3. ***Realizability Assumption***. There exists a hypothesis $h^* \in \mathcal{H}$ such that its true risk is zero, i.e., $L_{\mathcal{D}}(h^*) = 0$.

***Lemma* 3.1.1.** *ERM Zero Training Error Under Realizability.* Assume Realizability Assumption. For any sample S and any ERM solution $h_S \in \mathrm{ERM}_{\mathcal{H}}(S)$, we have $L_S(h_S) = 0$.

Proof. By realizability, there exists $h^\star \in \mathcal{H}$ with $L_{\mathcal{D}}(h^\star) = 0$, which implies $L_S(h^\star) = 0$ with probability 1 over the draw of S (and in any case $L_S(h^\star) \geq 0$). Therefore $\min_{h\in\mathcal{H}} L_S(h) \leq L_S(h^\star) = 0$. Since empirical risk is nonnegative, any empirical risk minimizer h_S satisfies $L_S(h_S) = 0$. ■

Our learning algorithm fails if it selects a hypothesis h_S that has zero empirical risk but a high true risk ($L_{\mathcal{D}}(h_S) > \epsilon$). The following lemmas formalise this failure condition and its probability.

***Lemma* 3.1.2.** *ERM Failure Implies a Misleading Sample.* Let $\mathcal{H}_B = \{h \in \mathcal{H} : L_{\mathcal{D}}(h) > \epsilon\}$. Under Realizability Assumption, if $h_S \in \mathrm{ERM}_{\mathcal{H}}(S)$ satisfies $L_{\mathcal{D}}(h_S) > \epsilon$, then S lies in the set

$$M \;=\; \bigcup_{h\in\mathcal{H}_B} \{S :\ L_S(h) = 0\}.$$

Equivalently, $\{S :\ L_{\mathcal{D}}(h_S) > \epsilon\} \subseteq M$.

Proof. By Lemma 3.1.1, $L_S(h_S) = 0$. If additionally $L_{\mathcal{D}}(h_S) > \epsilon$, then $h_S \in \mathcal{H}_B$ and $S \in \{S : L_S(h_S) = 0\} \subseteq M$. Hence the stated inclusion. ■

***Lemma* 3.1.3.** *Bad Hypotheses Look Perfect With Small Probability.* Fix $\epsilon \in (0, 1)$ and $h \in \mathcal{H}$ with $L_{\mathcal{D}}(h) > \epsilon$. For an i.i.d. sample $S \sim \mathcal{D}^m$,

$$\mathbb{P}(L_S(h) = 0) \leq (1 - \epsilon)^m \leq e^{-\epsilon m}.$$

Proof. For one fresh draw $(x, y) \sim \mathcal{D}$, $\mathbb{P}[h(x) = y] = 1 - L_{\mathcal{D}}(h) < 1 - \epsilon$. By independence over m examples, $\mathbb{P}(L_S(h) = 0) = \prod_{i=1}^m \mathbb{P}[h(x_i) = y_i] \leq (1 - \epsilon)^m$. Finally, use $1 - x \leq e^{-x}$ for $x \in [0, 1]$ to obtain $(1 - \epsilon)^m \leq e^{-\epsilon m}$. ■

With these lemmas, we can prove the main result for finite hypothesis classes.

***Lemma* 3.1.4.** *Union Bound.* For any set of events $A_1, \ldots, A_k$, the probability of their union is bounded by the sum of their individual probabilities:

$$\mathbb{P}\left(\bigcup_{i=1}^{k} A_i\right) \leq \sum_{i=1}^{k} \mathbb{P}(A_i).$$

***Proposition* 3.1.1.** *Finite Class Sample Complexity.* For a finite hypothesis class $\mathcal{H}$, if the realizability assumption holds, for any $\epsilon, \delta \in (0,1)$, if the sample size m satisfies

$$m \geq \frac{1}{\epsilon} \ln\left(\frac{|\mathcal{H}|}{\delta}\right)$$

then with probability at least $1-\delta$, the $\mathrm{ERM}_{\mathcal{H}}$ algorithm will return a hypothesis h_S with $L_{\mathcal{D}}(h_S) \leq \epsilon$.

Proof. Let $\mathcal{H}_B = \{h \in \mathcal{H} : L_{\mathcal{D}}(h) > \epsilon\}$. By Lemma 3.1.2,

$$\mathbb{P}(L_{\mathcal{D}}(h_S) > \epsilon) \;\leq\; \mathbb{P}\left(\bigcup_{h \in \mathcal{H}_B} \{S :\; L_S(h) = 0\}\right) \;\leq\; \sum_{h \in \mathcal{H}_B} \mathbb{P}(L_S(h) = 0),$$

where we used the Union Bound. By Lemma 3.1.3, each summand is at most $e^{-\epsilon m}$, so

$$\mathbb{P}(L_{\mathcal{D}}(h_S) > \epsilon) \;\leq\; |\mathcal{H}_B|\, e^{-\epsilon m} \;\leq\; |\mathcal{H}|\, e^{-\epsilon m}.$$

To make this at most δ, it suffices that $|\mathcal{H}|e^{-\epsilon m} \leq \delta$, i.e. $m \geq \frac{1}{\epsilon}\ln\left(\frac{|\mathcal{H}|}{\delta}\right)$. ■

This result is a concrete example of Probably Approximately Correct (PAC) learning, which we now formally define [9].

3.1.6 Probably Approximately Correct (PAC) Learning

The theory of Probably Approximately Correct (PAC) learning provides a formal guarantee that the ERM strategy, when applied over a suitable hypothesis class $\mathcal{H}$, will not overfit.

Definition 3.1.4. ***PAC Learnability***. A hypothesis class $\mathcal{H}$ is PAC learnable if there is an algorithm and a function $m_{\mathcal{H}}(\epsilon, \delta)$ such that for any distribution $\mathcal{D}$, given $m \geq m_{\mathcal{H}}(\epsilon, \delta)$ i.i.d. samples, the algorithm produces a hypothesis $h \in \mathcal{H}$ that, with probability at least $1 - \delta$, satisfies $L_{\mathcal{D}}(h) \leq \epsilon$.

Corollary. *Finite Classes are PAC Learnable.* Every finite $\mathcal{H}$ is PAC learnable in the realizable case with sample complexity

$$m_{\mathcal{H}}(\epsilon, \delta) \;\leq\; \left\lceil \frac{1}{\epsilon} \ln\left(\frac{|\mathcal{H}|}{\delta}\right) \right\rceil.$$

Proof. Instantiate Proposition 3.1.1 with $m = \left\lceil \frac{1}{\epsilon}\ln\left(\frac{|\mathcal{H}|}{\delta}\right)\right\rceil$ and use the definition of PAC learnability. ■

Definition 3.1.5. ***Sample Complexity***. The number of samples required is known as the sample complexity and depends on the accuracy parameter, ϵ, and the confidence parameter, δ.

3.1.7 Generalisations of the Learning Model

The basic PAC model can be extended to be more applicable to real-world problems.

Agnostic PAC Learning The standard PAC model's realizability assumption is often too strong for practical tasks. The Agnostic PAC model removes it. The goal is to find a hypothesis h whose error is not much worse than the best possible hypothesis within the class. This requires a benchmark for the lowest possible error. For classification, this is the Bayes optimal classifier.

