CS2AI Artificial Intelligence

Problem Solving & Reinforcement Learning

Lecture Notes

Table of Contents

Session 1: Rational Agents

1.1 Introduction to Intelligent Agents

Artificial intelligence research is fundamentally concerned with building agents that act rationally in their environments. An agent, in the broadest sense, is anything that can perceive its environment through sensors and act upon that environment through actuators. The word itself derives from the Latin agere, meaning 'to do', which captures the essence of agency: the capacity for action. Human beings are agents whose sensors include eyes, ears, and other sensory organs, and whose actuators include hands, legs, and vocal apparatus. A robotic agent might use cameras and infrared range finders as sensors, and various motors as actuators.

The study of intelligent agents provides a unifying framework for the diverse sub-fields of artificial intelligence. Whether we are designing a chess-playing program, a self-driving car, or a recommendation system, we can analyse the problem in terms of the agent's percepts (what it senses), the actions available to it, the environment in which it operates, and the goals it seeks to achieve. This session introduces the foundational definitions, the PEAS framework for specifying task environments, and the five principal agent architectures that underpin modern AI system design.

Key Concept — Intelligent Agent

An intelligent agent perceives its environment through sensors and acts upon that environment through actuators. A rational agent selects actions that are expected to maximise its performance measure, given its percept sequence and any built-in knowledge about the environment.

1.2 The PEAS Framework

Designing an intelligent agent requires a precise specification of its task environment. The PEAS framework provides a structured approach to this specification by identifying four key elements: the Performance measure (an objective criterion for judging the agent's success), the Environment (the external world in which the agent operates), the Actuators (the mechanisms through which the agent affects the environment), and the Sensors (the mechanisms through which the agent perceives the environment). Together, these four elements define the problem that the agent must solve.

Consider, for example, an automated taxi driver. Its performance measure might include safe arrival at the destination, minimising travel time, obeying traffic laws, and maximising passenger comfort. The environment consists of roads, other traffic, pedestrians, and weather conditions. Its actuators include the steering wheel, accelerator, brake, and indicators. Its sensors include cameras, lidar, GPS, and an odometer. Specifying PEAS at the outset ensures that the designer has a clear understanding of what the agent must achieve and the constraints under which it operates.

A rational agent is one that, for each possible percept sequence, selects an action that is expected to maximise its performance measure. Rationality does not imply omniscience — a rational agent need not know the actual outcome of its actions in advance. Instead, it makes the best decision it can

given the information available to it. This distinction is crucial: rationality is about expected performance, not guaranteed perfection. Rational agents also engage in information gathering (performing actions to modify future percepts) and learning from gathered information, unless the environment is fully known and predictable.

1.3 The Five Agent Types

The field of AI identifies five principal agent architectures, distinguished by the mechanism each uses to select actions. These range from the simplest reactive systems to sophisticated learning architectures. Understanding these types is essential because the choice of architecture determines the agent's capabilities, its computational requirements, and the kinds of environments in which it can operate effectively.

1.3.1 Simple Reflex Agents

Simple reflex agents are the most elementary form of intelligent agent. They respond directly to the current percept, ignoring the rest of the percept history. Their decision-making is governed by condition–action rules of the form 'if condition then action'. Because they do not maintain any internal state, simple reflex agents can only succeed in environments that are fully observable — that is, environments where the current percept provides all the information needed to make the correct decision.

Typical use cases for simple reflex agents include environmental monitoring (where a sensor reading directly triggers an alert), quality control and inspection (where a defect triggers a rejection action), and basic process automation and resource allocation. Their simplicity is both their strength and their limitation: they are fast and easy to implement, but they cannot handle partially observable or dynamic environments where past percepts contain relevant information.

1.3.2 Model-Based Reflex Agents

Model-based reflex agents extend the simple reflex architecture by maintaining an internal state that tracks aspects of the world not directly evident in the current percept. This internal model of the world allows the agent to handle partially observable environments. The agent updates its internal state using two kinds of knowledge: knowledge of how the world evolves independently of the agent, and knowledge of how the agent's own actions affect the world.

This architecture is particularly suited to dynamic environments such as robotics and autonomous vehicles, where the agent must reason about the positions and velocities of objects that are not always visible. Game-playing AI systems and large enterprise automation platforms also benefit from model-based reflex architectures, as they must track the state of complex, changing environments.

1.3.3 Goal-Based Agents

Goal-based agents go beyond reflex architectures by incorporating an explicit representation of a desirable goal state. Rather than simply reacting to the current situation, a goal-based agent considers the future consequences of its actions and selects those that will bring it closer to achieving its goal. This requires the agent to engage in search and planning — topics explored in detail in Session 2.

