Rational Agents

Rationality
Depends on performance measure and properties of environment

Perfect Rationality
Select action that maximizes (or minimizes) the expected value of future performance given available resources

Properties of Environments
-Fully vs partially vs not observable: how much does agent know about the environment? -Single-agent vs multi-agent. -Deterministic vs non-deterministic vs stochastic: does agent know what happens after performing an action (full control - deterministic), are probabilities involved? -Episodic vs sequential: are there episodes of independent, similar decisions or a sequence of decisions that influence another? -Static vs Dynamic: static when waiting a short amount of time does not alter the current state, dynamic when it does. -Discrete vs Continuous

Problem-solving methods
Problem-specific, General, Learning

Search
A state can be successor of itself.

State Space
6-tuple $S = (S, A, \text{cost}, T, s_I, S_*)$ where: $s_I \in S$ is an initial state and S_* is a subset of S that represents the set of goal states.

Search nodes
Store a reached state (is part of the state space), how it was reached and at which minimal cost (g-value), and a pointer to the parent node

Open List
Organizes leaves of a search tree. Determines the strategy for which the next node to be expanded is chosen. Must allow to determine and remove the node that is expanded next and also to insert the node that is candidate for future expansion

Closed List
contains states (whereas open list contains nodes), remembers expanded states to avoid duplicate expansion of the same state. Must allow find and insert.

Tree Search
Duplicates possible, tree can have unbounded depth, no guarantee to reach the optimal solution.

Graph Search
recognizes/prunes duplicates. Depth of search tree is bounded

Properties of Search Algorithms
-Completeness: algorithm finds a solution if one exists (semi-complete) and it terminates if no solutions exist (first + second property: complete). -Optimality: if solutions returned by the algo are always optimal. -Time Complexity: worst case analysis, function of branching factor b (highest number of applicable actions (max n. of successors) for a state and search depth d (maximal depth of the search tree) -Space Complexity: Worst case, measured in how many nodes are in memory at same time. Function of b and d.

Breadth-First graph search
Expands nodes in order of generation (FIFO). first node generated is also the one expanded next. Searches layer by layer. Always finds most superficial/simple goal state first. Stops once goal state is generated. If there are multiple paths to a goal state, it finds the shortest one. Always yields a solution with smallest length, but not the solution with lowest cost.

BFS - Properties
Complete, optimal if all actions have the same cost, time and space complexity of $O(b^d)$. Special case of UCS.

Uniform cost search
Identical to Dijkstra: searches for shortest paths in terms of cost to a goal node. Graph search (no duplicates). Does not stop once goal state is generated (a better solution can exist), but it does when goal state gets expanded. Guarantees that every state is expanded on the cheapest path. Always finds cheapest goal state first.

UCS - Properties
Complete, optimal, time and space complexity depend on distribution of action costs

Depth First tree search expands node in opposite order of generation (LIFO). Deepest node expanded first, stops when goal node is generated (like BFS).

DFS - Properties
complete for acyclic state spaces, neither complete nor semi-complete otherwise (can get stuck in a cycle). Not optimal (no matter costs). Time complexity $O(b^m)$, where m is maximal search depth. Space complexity O(bm). No closed list, only open list is kept.

Depth Limited Search
Parameterized with depth limit l, LIFO tree search. Prunes search nodes at depth d >= l and stops expanding. Duplicates are not pruned. If l = 2, then node at depth 2 are present but children (l = 3) not.

Iterative Deepening DFS Performs a sequence of depth-limited searches, and increases depth limit l in each iteration. Wasteful: each iteration repeats all useful work done in previous iterations. Can be preferred over depth-first search (does not get stuck in cycles as DFS does)

IDDFS - Properties
Complete for acyclic state spaces, and semi-complete otherwise. Optimal in UCS setting. Time complexity $O(b^d)$, space complexity O(bd) (during each iteration tree search is performed and not graph search)

Uninformed
Means that no information besides formal definition is used to solve a problem

Heuristic function
takes a state as input, and returns a non-negative or infinity (when goal not reachable) heuristic value as output. Heuristics are defined for states and not for search nodes. h(s) estimates distance (cost of cheapest path) from s to the closest goal state. Meaning: goal distance estimator.

Properties of heuristics
the closer h is to true goal distance, the more efficient the search using h. Heuristics are real values, not only natural.

Perfect heuristic
- safe: if $h^*(s) = \infty$ for all $s \in S$ with $h(s) = \infty$
- goal-aware: $h(s) = 0$ for all goal states $s \in S_*$
- admissible: $h(s) \leq h^*(s)$ for all states $s \in S$ (heuristic value never larger than true value, heuristic function cannot indicate that a goal state is further away from any state than it actually is.)
- consistent: $h(s) \leq cost(a) + h(s')$ for all transitions $s \xrightarrow{a} s'$ (heuristic of transition does not drop more than cost of the action, heuristic function never overestimates the cost of reaching the goal node from the current node).