***Lemma* 3.1.5.** *Bayes Optimality for 0–1 Loss.* Let $\eta(x) = \mathbb{P}[y = 1 \mid x]$ and define the Bayes classifier $f_{\mathcal{D}}(x) = \mathbf{1}\{\eta(x) \geq 1/2\}$. Then for any classifier $g : \mathcal{X} \to \{0, 1\}$,

$$L_{\mathcal{D}}(f_{\mathcal{D}}) \;\leq\; L_{\mathcal{D}}(g).$$

Moreover,

$$L_{\mathcal{D}}(g) - L_{\mathcal{D}}(f_{\mathcal{D}}) \;=\; \mathbb{E}_x[|2\eta(x) - 1|\;\mathbf{1}\{g(x) \neq f_{\mathcal{D}}(x)\}] \geq 0.$$

Proof. Condition on x. For any $a \in \{0, 1\}$, $\mathbb{P}(y \neq a \mid x) = a\,(1 - \eta(x)) + (1 - a)\,\eta(x)$. This is minimized by choosing $a = 1$ when $\eta(x) \geq 1/2$ and $a = 0$ otherwise, which is exactly $f_{\mathcal{D}}(x)$. The excess risk identity follows by writing

$$\mathbb{P}(y \neq g(x) \mid x) - \mathbb{P}(y \neq f_{\mathcal{D}}(x) \mid x) = |2\eta(x) - 1|\;\mathbf{1}\{g(x) \neq f_{\mathcal{D}}(x)\},$$

and integrating over x. ■

Since we cannot hope to outperform the Bayes optimal predictor, the agnostic guarantee is that, with probability at least $1 - \delta$:

$$L_{\mathcal{D}}(h) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \epsilon$$

This provides a more robust and realistic framework.

Generalised Loss Functions The learning framework is not limited to binary classification. By generalising the measure of error, a wide variety of tasks can be accommodated. A loss function, $\ell(h, z)$, measures the penalty for a hypothesis h on a single example z. The goal is to minimise the expected loss, $L_{\mathcal{D}}(h) = \mathbb{E}_{z \sim \mathcal{D}}[\ell(h, z)]$.

Examples include:

- **Multiclass Classification:** The label set $\mathcal{Y}$ contains multiple categories. The 0-1 loss function is $\ell_{0\text{–}1}(h, (x, y)) = \mathbf{1}\{h(x) \neq y\}$.
- **Regression:** The task is to predict a real-valued outcome ($\mathcal{Y} \subseteq \mathbb{R}$). A common choice is the square loss, $\ell_{sq}(h, (x, y)) = (h(x) - y)^2$.

***Lemma* 3.1.6.** *0–1 Loss Equals Misclassification Probability.* For binary or multiclass classification with 0–1 loss, $L_{\mathcal{D}}(h) = \mathbb{P}_{(x,y) \sim \mathcal{D}}[h(x) \neq y]$.

Proof. By definition of the true risk with loss $\ell_{0\text{–}1}(h, (x, y)) = \mathbf{1}\{h(x) \neq y\}$ and the fact that $\mathbb{E}[\mathbf{1}\{A\}] = \mathbb{P}(A)$. ■

This generalisation allows the ERM principle and the PAC framework to provide a theoretical foundation for a vast range of machine learning problems.

3.2 From State-Space Search to Hypothesis-Space Search

A common thread unites all the algorithms in this §2.3: they systematically explore the state space without any problem-specific knowledge to guide them towards a goal. This paradigm of blind, exhaustive enumeration has a powerful analogue in the Machine Learning framework, specifically in the ERM principle over a finite hypothesis class $\mathcal{H}$.

In this analogy, the learning algorithm "searches" the hypothesis space $\mathcal{H}$ for the "best" hypothesis. The goal is not a specific state, but the hypothesis $h \in \mathcal{H}$ that minimises the true risk, $L_{\mathcal{D}}(h)$. Since the true risk is unknowable, the algorithm uses the empirical risk, $L_S(h)$, as a proxy. The ERM strategy is therefore equivalent to an exhaustive search of $\mathcal{H}$, evaluating each hypothesis on the training set S and selecting one that minimises this empirical cost.

This "search" is only meaningful if the empirical risk is a reliable estimate of the true risk. If not, finding a hypothesis with low training error provides no guarantee of good performance on new data. The theory of uniform convergence provides the formal assurance that, given a sufficiently large training sample, the empirical risk for *all* hypotheses in $\mathcal{H}$ will be close to their true risk. This ensures that the minimum of the empirical landscape corresponds to a point near the minimum of the true risk landscape.

Definition 3.2.1. ***ϵ-Representative Sample***. A training set S is called ϵ-representative with respect to a hypothesis class $\mathcal{H}$, a loss function ℓ, and a distribution $\mathcal{D}$, if for all $h \in \mathcal{H}$, $|L_S(h) - L_{\mathcal{D}}(h)| \leq \epsilon$.

If a sample is representative, the ERM principle is guaranteed to yield a hypothesis with near-optimal true risk.

***Lemma* 3.2.1.** *ERM Under Representativeness Yields Near-Optimal Risk.* Assume that a training set S is $(\epsilon/2)$-representative. Then, any hypothesis $h_S \in \mathrm{ERM}_{\mathcal{H}}(S)$ satisfies

$$L_{\mathcal{D}}(h_S) \leq \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h) + \epsilon.$$

Proof. For any $h \in \mathcal{H}$, we have

$$L_{\mathcal{D}}(h_S) \leq L_S(h_S) + \frac{\epsilon}{2} \leq L_S(h) + \frac{\epsilon}{2} \leq L_{\mathcal{D}}(h) + \frac{\epsilon}{2} + \frac{\epsilon}{2} = L_{\mathcal{D}}(h) + \epsilon.$$

The first and third inequalities hold because S is $(\epsilon/2)$-representative. The second inequality holds by the definition of h_S as an empirical risk minimiser. ■

This lemma shifts the problem of guaranteeing learnability to guaranteeing that a sample will be ϵ-representative with high probability. This is formalised by the uniform convergence property.

Definition 3.2.2. ***Uniform Convergence***. A hypothesis class $\mathcal{H}$ has the uniform convergence property if there exists a function $m_{\mathcal{H}}^{UC}(\epsilon, \delta)$ such that for any $\epsilon, \delta \in (0, 1)$ and any probability distribution $\mathcal{D}$, if $m \geq m_{\mathcal{H}}^{UC}(\epsilon, \delta)$, then an i.i.d. sample S of size m is ϵ-representative with probability at least $1 - \delta$.

To show that finite classes have this property, we use a concentration inequality to bound the deviation of the empirical risk from the true risk for a single hypothesis, and then apply the Union Bound to extend this guarantee to the entire class.

***Lemma* 3.2.2.** *Hoeffding's Inequality.* Let $\theta_1, \ldots, \theta_m$ be a sequence of i.i.d. random variables such that for all i, $\mathbb{E}[\theta_i] = \mu$ and $\mathbb{P}[a \leq \theta_i \leq b] = 1$. Then, for any $\epsilon > 0$,

$$\mathbb{P}\left[\left|\frac{1}{m}\sum_{i=1}^{m}\theta_i - \mu\right| > \epsilon\right] \leq 2\exp\left(\frac{-2m\epsilon^2}{(b-a)^2}\right).$$

***Proposition* 3.2.1.** *Finite Classes Enjoy Uniform Convergence.* Let $\mathcal{H}$ be a finite hypothesis class and let the loss function $\ell : \mathcal{H} \times \mathcal{Z} \to [0, 1]$. Then $\mathcal{H}$ has the uniform convergence property with sample complexity

$$m_{\mathcal{H}}^{UC}(\epsilon, \delta) \leq \left\lceil \frac{\ln(2|\mathcal{H}|/\delta)}{2\epsilon^2} \right\rceil.$$

Proof. Let S be a sample of m i.i.d. examples. The probability that S is not ϵ-representative is bounded using the Union Bound:

$$\mathbb{P}\left(\exists h \in \mathcal{H}, |L_S(h) - L_{\mathcal{D}}(h)| > \epsilon\right) \leq \sum_{h\in\mathcal{H}} \mathbb{P}(|L_S(h) - L_{\mathcal{D}}(h)| > \epsilon).$$

For each fixed h, $L_S(h)$ is an average of m i.i.d. variables $\ell(h, z_i)$, each with mean $L_{\mathcal{D}}(h)$ and bounded in $[0, 1]$. By Hoeffding's Inequality, each term in the sum is at most $2\exp(-2m\epsilon^2)$. Thus,

$$\mathbb{P}(\text{S is not } \epsilon\text{-representative}) \leq \sum_{h\in\mathcal{H}} 2\exp(-2m\epsilon^2) = 2|\mathcal{H}|\exp(-2m\epsilon^2).$$

Setting this bound to be at most δ and solving for m gives the result. ■

Corollary. *Finite Classes are Agnostic PAC Learnable.* For a finite hypothesis class $\mathcal{H}$ and a loss function ℓ with range $[0, 1]$, the ERM algorithm is an agnostic PAC learner with sample complexity

$$m_{\mathcal{H}}(\epsilon, \delta) \leq m_{\mathcal{H}}^{UC}(\epsilon/2, \delta) \leq \left\lceil \frac{2\ln(2|\mathcal{H}|/\delta)}{\epsilon^2} \right\rceil.$$

Proof. Follows directly from Lemma 3.2.1 and Proposition 3.2.1 by setting the representativeness requirement to $\epsilon/2$. ■

This result solidifies the analogy between uninformed search and learning. The sample complexity, which dictates the amount of "experience" needed to learn, is logarithmic in the size of the hypothesis space, $|\mathcal{H}|$. This is the learning-theoretic counterpart to the exponential time and space complexity, $O(b^d)$, required for uninformed state-space search. Both demonstrate that while finite domains are tractable in principle, exhaustive enumeration is inefficient. Without further guidance—heuristics in search, or inductive bias in learning—the complexity can become prohibitive. The need to overcome this limitation by incorporating problem-specific guidance is the central motivation for the informed search strategies explored in the next chapter.