The goal-based approach is especially useful for problem-solving tasks. For instance, a robotics warehouse automation system must plan a sequence of actions (pick, move, place) to fulfil an order. The key components are goal definition, planning (searching for a viable action sequence), action selection, and execution. While more flexible than reflex agents, goal-based agents treat all goal-satisfying outcomes as equally desirable, which can be a limitation when some solutions are preferable to others.

1.3.4 Utility-Based Agents

Utility-based agents address the limitation of goal-based agents by introducing a utility function that maps states to real-valued measures of 'happiness' or desirability. Rather than simply asking 'does this state satisfy my goal?', the utility-based agent asks 'how desirable is this state compared to alternatives?' This allows the agent to make trade-offs between competing objectives and to choose the action that maximises its overall expected utility.

Utility-based agents are well suited to complex, real-world decision-making. Self-driving cars, for example, must balance safety, speed, comfort, and legal compliance — all of which can be captured in a utility function. E-commerce platforms use utility-based reasoning for pricing optimisation and product recommendation, weighing customer satisfaction against profit margins. The utility-based approach requires perception, internal modelling, action generation, outcome prediction, utility assessment, and action selection.

1.3.5 Learning Agents

Learning agents represent the most sophisticated agent architecture. Unlike the previous four types, which operate with a fixed decision-making mechanism, learning agents improve their performance over time through experience. A learning agent consists of four conceptual components: the critic, which evaluates the agent's actions according to a fixed performance standard and provides feedback (for example, in the form of rewards or penalties); the performance element, which selects actions (and may itself be goal-based or utility-based); the learning element, which uses feedback and experience to modify the performance element so that it performs better in the future; and the problem generator, which suggests exploratory actions that may be sub-optimal in the short term but lead to new and informative experiences.

The distinction between learning agents and the other four types is fundamental. Simple reflex, model-based reflex, goal-based, and utility-based agents each describe a particular decision-making mechanism; the learning agent architecture describes a meta-level process that can improve any of these mechanisms. In effect, the performance element of a learning agent may be a simple reflex agent, a model-based agent, or a utility-based agent — the learning element wraps around it and gradually refines it. This makes the learning agent architecture central to reinforcement learning, which is explored in Session 4.

1.4 Comparison of Agent Types

The following table provides a systematic comparison of the five agent types, highlighting the decision mechanism, internal state requirements, and typical use cases for each.

Agent Type	Decision Mechanism	Internal State	Use Cases
Simple Reflex	Condition–action rules applied directly to current percept	None	Environmental monitoring, quality control, basic process automation
Model-Based Reflex	Condition–action rules applied to updated internal state	Internal model tracking unobservable world aspects	Robotics, autonomous vehicles, game-playing AI, enterprise automation
Goal-Based	Search and planning to achieve an explicit goal state	Goal representation and world model	Warehouse automation, navigation, logistics planning
Utility-Based	Maximisation of a utility function over possible outcomes	World model and utility function	Self-driving cars, e-commerce pricing, multi-objective decision-making
Learning	Improvement of performance element through experience and feedback	Critic, learning element, problem generator, plus performance element state	Any domain where the agent must adapt: game playing, robotics, recommendation

1.5 Multi-Agent Systems

In practice, complex tasks often exceed the capabilities of any single agent architecture. Multi-agent systems address this by deploying multiple agents — potentially of different types — that collaborate to solve a problem. Each agent specialises in handling the part of the task for which its architecture is best suited. For example, a logistics system might use a simple reflex agent for real-time sensor monitoring, a goal-based agent for route planning, and a learning agent for demand forecasting.

Multi-agent systems are an active area of research and industrial application. Recent white papers from Google (2025) describe agentic architectures in which multiple AI agents coordinate to handle complex workflows, with each agent contributing specialised capabilities. IBM's research on AI agent use cases and agentic architecture similarly emphasises the power of composing agents into systems that are more capable than any individual agent. Understanding the five agent types is therefore not merely an academic exercise; it is the foundation for designing real-world multi-agent systems.

Key Concept — Multi-Agent Systems

All five agent types (simple reflex, model-based reflex, goal-based, utility-based, and learning) can be deployed together in a multi-agent system. Each agent specialises in the part of the task for which it is best suited, enabling the system as a whole to handle complex tasks that exceed the capabilities of any single agent.

Session 2: Problem Solving by Searching

2.1 Introduction to Problem Solving

Session 1 introduced the concept of a rational agent and the various architectures by which agents select actions. In many real-world scenarios, however, no single action is sufficient to achieve the agent's goal. A problem-solving agent must instead find a sequence of actions — a plan — that leads from the current state to a goal state. The theory and technology of building rational agents that can plan ahead to solve problems forms the core of this session.

The process of looking for a sequence of actions that reaches the goal is called search. A search algorithm takes a problem as input and returns a solution in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out in an execution phase. Search is one of the most fundamental techniques in artificial intelligence, underpinning applications from route planning and puzzle solving to game playing and automated theorem proving.