Example: a heuristic with h(1) for a goal state is not admissible. Theorems: 1) admissible => safe + goal-aware 2) goal-aware + consistent => admissible (consistency alone is not sufficient)

Heuristic search
use heuristic functions to partially or fully determine the order of node expansion in the open list. If consistent, we always expand on cheapest path.

Best-first search
Heuristic search algorithm that expands best node in each iteration. Decision on which node is best uses heuristics. Graph search. Neither LIFO nor FIFO, but instead node with smallest heuristic value is expanded next. UCS is a best first search algorithm.

Best-first search Algorithms
- f(n) = g(n) = (n.pathcost) -> UCS: only path cost counts (uninformed) - f(n) = h(n) -> greedy Best-first Search. mark g-value just for clarity. - f(n) = g(n) + h(n) -> A* - f(n) = g(n) + w*h(n) -> weighted A* -> using w switching between greedy best first search(w=∞),A*(w = 1)and uniform cost search (w = 0) Remember to include path cost of previous node too!

Best-first search - Reopening
refers to the process of reexpanding a node that has already been expanded and closed. This can happen when a better path to the node is discovered during the search.

Greedy Best-first Search
No reopening and the order in open list is from smallest heuristic value to largest

Greedy Best-first Search - Properties
complete (like all variants of best-first graph search). Not optimal, no matter the properties of the heuristic. Time and space complexity depend on heuristics.

A*
Trades off path cost and proximity (closeness) to goal. Algo does not stop once goal node is reached. It is false that GBFS never generates more nodes than A*. GBFS should be used to come up with any solution, whereas A* with an optimal one. GBFS and A* do not prune on node generation, but on node expansion. After reaching a goal node, expand later nodes with smaller f values. Both stop once the goal state is selected for expansion.

A* - Properties
Complete. - With reopening: optimal with admissible heuristics. Without reopening: optimal with admissible and consistent heuristics -> nodes are expanding in order of increasing f-values, never expands a node that has already been expanded, and is optimal

Weighted A*
A* parameterized with weight w. Good to be used iff solution may be sub-optimal by a constant factor. If solution must be guaranteed to be optimal better to use A*, and if it is entirely irrelevant then GBFS is best.

Weighted A* - Properties
Weight parameter controls greediness of search: when w = 0, same as UCS, when w = 1 same as A*, when w -> ∞, like greedy best-first search If h admissible, then found solution guaranteed to be at most w times as expensive as optimum when reopening used. If h admissible and consistent same properties, with no reopening needed.

Exercise Questions
A state space always leads to a finite search tree -> False, e.g. when there are cycles.

Constraint Satisfaction Problems
All CSPs can be compiled to a state space search problem. A solution for a CSP is an assignment to all variables that satisfies all constraints. Domain: set of values that a variable can be assigned to
3-tuple $C = \langle V, \text{dom}, (C_{uv}) \rangle$.

Performance measure
All solutions are equally good (when multiple optimal ones exist)

Binary Constraints
Express which joint assignments to u and v are allowed in solutions, it is trivial if there is no restriction on the joint assignment (usually not displayed)

Partial Assignment
Values are assigned to some or to all variables

Total Assignment
defined on all variables (all vars assigned)

Consistency
Variables do not violate the constraint that defines them.

Solution
Let C be a CSP. A consistent and total assignment of C is a solution of C. Consistent partial assignments α cannot necessarily be extended to a solution, it means that so far no constraint is violated

Backtracking
Discard inconsistent partial assignments. Backtracking is a form of depth-first search, which is well suited for CSPs because the path length is bounded by the number of variables, solutions are located at the same depth (lowest search layer), state space is a directed tree, where the initial state is the root (no duplicates). Don't already generate all values on a level (like in normal DFS) only generate one at a time and expand down if it is consistent. Naive Backtracking: no inference.

Inference
Derive equivalent, but tighter constraints (implied by the given constraints) to reduce the size of the search space. More inference leads to a smaller search space

Variable Orders
Select-unassigned-variable method in backtracking search allows to influence order in which variables are considered for assignment. Goal: make critical decisions as early as possible

Minimum Remaining Values
prefer variables that have small domains, intuition: few sub-trees lead to a smaller tree, so solution is found more quickly

Most Constraining Variable
prefer variables contained in many nontrivial constraints, intuition: constraints are tested early, inconsistencies are so tested early: smaller tree

Value Orders
Order-domain-values method (pick value so that it's likely to find a value for all other unassigned variables) in backtracking search allows to influence order in which values of the selected variable are considered. Goal: make most promising assignments first

Conflict
happens when a pair of values for an assignment not part of the associated constraint

Forward checking
For all unassigned variables in a partial assignment, remove all values from the domain that are in conflict with already assigned variable/value pairs in the partial assignment. CSPs before and after FC have the same solution. With FC, binary constraints are not affected, but domains are (unary constraints).