Bibliography

[1] Turing AM. Computing machinery and intelligence. [Internet]. *Mind*. 1950;59(236):433–460. [cited 2025 Jan 20]. Available from: https://doi.org/10.1093/mind/LIX.236.433.

[2] Samuel AL. Some studies in machine learning using the game of checkers. [Internet]. *IBM Journal of Research and Development*. 1959;3(3):210–229. [cited 2025 Jan 20]. Available from: https://doi.org/10.1147/rd.33.0210.

[3] Mitchell TM. Machine learning. New York: McGraw-Hill; 1997. (McGraw-Hill Series in Computer Science). [cited 2025 Jan 20]. ISBN: 978-0-07-042807-2. Available from: https://www.cs.cmu.edu/~tom/mlbook.html.

[4] Haugeland J. Artificial intelligence: the very idea. Cambridge, MA: MIT Press; 1985. [cited 2025 Jan 20]. ISBN: 978-0-262-58095-3. Available from: https://doi.org/10.7551/mitpress/1170.001.0001.

[5] Russell SJ and Norvig P. Artificial intelligence: a modern approach. 4th ed. Hoboken, NJ: Pearson; 2020. [cited 2025 Jan 20]. ISBN: 978-0-13-461099-3. Available from: https://aima.cs.berkeley.edu/.

[6] GeeksforGeeks. Agents in AI. [Internet]. 2025 [last updated 2025 Aug 14; cited 2025 Jan 20]. Available from: https://www.geeksforgeeks.org/artificial-intelligence/agents-artificial-intelligence/.

[7] Guzik EE and Byrge C and Gilde C. The originality of machines: AI takes the Torrance Test. [Internet]. *Journal of Creativity*. 2023;33(3):100065. [cited 2025 Jan 20]. Available from: https://doi.org/10.1016/j.yjoc.2023.100065.

[8] Shalev-Shwartz S and Ben-David S. Understanding machine learning: from theory to algorithms. Cambridge: Cambridge University Press; 2014. [cited 2025 Jan 20]. ISBN: 978-1-107-05713-5. Available from: https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/understanding-machine-learning-theory-algorithms.pdf.

[9] Valiant LG. A theory of the learnable. [Internet]. In: Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing (STOC); 1984 Apr 30-May 2; Washington, D.C., USA. New York: ACM; 1984. p. 436–445. [cited 2025 Jan 20]. Available from: https://doi.org/10.1145/800057.808710.

[10] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I. Attention Is All You Need. [Internet]. arXiv. 2023. [cited 2025 Jan 24]. Available from: https://arxiv.org/abs/1706.03762.

Appendix A

Appendix: Fun Addons

A.1 Advanced Logic: Completeness, Equality, and Paramodulation

This section explores several advanced topics related to the resolution proof system introduced in §2.8.1, namely the theoretical limits of logical systems and a specialised inference rule for handling equality.

A.1.1 Completeness, Decidability, and Incompleteness

Definition A.1.1. ***Complete Proof System***. A proof system is complete if it can derive any sentence that is logically entailed by the knowledge base. Formally, if KB $\vDash \alpha$, then a complete system guarantees that KB $\vdash \alpha$. Robinson's Completeness Theorem established that resolution refutation is a complete proof system for First-Order Logic.

This property makes FOL **semi-decidable**. If a conclusion is entailed (i.e., a proof exists), a resolution theorem prover will eventually halt and find the proof. However, if the conclusion is not entailed, the prover is not guaranteed to halt; it may search for a proof indefinitely. This contrasts with propositional logic, which is decidable.

The scope of completeness is limited when arithmetic is introduced. Gödel's Incompleteness Theorem states that no consistent and complete proof system can exist for FOL combined with arithmetic (addition and multiplication). For any such system, there will either be true sentences that are not provable (incompleteness) or provable sentences that are not true (inconsistency). The proof relies on the ability of arithmetic to construct self-referential sentences, such as the Gödel-sentence P, which asserts its own unprovability:

$$P = \text{"P is not provable."}$$

If P is true, then the system is incomplete because it cannot prove a true sentence. If P is false, then it must be provable, meaning the system is inconsistent because it can prove a false sentence.

A.1.2 Handling Equality in First-Order Logic

The equality symbol, $=$, is a special predicate with a fixed, intended meaning. One method to handle equality is to add axioms to the knowledge base that define its properties. Equality is an equivalence relation, which requires axioms for reflexivity, symmetry, and transitivity:

1. $\forall x.\ x = x$ (Reflexivity)
2. $\forall x, y.\ x = y \to y = x$ (Symmetry)
3. $\forall x, y, z.\ (x = y \land y = z) \to x = z$ (Transitivity)

However, these are insufficient. The principle of substitution (that equals may be substituted for equals), must also be captured. This requires an axiom schema for every predicate and function symbol in the language, which is highly impractical. For example, for each predicate P:

$$\forall x, y.\ x = y \to (P(x) \leftrightarrow P(y))$$

Due to this inefficiency, specialised inference rules are used instead of an axiomatic approach.

A.1.3 The Paramodulation Rule

Paramodulation is an inference rule that integrates reasoning about equality directly into the resolution process. It allows for the substitution of equal terms within clauses. The general rule is as follows: given two clauses, where one contains an equality statement and the other contains a term that unifies with one side of the equality, a new clause is inferred.

Formally, from clauses $\alpha \lor (s = t)$ and $\beta \lor \gamma[r]$, where $\gamma[r]$ is a literal containing a term r:

$$\frac{\alpha \lor (s = t) \quad \beta \lor \gamma[r]}{(\alpha \lor \beta \lor \gamma[t])\theta} \quad \text{where } \theta = \text{Unify}(s, r)$$

The substitution θ is the most general unifier of s and r. The resulting clause combines the disjuncts from both parent clauses (α and β), applies the substitution θ to them, and includes the modified literal $\gamma[t]\theta$, where the term r has been replaced by t.

Example 1: Simple Substitution Consider the premises $F(x) = B$ and $Q(y) \lor W(y, F(y))$. We can apply paramodulation:

- Clause 1: $F(x) = B$. Here, α is empty, $s = F(x)$, and $t = B$.
- Clause 2: $Q(y) \lor W(y, F(y))$. Here, β is $Q(y)$, the literal $\gamma[r]$ is $W(y, F(y))$, and the term r is $F(y)$.
- Unification: $\text{Unify}(s, r) = \text{Unify}(F(x), F(y))$ yields the substitution $\theta = \{x/y\}$.
- Result: The inferred clause is $(Q(y) \lor W(y, B))\theta$. Applying θ has no effect on this clause, so the result is $Q(y) \lor W(y, B)$.

Example 2: Complex Substitution If the first clause has additional literals, they are carried over into the result. Consider the premises $P(x) \lor (F(x) = B)$ and $Q(y) \lor W(y, F(y))$.