2.2 Problem Formulation

Before an agent can search for a solution, the problem must be precisely formulated. Problem formulation begins with goal formulation — deciding what constitutes a successful outcome based on the agent's current situation and its performance measure. The problem is then defined by specifying five components:

- Initial state: the state in which the agent starts (e.g., In(Arad)).
- Possible actions: the set of actions available to the agent in a given state (e.g., from In(Arad), the valid actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}).
- Transition model: a description of what each action does, mapping a state and an action to a resulting state (e.g., RESULT(In(Arad), Go(Zerind)) = In(Zerind)).
- Goal test: a function that determines whether a given state is a goal state (e.g., is the state In(Bucharest)?).
- Path cost function: a function that assigns a numeric cost to each path through the state space, typically as the sum of the costs of individual actions along the path (e.g., total distance in kilometres).

These five components implicitly define the state space of the problem: the set of all states reachable from the initial state by any sequence of actions. The state space forms a directed graph in which the nodes are states and the edges are actions. Searching this graph for a path from the initial state to a goal state is the central computational challenge.

Key Concept — Problem Formulation

A search problem is defined by five components: the initial state, the available actions, the transition model, the goal test, and the path cost function. Together, these define the state space — a directed graph of all reachable states — which the search algorithm must explore to find a solution.

2.3 The Romania Navigation Example

A classic example used to illustrate search algorithms is the Romania road map problem, in which the agent must find the shortest route from Arad to Bucharest. The initial state is In(Arad). The actions correspond to driving between neighbouring cities. The transition model maps each drive action to the resulting city. The goal test checks whether the agent has reached Bucharest. The path cost is the total distance in miles along the route.

From the initial state, the search algorithm generates a search tree by expanding nodes — that is, by considering all the actions available from a state and generating the resulting successor states. At depth 1, expanding Arad yields three successors: Sibiu, Timisoara, and Zerind. At depth 2, each of these is expanded in turn, producing further successors. Cycles arise when a node in the tree corresponds to a state already visited along the current path (for example, going from Arad to Sibiu and back to Arad). Detecting and handling these cycles is important for the efficiency and correctness of the search.

The search tree can grow very large — by depth 7, it contains dozens of nodes and multiple paths to Bucharest. The challenge for the search algorithm is to find a solution efficiently without expanding the entire tree. Different search strategies address this challenge in different ways, as discussed in the following sections.

2.4 Uninformed Search Strategies

Uninformed (or blind) search strategies use only the information available in the problem definition — they have no additional knowledge about how close a state is to the goal. The three principal uninformed strategies are depth-first search, breadth-first search, and uniform-cost search.

2.4.1 Depth-First Search (DFS)

Depth-first search always expands the deepest unexpanded node in the search tree. It uses a stack (last-in, first-out) as its data structure, which gives it a low memory requirement: it only needs to store a single path from the root to a leaf, plus the unexpanded sibling nodes along that path. However, DFS can get trapped in infinite loops if cycles are not detected, and it does not guarantee finding the shortest or least-cost solution — it finds the 'leftmost' solution in the search tree, ignoring path costs entirely.

Two important variants address the cycle problem. Depth-limited search imposes a fixed maximum depth, preventing the algorithm from descending indefinitely but at the risk of missing solutions deeper than the limit. Iterative deepening search combines the space efficiency of DFS with the completeness of BFS by running depth-limited searches with progressively increasing depth thresholds.

2.4.2 Breadth-First Search (BFS)

Breadth-first search always expands the shallowest unexpanded node. It uses a queue (first-in, first-out) as its data structure. BFS is complete — it will always find a solution if one exists — and it finds the shallowest solution, which is optimal when all actions have equal cost. However, BFS requires

storing all generated nodes in memory, which can be prohibitive for problems with large branching factors.

2.4.3 Uniform-Cost Search

Uniform-cost search (also known as Dijkstra's algorithm) always expands the node with the lowest cumulative path cost from the initial state. It uses a priority queue ordered by path cost. Unlike BFS, which finds the shallowest solution, uniform-cost search finds the lowest-cost solution, making it optimal for problems where actions have varying costs. The trade-off is increased computational expense, as the algorithm must maintain and sort the priority queue.

In the Sibiu-to-Bucharest sub-problem, for instance, DFS and BFS both find the path Sibiu → Fagaras → Bucharest (310 km) in two expansion steps. Uniform-cost search, by contrast, expands nodes in order of cumulative cost and discovers the cheaper path Sibiu → Rimnicu Vilcea → Pitesti → Bucharest (278 km) in five expansion steps. This illustrates the fundamental trade-off: uninformed strategies that consider costs are more expensive computationally but yield better solutions.

2.5 Informed (Heuristic) Search Strategies

Informed search strategies use additional knowledge — a heuristic function — to guide the search towards the goal more efficiently. A heuristic function h(n) estimates the cost of the cheapest path from node n to a goal state. By incorporating this estimate, informed strategies can dramatically reduce the number of nodes expanded compared to uninformed search.