Arc consistency
Node consistency: single variable
Arc consistency: two variables (do not need to be symmetric) For every assignment to a variable u, there must be a suitable assignment for every other variable v. If not, remove values of u for which no suitable partner assignment to v exists -> tighter unary constraint on u. Stronger than forward checking. Trivial constraint: satisfied. No symmetry implied.
AC-3: for each binary constraint, put <u, v> and <v,u> in queue, update domains, put next ones in queue and then provide final domains. CSP arc consistent if every variable is arc consistent with respect to every other variable (for every value of u there is value of v that is consistent, and viceversa), not symmetrical definition, v always arc consistent with respect to v' if constraint between the two is trivial

Path consistency
arc consistency extended to three variables: for every joint assignment to variables u, v there must be a suitable assignment to every third variable w. If not, remove pairs of values u and v for which no suitable partner assignment to w exists -> tighter binary constraint on u and v. Arc consistency and path consistency do not imply each other. Not stronger than arc consistency, as it does not change the domains of the variables. Useful if there are binary constraints among 3 or more variables.

Exercise Sheet
If depth of the optimal solution is known, use depth-limited search rather than IDDFS

Games

Game
7-tuple $S = \langle S, A, T, s_I, S^*, \text{reward}, \text{player} \rangle$, where e.g., $S = \{sI\} \cup \{sx | x \in \{r, p, s\}\} \cup \{sxy | x, y \in \{r, p, s\}\}$ reward : $S^* \rightarrow$ assigns to every goal state a real value number. and player function player : $S \setminus S^* \rightarrow \{\max, \min\}$. It is described for every non-terminal state. Initial state: incoming arrow. Goal/Terminal States: Double circles

Games - Properties
Finite set of states and finite set of actions players can play. Effect of an action can be predicted. Terminal state reached after a finite number of steps. Game ends when a terminal state is reached. Terminal states yield a reward, two players, called max and min. Both players observe the entire state (perfect information). Reward for max is opposite of reward for min -> zero-sum game.

Policy
Strategy that determines an agent's action in various states of the game. It maps game states to actions the agent should take.

Partial Policy
incomplete strategy that specifies actions for some. but not all, game states. Does not cover ensure state space of the game.

Total Policy
complete strategy that specifies actions for every possible game state. Maps every game state to a corresponding action.

Minimax search
DFS in game tree, determine state value for terminal state with reward function. Compute state value of inner nodes from below to above through tree: for min: state value is minimum of state values of children. For max: state value is maximum of state value of children. Minimax yields optimal policy (for acyclic finite trees) when the opponent plays perfectly. Max obtains at least the state value computed for the root, no matter how min plays. During each tree height, max and min get alternated, root is max (triangle up), min is triangle down

Evaluation function
Used to approximate the state value of a given state, ∞ not allowed to be mapped. Similar to heuristic function. Idea is to search only up to a predefined path. High values should relate to high winning changes (close to true state values)

Alpha-Beta pruning
Optimization of minimax where we use two values α and β during minimax search such that α is lower bound on reward of max and β is upper bound on reward of min. For every recursive call it holds that a subtree is not interesting if: 1) its state value is <= α because max will never enter it when playing optimally. 2) if its state value is >= β because min will never enter it when playing optimally. 3) does not have to be searched further (prunes) if $\alpha >= \beta$. Prunes parts of tree that are never reached under perfect play. It does not maintain upper and lower bound on relevant search depth. In root start with α = - ∞ and $\beta = +\infty$. When going downwards: Pass α and β the way they are, when upwards always swap values.
Perfect move ordering: best move in every state is considered first. For every game, minimax and alpha-beta search play the same action in the initial state Max node: update alpha, min node: update beta.

Monte-Carlo tree search
involves sampling of game simulations and estimation of state quality by averaging over simulation results. Environment is same as games. Tree search variant (no closed list), starting from root node it simulates game by performing decision in the tree in single trajectories (by extending step by step), no explicit open list. Search nodes are the only used data structure. During each trial add at most one search node to the tree (if trial ends in a terminal node nothing gets added). MCTS builds an asymmetric game tree
Each iteration consists of four phases: 1) Selection: apply tree policy through tree until a terminal node or a node with a missing successor is reached 2) Expansion: for one of the missing successors, add a search node to the game tree 3) Simulation: apply a default policy from the added successor until a terminal state is reached (no search nodes are added to the tree). Simulate the game until the end (terminal state gets reached), without remembering any information. 4) Backpropagation: update the visited search node in reverse order: increase visit counter N by 1 and update state value estimate with the reward from the reached terminal state by computing the average. Perform next iteration if resources allow and play action with highest state value estimate otherwise. If all successors have already been added and the visited state is not terminal apply search policy which determines how search is performed. To guarantee optimality of MCTS in the limit use tree policy that is greedy in the limit and explores forever (default policy is irrelevant, even if it is optimal).