- Clause 1: $P(x) \lor (F(x) = B)$. Here, α is $P(x)$.
- Clause 2 and Unification are the same as in the previous example, with $\theta = \{x/y\}$.
- Result: The inferred clause is $(\alpha \lor \beta \lor \gamma[t])\theta$. Substituting the parts gives $(P(x) \lor Q(y) \lor W(y, B))\theta$. Applying the substitution θ yields the final result: $P(y) \lor Q(y) \lor W(y, B)$.

Paramodulation, combined with resolution, provides a complete proof system for first-order logic with equality.

A.2 Natural Language Processing

Language is the primary medium through which humans exchange information about the world. From a computational perspective, communication is an intentional process involving the production and perception of signs drawn from a shared, conventional system. The study of Natural Language Processing (NLP) aims to enable machines to understand, interpret, and generate human language. This process can be deconstructed into a series of component steps, beginning with the formal structure of language itself.

A.2.1 Formal Grammars and The Chomsky Hierarchy

A formal language is a set of strings, which may be infinite. The rules that specify which strings belong to the language are defined by a **grammar**. A grammar consists of a finite set of rewrite rules operating on a vocabulary of terminal symbols (the words of the language) and non-terminal symbols (syntactic categories). For instance, a simple grammar might include the rule $S \to NP\ VP$, stating that a sentence (S) can be rewritten as a noun phrase (NP) followed by a verb phrase (VP).

Grammatical formalisms can be organised by their expressive power into the Chomsky Hierarchy, which includes four main classes:

- **Type-0 (Recursively Enumerable):** Defined by unrestricted grammars, where rewrite rules are of the form $\alpha \to \beta$, with no constraints on α or β.
- **Type-1 (Context-Sensitive):** Rules are of the form $\alpha A \beta \to \alpha \gamma \beta$, where a non-terminal A can be rewritten as γ only in the context of α and β. The right-hand side must contain at least as many symbols as the left.
- **Type-2 (Context-Free):** The most commonly used formalism in NLP, where the left-hand side of a rule must be a single non-terminal symbol (e.g., $S \to NP\ VP$).
- **Type-3 (Regular):** The most restrictive class, with rules of the form $X \to a$ or $X \to aY$. These grammars define regular languages and are equivalent to finite automata.

Most NLP systems are based on Context-Free Grammars (CFGs) due to their balance of expressive power and computational tractability. A CFG consists of a lexicon of terminal symbols (words) categorised by their part of speech (e.g., `Noun`, `Verb`) and a set of structural rewrite rules.

A.2.2 The Communication Process

Communication can be viewed as an action, or a speech act, where a speaker formulates an utterance to achieve a specific intention in a hearer. This process involves distinct stages for both speaker and hearer.

Speaker: Generation The speaker begins with an **intention**, a specific proposition they wish the hearer to know or an action they want them to perform. This intention is transformed into a linguistic form through **generation**, resulting in a sentence. Finally, **synthesis** converts this sentence into an audible waveform or written text. For example, the intention Know(Hearer, ¬Alive(Wumpus)) might be generated as the sentence "The wumpus is dead."

Hearer: Analysis The hearer's task is to reverse this process, beginning with perception and culminating in the incorporation of the speaker's intent. This analysis pipeline involves several

stages:

1. **Perception:** The physical waveform is converted into a sequence of words.
2. **Syntactic Analysis (Parsing):** The word sequence is analysed to determine its grammatical structure according to the rules of a grammar. The output is typically a parse tree, which represents the hierarchical syntactic relationships between the words, as shown in Figure A.1.
3. **Semantic Interpretation:** The meaning of the utterance is derived from the parse tree. Using the principle of compositional semantics, the meaning of a phrase is constructed from the meanings of its constituent parts. This stage produces a formal representation of the sentence's meaning, often in first-order logic.
4. **Pragmatic Interpretation:** Contextual information is applied to the semantic interpretation to resolve indexicals (e.g., "I", "here", "now") and other context-dependent expressions.
5. **Disambiguation:** Throughout the process, ambiguity must be resolved. The hearer must select the most likely interpretation from multiple candidates at the lexical, syntactic, semantic, and pragmatic levels.
6. **Incorporation:** Finally, the disambiguated, context-aware meaning is incorporated into the hearer's knowledge base.

A.2.3 Syntactic Analysis: Parsing

Parsing is the process of finding a valid parse tree for a string of words given a grammar. The two primary strategies are top-down and bottom-up parsing.

- **Top-down parsing** begins with the start symbol (e.g., S) and applies grammar rules to expand non-terminals, attempting to derive the input sentence.
- **Bottom-up parsing** starts with the input words and applies grammar rules in reverse, successively replacing sequences of symbols with a non-terminal until the start symbol S is reached.

A bottom-up parse for the sentence "the wumpus is dead" is traced in Table A.1.

Table A.1: Trace of a bottom-up parse for "the wumpus is dead".

Nodes	**Subsequence**	**Rule Applied**
the wumpus is dead	`the`	Article → the
Article wumpus is dead	`wumpus`	Noun → wumpus
Article Noun is dead	Article Noun	NP → Article Noun
NP is dead	`is`	Verb → is
NP Verb dead	`dead`	Adjective → dead
NP Verb Adjective	VP Adjective	VP → VP Adjective
NP VP	NP VP	S → $NP\ VP$
S		**Success**

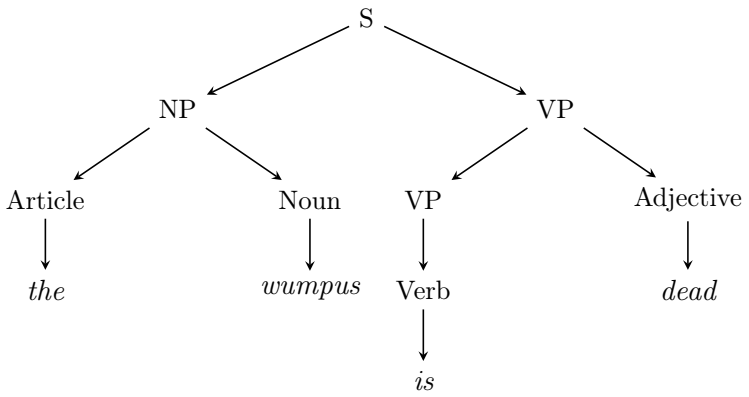


Figure A.1: A parse tree for the sentence "the wumpus is dead", showing its syntactic structure.

A.2.4 Augmented Grammars and Semantic Interpretation

Standard CFGs often overgenerate, producing grammatically incorrect sentences such as "*her loves he". To handle linguistic constraints like subject-verb agreement or pronoun case, grammars can be augmented with features. For example, an NP non-terminal can be parameterised as NP(case), where the case can be subjective or objective.

Definite Clause Grammars This approach is formalised in Definite Clause Grammars (DCGs), where grammar rules are written as definite clauses in first-order logic. For example, the rule $S \rightarrow NP\ VP$ can be written as:

$$NP(s_1) \wedge VP(s_2) \Rightarrow S(s_1 + s_2)$$

Here, s_1 and s_2 are string representations, and parsing becomes a process of logical inference. Top-down parsing corresponds to backward chaining from the goal S(input_string), while bottom-up parsing is a form of forward chaining.

Semantic Interpretation with DCGs DCGs can be further augmented to perform semantic interpretation during the parse. Each grammar rule is associated with a semantic attachment that specifies how to compute the meaning of the construction from its parts. This is often achieved using lambda calculus. For instance, the rule for a transitive verb phrase can be written as:

$$VP(\text{rel}(\text{obj})) \rightarrow \text{Verb}(\text{rel})\ NP(\text{obj})$$

This rule states that the meaning of a verb phrase is derived by applying the verb's meaning (a relation, rel) to the noun phrase's meaning (an object, obj). A full parse tree with semantic attachments is shown in Figure A.2.