2.5.1 Greedy Best-First Search

Greedy best-first search always expands the node with the lowest heuristic value h(n), selecting the state it believes is nearest to the goal. While often fast in practice, greedy search is neither complete nor optimal: it can be led astray by misleading heuristics and may fail to find a solution even when one exists. In the Sibiu–Bucharest example, greedy search uses straight-line distances to Bucharest as the heuristic. It expands Sibiu, then selects Fagaras (h = 176) over Rimnicu Vilcea (h = 193), finding the path SFB (310 km) — which is not optimal.

2.5.2 A* Search

A* search is the most widely used informed search algorithm. It evaluates nodes using the function f(n) = g(n) + h(n), where g(n) is the actual cost of the path from the initial state to node n (the backward cost), and h(n) is the heuristic estimate of the cost from n to the goal (the forward cost). By combining these two components, A* balances the desire to explore cheap paths (favoured by uniform-cost search) with the desire to head towards the goal (favoured by greedy search).

In the Sibiu–Bucharest example, A* proceeds as follows. First, it expands Sibiu. The estimated total cost for Fagaras is f = 99 + 176 = 275, while for Rimnicu Vilcea it is f = 80 + 193 = 273. Since 273 < 275, A* expands Rimnicu Vilcea next. After expanding Rimnicu Vilcea, the frontier contains Fagaras (f = 275), Pitesti (f = 80 + 97 + 100 = 277), and the cycle back to Sibiu (f = 353). Pitesti is not yet cheaper than Fagaras, so A* might consider Fagaras, but ultimately upon expanding Pitesti it discovers the path S → Rv → P → B at cost 278. Since 278 > 275, A* must also expand Fagaras to verify no cheaper

path exists through it. The solution through Fagaras costs 310, confirming that S → Rv → P → B (278 km) is optimal. A* thus finds the optimal solution in four expansion steps.

Key Concept — A* Search

A* search evaluates each node using f(n) = g(n) + h(n), combining the actual backward cost g(n) with the heuristic forward estimate h(n). It combines the optimality of uniform-cost search with the speed of greedy search. A* is guaranteed to find the optimal solution provided the heuristic is admissible.

2.6 Admissible Heuristics

A* search is guaranteed to find the lowest-cost path if the heuristic function is admissible. A heuristic h(n) is admissible if, for every node n, h(n) ≤ h*(n), where h*(n) is the true cost to reach the goal from n. In other words, an admissible heuristic never overestimates the true cost — it is always optimistic. Straight-line distance is a classic example of an admissible heuristic for navigation problems: the straight-line distance between two cities never exceeds the actual road distance.

The choice of heuristic has a profound effect on the efficiency of A* search. A more informative (but still admissible) heuristic — one that provides estimates closer to the true cost — causes A* to expand fewer nodes, reducing both time and memory requirements. Designing good heuristics is therefore a critical skill in applying search algorithms to real-world problems.

2.7 Comparison of Search Strategies

The following table summarises the key properties of the search strategies discussed in this session.

Strategy	Data Structure	Completeness	Optimality	Key Property
Depth-First Search	Stack	No (may loop)	No	Low memory; finds leftmost solution
Breadth-First Search	Queue	Yes	Yes (equal costs)	Finds shallowest solution; high memory
Uniform-Cost Search	Priority Queue	Yes	Yes	Finds lowest-cost path; computationally expensive
Greedy Best-First	Priority Queue (by h)	No	No	Fast but can be misled by heuristic
A* Search	Priority Queue (by f = g + h)	Yes	Yes (if h admissible)	Combines speed and optimality; best general-purpose

Session 3: Adversarial Search and Game Theory

3.1 Introduction to Adversarial Search

In the search problems discussed in Session 2, the agent was the sole decision-maker in a benign environment. Adversarial search extends this framework to competitive environments — commonly called games — in which one or more adversarial agents are actively planning against our agent. The key difference is that the outcome depends not only on our agent's actions but also on the actions of the opponent, whose goals are in direct conflict with ours.

Whereas problem solving by search returns a plan (a fixed sequence of actions to achieve a goal), adversarial search returns a strategy or policy: a recommendation for the best possible action given some configuration of agents and adversaries. This distinction is fundamental — in a competitive environment, a fixed plan is useless because the opponent's moves are unpredictable. Mathematical game theory views any multi-agent environment (whether competitive or collaborative) as a game if the impact of each agent on the others is significant.

3.2 Deterministic Zero-Sum Games with Perfect Information

This session focuses on a specific class of games: deterministic, zero-sum games with perfect information. In such games, outcomes are deterministic (no element of chance), one player's gain is exactly equivalent to the opponent's loss (zero-sum), and all game states are fully observable to both players (perfect information). Classic examples include Tic-Tac-Toe, Chess, Checkers, and Go.