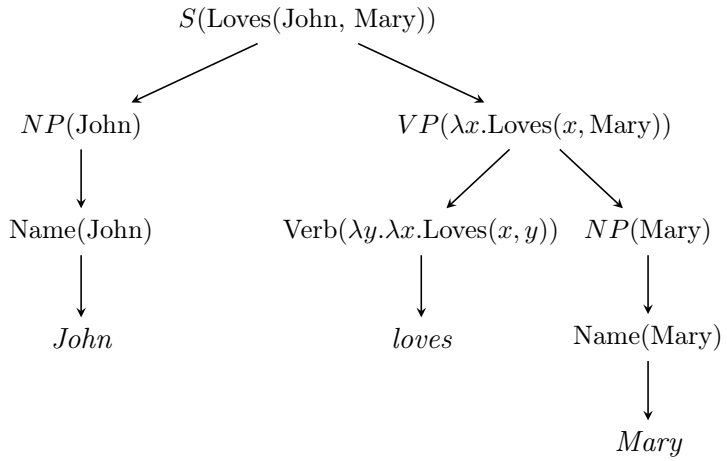


Figure A.2: A parse tree showing compositional semantic interpretation. The lambda expression for "loves" is applied to "Mary" to form the VP's meaning, which is then applied to "John" to form the sentence's meaning.

A.2.5 Ambiguity and Discourse

A central challenge in NLP is ambiguity, which occurs at multiple levels:

- **Lexical:** A single word has multiple meanings (e.g., "interest" as a financial term or a state of curiosity).
- **Syntactic:** A sentence has multiple valid parse trees. In "I smelled a wumpus in 2,2", the prepositional phrase "in 2,2" could modify "smelled" (the act of smelling occurred in 2,2) or "wumpus" (the wumpus is located in 2,2).
- **Semantic:** A phrase has multiple meanings even if its structure is unambiguous (e.g., "the IBM lecture" could be a lecture about IBM or one given by IBM).
- **Pragmatic:** The intended meaning depends on context beyond the sentence itself.

Disambiguation involves choosing the most likely interpretation given the situational context and world knowledge.

When processing multi-sentence discourse, an agent must also perform reference resolution to interpret pronouns and definite noun phrases (e.g., in "John flagged down the waiter. He ordered a sandwich," "He" refers to John). Coherent discourse is structured by coherence relations (e.g., explanation, cause, exemplification) that link sentences together, guiding the interpretation of the overall text.

A.3 Planning as Satisfiability: SATPlan

An alternative to search-based planning algorithms like Graphplan is to translate a planning problem into a Boolean satisfiability (SAT) problem. This approach, known as SATPlan, constructs a propositional logic sentence that has a satisfying assignment if and only if a valid plan of a specific length exists. The planning process then becomes an iterative search over plan lengths, using a SAT solver at each step.

The core idea is to create a set of propositional variables and a set of clauses that encode the entire planning problem up to a fixed time horizon, n.

Variables The translation requires two types of propositional variables:

- **Proposition Variables:** For every proposition P in the planning domain and every even time step $t \in \{0, 2, \ldots, n\}$, a variable P_t is created. P_t is true if proposition P holds at time t.
- **Action Variables:** For every action A and every odd time step $t \in \{1, 3, \ldots, n-1\}$, a variable A_t is created. A_t is true if action A is executed at time t.

Clauses A large conjunctive sentence is constructed from several types of clauses that constrain the possible assignments to these variables.

1. **Initial State:** For each proposition P that is true in the initial state, a unit clause P_0 is added. For each proposition Q that is false, a clause $\neg Q_0$ is added. Unlike Graphplan, which can be agnostic about unmentioned initial propositions, SATPlan requires a complete initial state specification.
2. **Goal State:** For each proposition G in the goal, a unit clause G_n is added (assuming G must be true).
3. **Action Axioms:** For each action A and each odd time step t, an axiom is created to link the action to its preconditions and effects. This takes the form of an implication:

   $$A_t \rightarrow (\text{Preconditions}_{t-1} \wedge \text{Effects}_{t+1})$$

   For example, $\text{Cook}_1 \rightarrow (\text{Clean}_0 \wedge \text{Dinner}_2)$. This is converted into a set of clauses, such as $(\neg\text{Cook}_1 \vee \text{Clean}_0)$ and $(\neg\text{Cook}_1 \vee \text{Dinner}_2)$.
4. **Explanatory Frame Axioms:** To address the frame problem, these axioms specify the only ways a proposition's truth value can change. For each proposition P, an axiom states that if P changes from true to false (or vice-versa) between two time steps, then one of the actions that could have caused this change must have occurred. For instance:

   $$(P_{t-1} \wedge \neg P_{t+1}) \rightarrow (A_t^1 \vee A_t^2 \vee \ldots)$$

   Here, $A^1, A^2, \ldots$ are all the actions that have $\neg P$ as an effect. The contrapositive of this axiom ensures that if none of those actions occur, the proposition does not change, thereby serving as a frame axiom.
5. **Conflict Exclusion Axioms:** To prevent the simultaneous execution of conflicting actions, mutex constraints are added. Two actions conflict if one's precondition is inconsistent with another's effect. For every pair of conflicting actions A and B and every time step t, a clause is added:

   $$\neg A_t \vee \neg B_t$$

The conjunction of all these clauses forms a single SAT problem. If a satisfying assignment is found, the action variables that are set to true constitute the plan. If not, the time horizon n is increased, and a new, larger SAT problem is generated and solved. While this approach generates very large formulae, modern SAT solvers like DPLL and WalkSAT can often solve them efficiently. The performance can be further improved by using domain-specific heuristics to guide the solver, for example, by prioritising branching on action variables within DPLL.

A.4 Planning Under Uncertainty

Classical planning assumes a deterministic world with complete knowledge of the initial state. When these assumptions are violated, more sophisticated techniques are required.

A.4.1 Conditional Planning

Conditional planning addresses uncertainty about the state of the world, particularly the initial state. The planner constructs a plan with branches, where the branch taken during execution depends on information gathered at runtime.

Consider an agent planning to fly from an airport. The agent does not know the departure gate initially but knows it can find out by reading a display in the lobby. The operators would include an information-gathering action like `ReadGate`, which has the precondition `AtLobby` and the effect `KnowWhether(Gate1)`. This effect does not change the physical world but alters the agent's knowledge state.

A conditional planner, such as a modified Partial-Order Planning (POP) algorithm, can incorporate such actions. When an information-gathering action is added to the plan, the plan splits into multiple branches, one for each possible outcome. For the airport example, after `ReadGate`, the plan would have a branch for the context `Gate1` and another for ¬`Gate1`. Subsequent planning steps, like `GoToGate1` and `BoardPlane1`, are placed within the appropriate context. This approach significantly increases the search space and complexity, making it impractical for all but small problems.

A.4.2 Replanning and Execution Monitoring

An alternative to constructing complex conditional plans is to create a simple, linear plan assuming determinism and then monitor its execution. This approach, known as replanning, is useful for handling execution errors or for deferring planning until necessary information is available.

The agent operates in a cycle:

1. **Plan:** Generate a plan from the current state to the goal.
2. **Execute:** Begin executing the plan, one action at a time.
3. **Monitor:** After each action, observe the resulting state of the world and compare it to the expected state.
4. **Replan:** If there is a discrepancy (an execution failure), halt execution, update the current state with the observed state, and return to step 1.

This strategy is more flexible than conditional planning as it does not attempt to anticipate all possible contingencies. Its main drawback is the computational cost of planning, which may be prohibitive in time-critical domains.

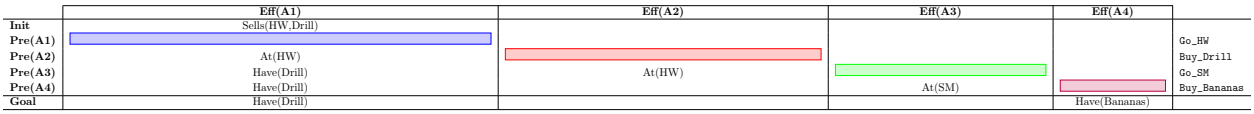
A.4.3 Intermediate Approaches: Universal Plans and Triangle Tables

The trade-off between pre-computation and online computation gives rise to a spectrum of strategies, from replanning (all computation is online) to universal planning (all computation is offline).

Universal Plan A universal plan is an extreme approach where a complete policy is computed offline. It is a mapping from every possible state to the optimal action to take in that state. During execution, the agent simply observes the current state, looks up the corresponding action in a pre-computed table, and executes it. This requires no online deliberation but assumes the world is fully observable and that the state space is small enough to be enumerated and stored, which is rarely feasible.