By contrast, probabilistic zero-sum games introduce an element of chance: Monopoly and Backgammon have perfect information but random elements (dice rolls), while Scrabble, Poker, and Bridge involve imperfect information (hidden cards or tiles). Non-zero-sum games — such as player-versus-environment games like Pac-Man or team sports — fall outside the scope of this session. The restriction to deterministic zero-sum games with perfect information allows us to develop powerful algorithms (MiniMax and alpha-beta pruning) that guarantee optimal play.

3.3 Game Formulation

A two-player zero-sum game is formulated as a search problem with two competing players, conventionally called MAX and MIN. MAX moves first, and the players alternate turns. At the end of the game, points are awarded to the winner and penalties to the loser. The game is defined by the following components:

- S_0 (initial state): the configuration of the game at the start.
- PLAYER(s): a function that returns which player has the move in state s.
- ACTIONS(s): the set of legal moves available in state s.
- RESULT(s, a): the transition model, returning the state that results from taking action a in state s.
- TERMINAL-TEST(s): a Boolean function that returns true when the game is over. States where the game has ended are called terminal states.

- UTILITY(s, p): a utility (or payoff) function that assigns a numeric value to a terminal state s for player p.

These components define a game tree — a search tree in which the nodes are game states, the edges are moves, and the leaf nodes are terminal states with associated utility values. In the Tic-Tac-Toe example, the game ends when one player has three in a row or all squares are filled. The utility values from MAX's perspective are typically +1 (win), 0 (draw), and −1 (loss).

Key Concept — Game Formulation

A two-player zero-sum game is defined by an initial state, a player function, legal actions, a transition model, a terminal test, and a utility function. MAX seeks to maximise the utility; MIN seeks to minimise it. The resulting game tree is the search space over which adversarial algorithms operate.

3.4 The MiniMax Algorithm

The MiniMax algorithm determines the optimal strategy for MAX by assuming that both players play optimally. The minimax value of a state is the utility that MAX can guarantee, assuming MIN also plays perfectly. For terminal states, the minimax value is simply the utility. For non-terminal states, the value is computed recursively: at MAX nodes, the minimax value is the maximum of the children's values; at MIN nodes, it is the minimum.

The algorithm works by first recursively computing the game tree down to the terminal states, then backing up the minimax values from the leaves to the root. If the tree is finite, MiniMax returns the optimal solution — the move that guarantees the best possible outcome for MAX against an optimal adversary.

3.4.1 MiniMax Worked Example

Consider a two-ply game tree in which MAX is at the root (node A) with three children B, C, and D (MIN nodes). Each MIN node has three children (terminal nodes) with the following utility values:

- Node B's children: 3, 12, 8
- Node C's children: 2, 4, 6
- Node D's children: 14, 5, 2

Since B, C, and D are MIN nodes, each takes the minimum of its children's values: B = min(3, 12, 8) = 3, C = min(2, 4, 6) = 2, and D = min(14, 5, 2) = 2. Node A is a MAX node, so it takes the maximum of its children: A = max(3, 2, 2) = 3. The optimal move for MAX is therefore to choose node B, guaranteeing a utility of 3 regardless of MIN's play.

In the larger group-activity example with nodes A through Z and beyond, the same principle applies recursively through multiple levels. The minimax value of the root node A was found to be −43, with MAX choosing to move to node B. The traversal order visits leaf nodes first (bottom-up), computing minimax values at each internal node as the maximum or minimum of its children's values, depending on which player's turn it is.

3.5 Alpha-Beta Pruning

The MiniMax algorithm has a time complexity of O(b^m), where b is the branching factor (number of legal moves per state) and m is the maximum depth of the tree. For complex games, this is prohibitively expensive. Alpha-beta pruning addresses this limitation by eliminating branches of the game tree that cannot possibly influence the final decision, while still returning the same result as the full MiniMax algorithm.

3.5.1 Alpha-Beta Pruning Step by Step

Consider the two-ply example with nodes A (MAX), B, C, D (MIN), and terminal values [3, 12, 8] under B, [2, 4, 6] under C, and [14, 5, 2] under D. Alpha-beta pruning maintains two values: α (the best value MAX can guarantee so far) and β (the best value MIN can guarantee so far). Initially, α = −∞ and β = +∞.

The algorithm first explores node B. The first child of B has value 3, so B's value is at most 3 (since B is a MIN node, β = 3). The second child has value 12, which is greater than 3 — MIN would avoid this move, so B's value remains at most 3. The third child has value 8, again greater than 3. B's final minimax value is therefore 3. MAX now knows it can guarantee at least 3, so α = 3.

Next, the algorithm explores node C. The first child of C has value 2. Since C is a MIN node, C's value is at most 2. But MAX already has a guaranteed value of 3 from B. Since 2 < 3, MAX would never choose C — there is no need to examine C's remaining children. This is an alpha cutoff: the entire subtree under C is pruned.

The algorithm then explores node D. The first child of D has value 14, so D's value is at most 14. Since 14 > 3 (α), we must continue exploring. The second child has value 5 (D ≤ 5), still above α. The third child has value 2, giving D a final value of 2. Since 2 < 3 (α), MAX would not choose D. The final decision is to move to B with a guaranteed value of 3.

In this example, alpha-beta pruning saved us from examining two of C's three children. In the best case ('perfect ordering', where the best moves are explored first), alpha-beta pruning can effectively double the search depth achievable in the same computation time, since its effective branching factor is reduced from b to approximately √b.

Key Concept — Alpha-Beta Pruning

Alpha-beta pruning is an optimisation of MiniMax that eliminates branches of the game tree that cannot affect the final decision. It maintains two bounds: α (the best value MAX can guarantee) and β (the best value MIN can guarantee). When α ≥ β, the remaining children of the current node are pruned. With perfect move ordering, alpha-beta pruning can effectively double the searchable depth.

3.6 Cutoff Tests and Heuristic Evaluation Functions

Even with alpha-beta pruning, searching to the terminal states of a complex game is infeasible. Chess, for example, has a branching factor of approximately 35 and game trees that extend to roughly 80 moves deep, yielding a state space of approximately 10^150 — vastly exceeding the

estimated 10^80 atoms in the observable universe. In practice, therefore, game-playing programs must cut off the search before reaching terminal states and use a heuristic evaluation function to estimate the utility of the non-terminal states at the search frontier.

The MiniMax or alpha-beta algorithm is modified in two ways. First, the terminal test is replaced by a cutoff test that decides when to stop expanding the tree (typically at a fixed depth). Second, the utility function is replaced by a heuristic evaluation function EVAL that estimates the position's utility. For chess, a simple evaluation function might assign material values (pawn = 1, knight = 3, bishop = 3, rook = 5, queen = 9) and positional values (king safety = 0.5, good pawn structure = 0.5), computing a weighted linear combination normalised to the utility range.

The depth of the cutoff test determines the strength of the program. A 4-ply lookahead (b^4 ≈ 1.5 million states for chess) produces a novice-level player. An 8-ply lookahead corresponds roughly to a human master. Deep Blue, which defeated world champion Garry Kasparov in 1997, could search to 12 or more plies in some positions, sometimes extending to 40-ply searches for critical lines. The combination of deeper search and more sophisticated evaluation functions has driven the remarkable progress of game-playing AI.

Search Depth (Plies)	Approximate States Evaluated (b = 35)	Playing Strength
4	~1.5 million	Novice (hopeless)
8	~2.3 trillion	Human master / typical PC
12+	~10^18+	Grandmaster / Deep Blue level
40 (selective)	Variable	Superhuman in tactical lines

Session 4: Reinforcement Learning

4.1 Sequential Decision Making

The previous sessions explored agents that plan in environments with known models (search) or against adversaries in fully observable games (adversarial search). Reinforcement learning (RL) addresses a fundamentally different setting: an agent must learn to make good decisions through interaction with an initially unknown environment. The agent selects actions, observes the resulting state and reward, and uses this feedback to improve its behaviour over time.

In sequential decision making, the agent's goal is to select a sequence of actions that maximises the total cumulative reward over time. This is not simply about immediate gratification — the agent must balance short-term rewards against long-term consequences. A helicopter performing stunt manoeuvres, for example, receives a reward for following the desired trajectory and a penalty for crashing, but it might also need to refuel (a short-term cost) to prevent a future crash. Similarly, an investment portfolio manager receives rewards for profits but must accept that some investments take months to mature.

Key Concept — The Reward Hypothesis

The reward hypothesis states that all goals can be described as the maximisation of expected cumulative reward. While this assumption has been debated (see the NeurIPS 2022 paper 'Challenging the Reward Hypothesis'), it provides a powerful and practical framework for formulating RL problems.

4.2 Reinforcement Learning vs Planning

There are two fundamental approaches to sequential decision making. In planning (also called deliberation, reasoning, or introspection), a model of the environment is known in advance. The agent performs computations with its model — without any external interaction — to derive an improved policy. The search algorithms of Session 2 are examples of planning: the agent knows the state space, the transition model, and the costs, and it computes an optimal plan before acting.

In reinforcement learning, by contrast, the environment is initially unknown. The agent must interact with the environment, observe the consequences of its actions, and gradually improve its policy through trial and error. There is no pre-built model to consult — the agent must learn from experience. This makes RL more broadly applicable than planning (it works even when the environment is too complex to model), but also more challenging (the agent must explore efficiently and learn from potentially sparse and delayed rewards).

4.3 Components of an RL Agent

An RL agent may possess up to three major components, each serving a distinct role in the agent's decision-making process.