Triangle Tables An intermediate approach, developed by Fikes and Nilsson for the STRIPS planner, uses a data structure called a triangle table to create a more robust execution strategy for a single, linear plan. A triangle table, illustrated in Figure A.3, organises a plan's actions and the causal links between them.

Figure A.3: A Triangle Table for a shopping plan.



Note. Each row corresponds to an action and its preconditions. Each column lists the effects of the action at the top. The coloured bars represent different kernels.

The table is a lower-triangular matrix where rows correspond to actions in the plan, and columns represent the effects of those actions. An entry in cell (i, j) contains a proposition that is a precondition of action A_i and an effect of action A_j. The execution rule is to find the **highest true kernel** and execute its associated action. A kernel is a rectangular block of conditions anchored at the bottom-left of the table.

For instance, the kernel for the final action, `Buy_Bananas`, consists of all its preconditions (`Have(Drill)`, `At(SM)`). If this set of conditions is true, the agent executes `Buy_Bananas`, regardless of how that state was reached. If not, it checks the next highest kernel (e.g., for `Go_SM`). This strategy allows the agent to skip steps if their effects are already achieved serendipitously or to repeat steps if their effects are undone, providing greater flexibility than rigid linear execution without the complexity of full conditional planning. If even the initial conditions for the first action are not met, the table execution fails, and a full replan is required.

A.5 Representing Uncertainty

While formal logic, as discussed in §2.8.1, provides a rigorous framework for reasoning, its primary mechanism for representing uncertainty is disjunction. An agent can express uncertainty by stating that the world is in one of a set of possible states (e.g., BoxIsRed $\vee$ BoxIsBlue), but it cannot quantify the relative likelihood of these states. For an agent to make rational decisions in complex environments, a more nuanced representation of uncertainty is required. Probability theory provides a quantitative language for encoding and reasoning with degrees of belief.

A.5.1 Interpretations of Probability

There are two principal philosophical interpretations of what a probability value represents.

The Frequentist View The traditional, or frequentist, interpretation defines probability as a statement about the long-run frequency of an event. A statement like "P(Heads) = 0.5" is taken to mean that in a long series of coin flips, the proportion of heads will converge to 0.5. Probability is seen as an objective property inherent in the random process, and its value is estimated from repeated measurements. This view becomes problematic when applied to unique, non-repeatable events. For example, assigning a frequentist probability to the proposition "the sun will come up tomorrow" is difficult, as "tomorrow" is a unique event for which there are no repeated trials.

The Subjectivist (Bayesian) View An alternative, more suited to artificial intelligence, is the subjectivist or Bayesian interpretation. Here, probability is a model of an agent's personal degree of belief in a proposition. A statement $P(A) = 0.8$ reflects the agent's confidence in the truth of A, given its current knowledge. Under this view, beliefs cannot be objectively "wrong" in the same way a frequentist estimate can be; they are private to the agent. However, an agent's beliefs can be inconsistent. The axioms of probability provide a framework for ensuring that an agent's degrees of belief are coherent. This coherence can be justified pragmatically through the Dutch Book argument, which shows that an agent with inconsistent beliefs can be exploited in a series of bets, guaranteeing a loss.

A.5.2 The Axioms of Probability

Probability theory is built upon a small set of axioms that govern the assignment of probabilities to events. The framework begins with a universe U of mutually exclusive and exhaustive atomic events, which represent the possible states of the world, analogous to interpretations in logic.

Definition A.5.1. ***Event***. An **event** is any subset of U.

Definition A.5.2. ***Probability Function***. A probability function P maps events to the interval $[0, 1]$ and must satisfy the following axioms:

1. $P(U) = 1$ and $P(\emptyset) = 0$. The probability of the universe (a tautology) is 1, and the probability of the empty set (a contradiction) is 0.
2. For any event A, $0 \leq P(A) \leq 1$.
3. For any two events A and B, the probability of their union is given by:

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

This third axiom, illustrated in Figure A.4, corrects for double-counting the atomic events in the intersection of A and B. From these axioms, all other laws of probability, such as $P(\neg A) = 1 - P(A)$, can be derived.

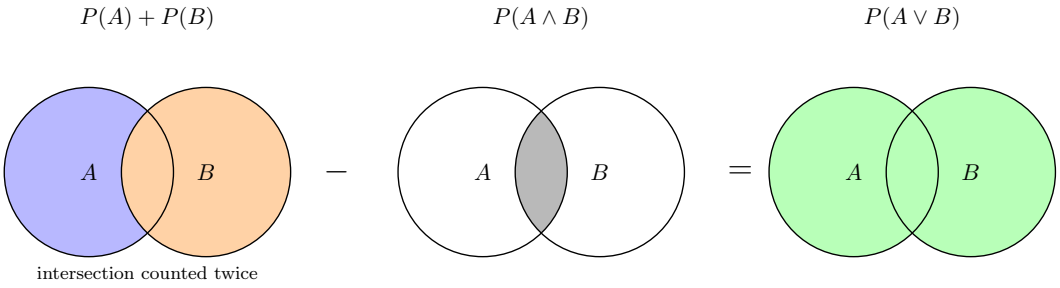


Figure A.4: Area intuition for the addition rule. Summing $P(A)$ and $P(B)$ counts the overlap twice; subtracting $P(A \wedge B)$ leaves exactly the union area, $P(A \vee B)$.

A.5.3 Random Variables and Joint Distributions

In practice, it is convenient to work with **random variables**, which can be thought of as functions mapping outcomes in the universe to a domain of values. For our purposes, we will primarily consider propositional random variables, which take Boolean values. For a variable `Cavity`, the statement $P(\texttt{Cavity} = \text{true}) = 0.1$ assigns a probability to one of its possible values.

Definition A.5.3. ***Joint Probability Distribution***. When a domain involves multiple variables, the complete specification of the probability distribution is the joint probability distribution. The joint distribution assigns a probability to every possible atomic event, defined by a complete assignment of values to all variables. For two variables, `Cavity` and `Toothache`, the joint distribution is shown in Table A.2.

Table A.2: A joint probability distribution for the propositional variables `Cavity` and `Toothache`. The four cells represent the four mutually exclusive atomic events, and their probabilities sum to 1.

	`Toothache`	`¬Toothache`
`Cavity`	0.04	0.06
`¬Cavity`	0.01	0.89

The joint distribution is fundamental because it allows any probabilistic query about the domain to be answered through a process of inference by enumeration. For example, the marginal probability of a single variable can be calculated by summing over all other variables. From Table A.2, the probability of having a cavity is:

$$P(\texttt{Cavity}) = P(\texttt{Cavity} \wedge \texttt{Toothache}) + P(\texttt{Cavity} \wedge \neg\texttt{Toothache}) = 0.04 + 0.06 = 0.1$$

A.6 Conditional Probability and Independence

A.6.1 Conditional Probability and Bayes' Rule

Conditional probability allows an agent to update its beliefs in light of new evidence. The conditional probability of A given B, denoted $P(A|B)$, is the probability of A in the subset of the universe where B is known to be true. It is defined as:

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}, \quad \text{provided } P(B) > 0$$

Using the joint distribution in Table A.2, we can compute the probability of a cavity given a toothache:

$$P(\texttt{Cavity}|\texttt{Toothache}) = \frac{P(\texttt{Cavity} \wedge \texttt{Toothache})}{P(\texttt{Toothache})} = \frac{0.04}{0.04 + 0.01} = \frac{0.04}{0.05} = 0.8$$

Rearranging the definition of conditional probability yields the product rule, $P(A \wedge B) = P(A|B)P(B)$, which leads directly to **Bayes' Rule**:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Bayes' rule is central to diagnostic reasoning. It allows an agent to compute the probability of a cause (e.g., a disease) given an effect (e.g., a symptom) using the causal probability $P(\text{Effect}|\text{Cause})$. This is valuable because causal knowledge is often more stable and easier to acquire than diagnostic knowledge. The term $P(B)$ in the denominator can be computed by conditioning on A: $P(B) = P(B|A)P(A) + P(B|\neg A)P(\neg A)$.