4.3.1 Policy

The policy is the agent's behaviour function — a mapping from states to actions. It defines what the agent does in each state. A policy can be deterministic, written as $a = \pi(s)$, meaning that in state s the agent always takes action a. Alternatively, it can be stochastic, written as $\pi(a|s) = P(A = a \mid S = s)$, meaning that the agent chooses action a with a certain probability. In the maze example from the lecture slides, the deterministic policy is visualised as arrows in each cell indicating the direction the agent should move.

4.3.2 Value Function

The value function (also called the utility function in some texts) maps states to a prediction of future reward. Specifically, $v\pi(s)$ represents the expected cumulative reward starting from state s and following policy π thereafter. The value function enables the agent to evaluate the long-term desirability of states, which can then be used to select better actions. In the maze example, the value function assigns numbers to each cell representing the expected number of steps remaining to reach the goal (with negative values, since each step incurs a reward of −1).

4.3.3 Model

The model is the agent's internal representation of the environment. It predicts what the environment will do next, given the current state and the agent's action. Specifically, the model predicts both the immediate next state and the immediate reward. In the maze example, the agent may have learnt its own internal map of the maze (which cells are walls, which are open) and that each step incurs a reward of −1. The model may be imperfect — the agent's representation of the environment need not match reality exactly.

Key Concept — RL Agent Components

An RL agent's behaviour is governed by up to three components: the policy (a mapping from states to actions), the value function (a prediction of expected future reward from each state), and the model (an internal representation of the environment's dynamics and rewards). Not all components need to be present in every RL agent.

4.4 RL Agent Types

Not all three components (policy, value function, model) need to be present in an RL agent. The choice of which components to include gives rise to several distinct agent types, each with different strengths and trade-offs.

Classification	Agent Type	Components Present	Description
By learning method	Policy-Based	Policy (no value function)	The policy is trained directly without estimating a value function.
By learning method	Value-Based	Value function (implicit	The policy is derived

		policy)	implicitly from the value function by choosing the action that leads to the highest-valued next state.
By learning method	Actor-Critic	Policy + Value function	Both the policy (actor) and the value function (critic) are maintained and trained jointly.
By model usage	Model-Free	No model	The agent learns directly from experience without building an internal model of the environment.
By model usage	Model-Based	Model present	The agent builds and uses an internal model of the environment to plan and improve its policy.

These classifications are orthogonal: a model-free agent can be policy-based, value-based, or actor-critic. Q-learning, which we examine in detail below, is a model-free, value-based method.

4.5 Q-Learning

Q-learning is one of the most important and widely used reinforcement learning algorithms. It is a model-free, value-based method in which the agent learns a Q-function (also called an action-value function) that estimates the expected cumulative reward of taking a given action in a given state and then following the optimal policy thereafter.

4.5.1 The Q-Table

The Q-function is stored in a Q-table — a two-dimensional table indexed by states (rows) and actions (columns). Each entry Q(s, a) represents the agent's current estimate of the value of taking action a in state s. Initially, all Q-values are set to zero (the agent knows nothing). As the agent interacts with the environment and receives rewards, it updates the Q-table entries to reflect its growing knowledge. Once training is complete, the agent can derive its policy by selecting, in each state, the action with the highest Q-value.

4.5.2 The Q-Update Rule

The core of Q-learning is the update rule that refines Q-values after each step. When the agent is in state S_t, takes action A_t, receives reward R_{t+1}, and transitions to state S_{t+1}, the Q-value is updated as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot [R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Each term in this equation has a specific meaning. The learning rate α (alpha) controls how much the new information overrides the old estimate: α = 0 means the agent learns nothing, while α = 1 means the agent completely replaces the old estimate. The discount factor γ (gamma) determines how much the agent values future rewards relative to immediate ones: γ = 0 makes the agent myopic (caring only about immediate reward), while γ = 1 gives future rewards equal weight. The expression R_{t+1} + γ · max_a Q(S_{t+1}, a) is the target — the agent's new estimate of the total value, combining the immediate reward with the discounted value of the best action in the next state. The difference between this target and the current estimate Q(S_t, A_t) is called the temporal difference (TD) error.

Key Concept — Q-Learning Update Rule

Q(S_t, A_t) ← Q(S_t, A_t) + α · [R_{t+1} + γ · max_a Q(S_{t+1}, a) − Q(S_t, A_t)]. The learning rate α controls update magnitude, the discount factor γ balances immediate vs future reward, and the temporal difference (TD) error drives learning by comparing the new estimate to the old.

4.5.3 The Frozen Lake Example

The Frozen Lake environment is a 4×4 grid (16 states, numbered 0–15) with four possible actions: Left, Right, Up, and Down. The agent starts at state 0 (top-left corner) and must reach the goal at state 15 (bottom-right corner), receiving a reward of +1 upon arrival. Some cells contain holes that end the episode. The Q-table has 16 rows (one per state) and 4 columns (one per action), initialised to zero.

To illustrate how Q-values propagate backward from the goal, consider the simplified case with learning rate α = 1 and discount factor γ = 1. With these settings, the update rule simplifies to Q(S_t, A_t) = R_{t+1} + max_a Q(S_{t+1}, a). The first non-zero Q-value arises when the agent discovers the goal: if the agent is in state 14 and moves Right to reach state 15 (the goal), it receives reward +1. Since all Q-values for state 15 are zero, Q(14, Right) = 1 + 0 = 1.

In subsequent episodes, the agent can reach state 14 from state 13 by moving Right. Now Q(13, Right) = 0 + max_a Q(14, a) = 0 + 1 = 1, because the reward for the step itself is 0 (only the goal gives +1) and the best Q-value in state 14 is 1. Similarly, Q(9, Down) = 0 + max_a Q(13, a) = 1, because state 9's downward neighbour is state 13. The pattern continues: Q(8, Right) = 0 + max_a Q(9, a) = 1, Q(4, Down) = 0 + max_a Q(8, a) = 1, and Q(0, Down) = 0 + max_a Q(4, a) = 1.

Over many episodes, Q-values propagate backward from the goal through all states that lead to it. The resulting Q-table encodes an optimal policy: in each state, the agent simply selects the action with the highest Q-value. This backward propagation of value — from goal to start — is the fundamental mechanism by which Q-learning discovers good policies without any prior knowledge of the environment.

4.6 Temporal Difference Learning

Q-learning belongs to the family of temporal difference (TD) methods. The defining characteristic of TD learning is that the agent updates its estimates after every single step of interaction, rather than waiting until the end of an episode. This is in contrast to Monte Carlo methods, which update

estimates only after a complete episode has been observed. TD methods are more efficient in practice because they can learn from incomplete episodes and update incrementally.

The temporal difference error — the discrepancy between the agent's new estimate ($R_{t+1} + \gamma \cdot \max_a Q(S_{t+1}, a)$) and its old estimate ($Q(S_t, A_t)$) — drives the learning process. When the TD error is positive, the agent has received more reward than expected, and the Q-value is increased. When it is negative, the Q-value is decreased. Over time, the TD errors converge to zero as the Q-values approach their true values.

4.7 Off-Policy vs On-Policy Learning

Q-learning is an off-policy algorithm. This means that during training, the agent uses one policy for acting in the environment (the behavioural policy, which typically includes exploration) and a different policy for updating the Q-function (the target policy, which is always greedy with respect to the current Q-values). The separation of behavioural and target policies is what makes Q-learning 'off-policy': the updates are independent of the specific actions the agent actually took during exploration.

The alternative is on-policy learning, exemplified by the SARSA algorithm (State–Action–Reward–State–Action). In SARSA, the same policy that generates the agent's behaviour is also used for updates. Specifically, instead of using $\max_a Q(S_{t+1}, a)$ as Q-learning does, SARSA uses $Q(S_{t+1}, A_{t+1})$, where A_{t+1} is the action actually chosen by the behavioural policy in the next state. On-policy methods tend to be more conservative and safer (they evaluate the policy they are actually following), while off-policy methods like Q-learning can be more sample-efficient because they can learn about the optimal policy even while following an exploratory policy.

Key Concept — Off-Policy vs On-Policy

Q-learning is off-policy: it uses a greedy target policy ($\max_a Q$) for updates regardless of the actions actually taken. SARSA is on-policy: it updates using the action actually selected by the behavioural policy. Off-policy methods can learn the optimal policy while exploring; on-policy methods evaluate the policy being followed.

References and Further Reading

- S. Russell & P. Norvig (2022), Artificial Intelligence: A Modern Approach, 4th edition. Pearson. Chapters 2, 3, 6, and 23. Available online at the University of Reading Library.
- UC Berkeley, Introduction to Artificial Intelligence (online textbook). Chapter 1: "Search", Chapter 3: "Games", Chapter 5: "RL". https://inst.eecs.berkeley.edu/~cs188/textbook/
- D. Silver (2015), Lectures on Reinforcement Learning. https://davidstarsilver.wordpress.com/teaching/
- S. Raschka & V. Mirjalili, Python Machine Learning, Chapter 18: Reinforcement Learning. Available online at the University of Reading Library.
- IBM, "AI Agent Use Cases". https://www.ibm.com/think/topics/ai-agent-use-cases
- IBM, "Agentic Architecture". https://www.ibm.com/think/topics/agentic-architecture
- Google (2025), Agents White Paper. https://www.kaggle.com/whitepaper-agents
- Google (2025), Agents Companion White Paper (multi-agent systems and advanced topics). https://www.kaggle.com/whitepaper-agent-companion
- "Challenging the Reward Hypothesis", NeurIPS 2022. https://neurips.cc/virtual/2022/65594