A.6.2 Independence and Conditional Independence

The primary challenge of probabilistic reasoning is that the size of the joint distribution grows exponentially with the number of variables. The concepts of independence and conditional independence are crucial for creating compact representations of probability distributions and enabling efficient inference.

Definition A.6.1. ***Independent Variables***. Two variables A and B are independent if knowing the value of one provides no information about the value of the other. This is formally expressed in three equivalent ways:

$$\begin{aligned} P(A \wedge B) &= P(A)P(B) \\ P(A|B) &= P(A) \\ P(B|A) &= P(B) \end{aligned}$$

Absolute independence is rare. A more common and powerful concept is conditional independence.

Definition A.6.2. ***Conditional Independence***. Two variables A and B are conditionally independent given a third variable C if, once the value of C is known, B provides no additional information about A:

$$P(A|B, C) = P(A|C)$$

For example, a `Toothache` and a `Spot in X-ray` are not independent. However, they can be considered conditionally independent given the presence of a `Cavity`. The cavity is the common

cause; once it is known to be present (or absent), the two symptoms become independent of each other. This structure allows for the decomposition of complex probabilistic relationships. For instance, when updating the belief in a cavity given both symptoms, the conditional independence assumption simplifies the calculation:

$$\begin{aligned} P(C|T,X) &\propto P(T,X|C)P(C) \quad \text{(Bayes' Rule)} \\ &= P(T|C)P(X|C)P(C) \quad \text{(Conditional Independence)} \end{aligned}$$

This decomposition, where multiple effects are assumed to be independent given a common cause, is the basis of the Naïve Bayes model and is a foundational principle for the more complex Bayesian networks used throughout probabilistic AI.

A.7 Large Language Models (LLMs)

The dominant paradigm in modern Natural Language Processing is the use of large, pre-trained generative models. These models, exemplified by LLMs, learn the statistical properties of language from vast corpora of text, enabling them to generate coherent and contextually relevant new text. Their architecture is almost universally based on the Transformer model [10], which replaced the recurrent structures of earlier sequence models with a mechanism based entirely on attention.

A.7.1 The Transformer Architecture

Most LLMs are autoregressive, meaning they generate text one token at a time, conditioning each new token on the sequence of previously generated tokens. The core task is to model the conditional probability distribution over a vocabulary V for a sequence of tokens $\mathbf{x} = (x_1, \ldots, x_T)$. The probability of the entire sequence is factorised as a product of conditional probabilities:

$$P(\mathbf{x}) = \prod_{t=1}^{T} P(x_t | x_1, \ldots, x_{t-1}; \theta),$$

where θ represents the model's parameters.

While training is fully parallel using a causal mask, inference is necessarily sequential. During pre-training, the objective is to maximise the log-likelihood of the training data, which corresponds to minimising the negative log-likelihood loss function:

$$\mathcal{L}(\theta) = -\sum_{t=1}^{T} \log P(x_t | x_1, \ldots, x_{t-1}; \theta)$$

The Transformer architecture achieves this by processing the entire input context in parallel using self-attention mechanisms. As the model has no inherent sense of sequence order, this information is supplied via positional encodings, which are added to the input embeddings. These encodings were originally defined using sine and cosine functions:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position, i is the dimension, and d_{model} is the embedding dimension. Modern decoder-only LLMs, however, commonly use rotary position embeddings (RoPE) rather than absolute sine/cosine encodings; see Figure A.5.

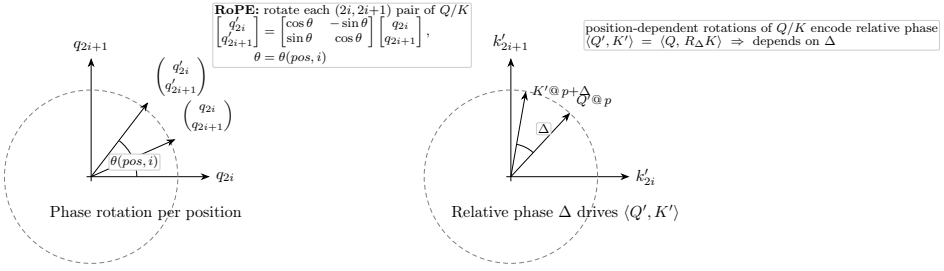


Figure A.5: RoPE rotates each $(2i, 2i{+}1)$ pair of Q/K on a phase circle by an angle $\theta(pos, i)$, preserving norms. The attention score $\langle Q', K' \rangle$ depends only on the *relative* phase Δ between positions, not on absolute positions.

Scaled Dot-Product Attention The fundamental building block of the Transformer is the attention mechanism. It operates on three inputs derived from the input sequence embeddings: queries (Q), keys (K), and values (V). These are matrices obtained by multiplying the input embedding matrix X by learned weight matrices W^Q, W^K, W^V. The attention function computes a weighted sum of the values, where the weight assigned to each value is determined by the compatibility of its corresponding key with a given query. The scaled dot-product attention, illustrated in Figure A.6, is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Here, d_k is the dimension of the key vectors. The scaling factor $\frac{1}{\sqrt{d_k}}$ is crucial for stabilising gradients during training.

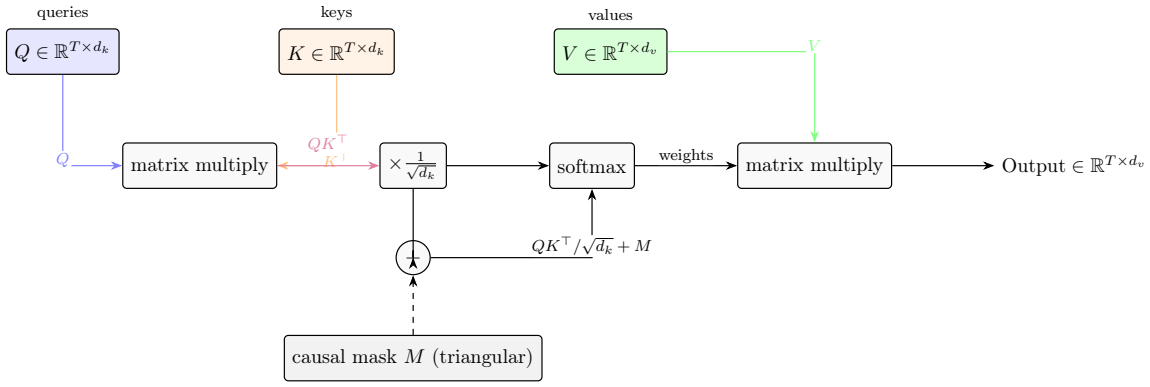


Figure A.6: Scaled dot-product attention. Scores $QK^\top$ are scaled by $1/\sqrt{d_k}$, masked with a causal M, normalised by softmax, and used to weight V.

Multi-Head Attention To allow the model to focus on different aspects of the input sequence, the Transformer employs multi-head attention. This involves running the scaled dot-product attention mechanism multiple times in parallel. The Q, K, and V matrices are linearly projected into different subspaces for each "head".

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

where each head is computed as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

The outputs of the heads are concatenated and projected back to the original dimension via a final weight matrix W^O. A standard Transformer layer combines a multi-head attention sub-layer with a position-wise feed-forward network (FFN), using residual connections and layer normalisation around each sub-layer.

A.7.2 Architectural Variants and Pre-training

While the original Transformer contained both an encoder and a decoder, many modern LLMs use only one of these components.

- **Decoder-only** models, such as the GPT series, are inherently suited for autoregressive generation. They use a causal or unidirectional attention mask, where the attention matrix A is

forced to be lower-triangular ($A_{ij} = 0$ for $j > i$). This ensures that the prediction for a token at position i can only depend on previous tokens.

- **Encoder-only** models, like BERT, use a bidirectional attention mechanism, allowing each token to attend to all other tokens in the sequence. They are typically not used for generation but are powerful for natural language understanding tasks. They are often pretrained using a Masked Language Modelling (MLM) objective, where a fraction of input tokens are masked and the model must predict them based on the surrounding context: $\mathcal{L}_{\text{MLM}} = -\sum_{i \in \text{masked}} \log P(x_i | \mathbf{x}_{\setminus i})$.

A.7.3 Mixture of Experts

To scale models to trillions of parameters without a commensurate increase in computational cost per inference, the Mixture of Experts (MoE) architecture replaces dense feed-forward network (FFN) layers with sparse MoE layers. An MoE layer comprises a set of K independent "expert" networks and a gating network, or router, that dynamically selects which experts process each token. The output for an input token x is a weighted sum of the outputs from the selected experts:

$$y = \sum_{k \in \text{Top-}m(x)} g_k(x) \cdot E_k(x)$$

Here, $E_k(x)$ is the output of the k-th expert, and $g_k(x)$ is the gating probability from the router, typically computed as $g(x) = \text{softmax}(W_g x)$. For sparsity, only the top-m experts (e.g., $m = 2$) with the highest gating probabilities are activated. To prevent routing collapse, an auxiliary load-balancing loss is added, and routing mechanisms often use a capacity factor to manage token overflow.

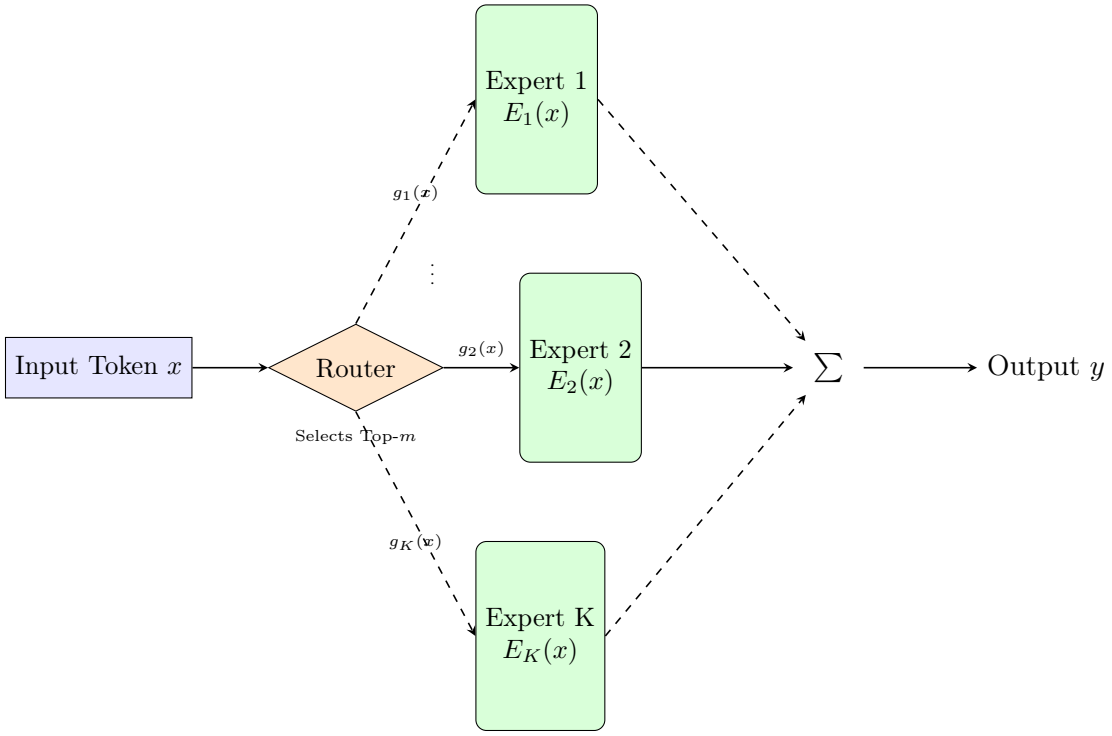


Figure A.7: An MoE layer. The router directs the input token to a sparse subset of experts (e.g., Expert 2 is chosen), and their outputs are combined to produce the final result.

A.7.4 Model Adaptation and Optimisation

Fine-Tuning After pre-training on a general corpus, LLMs are adapted to specific tasks through fine-tuning. This involves further training on a smaller, task-specific dataset. Parameter-Efficient Fine-Tuning (PEFT) methods reduce computational cost. Low-Rank Adaptation (LoRA) is a popular PEFT technique that freezes the pre-trained weights $W \in \mathbb{R}^{k\times d}$ and injects trainable low-rank matrices $A \in \mathbb{R}^{r\times d}$ and $B \in \mathbb{R}^{k\times r}$, modelling the update as $\Delta W = BA$ with $r \ll \min(k, d)$. Only B and A are updated during fine-tuning.

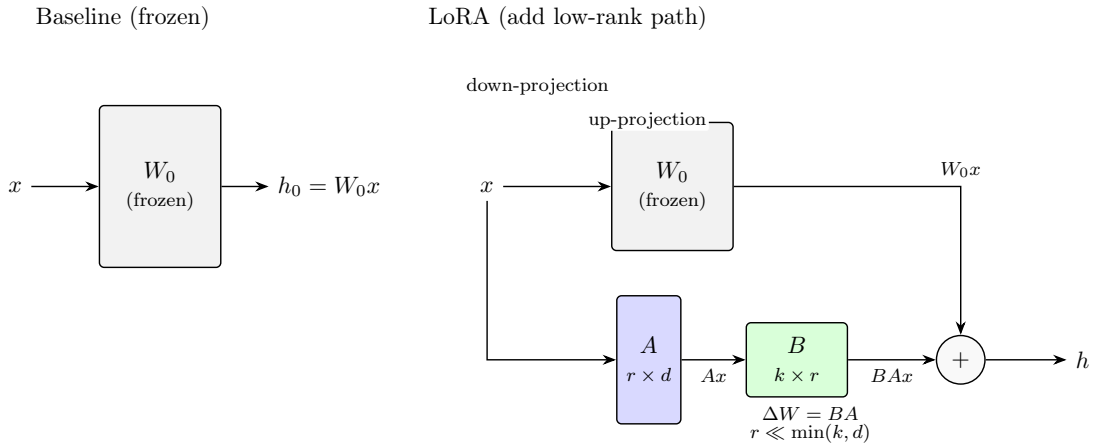


Figure A.8: The LoRA method. The output of the frozen pre-trained weight matrix W_0 is combined with the output of a low-rank adaptation path formed by trainable matrices A (down-projection) and B (up-projection).

Quantisation Quantisation reduces the memory footprint and computational requirements of LLMs by representing weights and activations with lower-precision numerical formats. Post-Training Quantisation (PTQ) converts a trained model's weights by applying a mapping function. Practical setups use clipping and an optional zero-point; for LLMs, Quantisation-Aware Training (QAT) and 4-bit formats like QLoRA (using NF4 with per-group scales) are widely used for a better trade-off between efficiency and performance.

Alignment To ensure LLMs behave in accordance with human values, they undergo alignment. Reinforcement Learning from Human Feedback (RLHF) involves training a reward model on human preferences, then using this model to fine-tune the LLM policy π_θ with an algorithm like Proximal Policy Optimisation (PPO). The objective maximises reward while penalising deviation from the original policy via a KL-divergence constraint: $D_{KL}(\pi_\theta||\pi_{\text{ref}}) \leq \epsilon$. Direct Preference Optimisation (DPO) offers a more direct method by deriving a loss function from preference data, bypassing the explicit reward model. The objective is to increase the relative likelihood of preferred responses (y_w) over rejected ones (y_l):

$$\mathcal{L}_{\text{DPO}}(\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right]$$

where σ is the sigmoid function and β is a temperature parameter.

Retrieval-Augmented Generation (RAG) To mitigate hallucination and provide access to timely or domain-specific information, RAG enhances the generation process by first retrieving

documents from an external knowledge base. Given a query, a retriever module finds relevant text chunks, often using dense embeddings (e.g., from sentence transformers) and cosine similarity search, sometimes in combination with sparse methods like BM25.

$$\text{sim}(q, d) = \frac{\mathbf{q} \cdot \mathbf{d}}{\|\mathbf{q}\| \|\mathbf{d}\|}$$

The retrieved documents are then concatenated with the original prompt and fed into the LLM, which generates a response grounded in the provided context.