Simulation Phase - Default Policy
Tree does not get changed. Used as heuristic (to come up with a value for a newly added search node). Only operates on states, never encounters search nodes during its run. Balances exploration and exploitation. Cheaper than tree policy as it does not influence where the search is focused on.

Monte Carlo Random Walk
In each state, select an action randomly until a terminal state is reached: cheap but uninformed.

Simulation Phase - Tree Policy
Exploit collected info to focus search on most promising areas: prefer successors with high (max) or low state value estimate (min). Explore areas that have not been investigated completely. Applied as long as we are in the known part of the tree (all successors are generated).

Search node
Stores a reached state, how it was reached, its successors (children), a state value estimate $\hat{v}$ and a visit counter N. The list of successors is sorted.

Asimptotic Optimality
If it explores forever (every state is added to the game tree and visited infinitely often), and is greedy in the limit: prob that an optimal action is selected converges to 1.

ϵ-greedy
Tree policy with constant parameter ϵ. With constant probability 1 - ϵ, pick a greedy action which leads to a successor with highest state value estimate (for max) and a successor with lowest state value estimate (for min). Otherwise, pick a non-greedy successor uniformly at random. Explores with probability ϵ and exploits otherwise. Explores forever but not greedy in the limit (prob to take greedy action does not increase over time). Decrease of ϵ leads the agent to exploit its current knowledge more often. Problem: when ϵ-greedy explores, all non-greedy actions are treated equally -> solution: use softmax.

Softmax
Tree policy with constant parameter $\tau > 0$. Select actions with a probability/frequency that directly relates to their state value estimate. Explores forever, but not greedy in the limit. Boltzmann exploration selects actions proportionally to:

For max: $P(n) \propto e^{\frac{n\hat{v}}{\tau}}$

For min: $P(n) \propto e^{-\frac{n\hat{v}}{\tau}}$

Upper Confidence Bounds 1
Balance exploration and exploitation -> preferring actions successful in earlier iterations (exploit) and have been selected rarely (explore). Asymptotically optimal. UCB1 selects action with highest upper confidence bound (takes into account how often an action is tried).

Expectiminimax
Depth-first search in game tree. Determines state value of terminal state with reward function. Compute state value of inner nodes from below to above through the tree: min and max like minimax, Chance: state value is probability-weighted sum of state values of children. Policy is action that maximizes state value. Yields optimal policy for finite trees (under assumption that opponent plays perfectly)

Markov Decision Processes

Stochastic game
7-tuple $S = \langle S, A, T, s_I, S^*, \text{reward}, \text{player} \rangle$
The set of applicable actions in s is $A(s)$, and $A(s) \neq \emptyset$ for all $s \in S$.

Expectimax
Depth-first search in search tree. State value at horizon is 0. Compute state values of inner nodes from below to above through the tree: in a max node, let the successor value of a child be the sum of reward and the child's state value. chance node: state value is probability-weighted sum of state values of children. Yields an optimal policy, and the agent obtains exactly the state value computed for the root in expectation. For calculating state value of root, choose child that yields highest reward and always include rewards from both children for calculation. Only max player.

Markov
Action outcomes only depend on current state and applied action.

Finite-horizon MDPs
6-tuple $M = \langle S, A, T, \text{reward}, s_I, h \rangle$ where horizon $h \in$ Terminate after fixed number of steps: agent-environment interaction is finite and solutions are acyclic.

Infinite-horizon MDPs
6-tuple $M = \langle S, A, T, R, s_I, \gamma \rangle$ where discount factor $\gamma \in (0, 1)$. S is set of states, A is set of actions, T(s, a, s') is transition function, R(s, a) is reward function.
Do not terminate, infinite interactions and solutions must be cyclic (otherwise couldn't have infinite interactions without revisiting a state). In infinite iteration, expected rewards may grow forever and policy yields infinite expected reward. Solution: prefer rewards now to rewards later: obtained rewards decay exponentially with discount factor γ, which ensures that the state values converge despite cycles and infinite horizon (infinite sequence of rewards). The smaller the discount factor, the more important are earlier (short term) rewards.

Bellman Equation
Let $M = \langle S, A, T, \text{reward}, s_I, \gamma \rangle$ be an MDP.
The Bellman equation for a state s of M is the set of equations
$v^*(s) := \max_{a \in A(s)} q^*(s, a)$,
where $v^*(s)$ is the state value of s and
$q^*(s, a) := \text{reward}(s, a) + \Sigma_{s' \in S} T(s, a, s') \cdot v^*(s')$
is the action- or Q-value of a in s. The solution $v^*(s)$ of the Bellman equation describes the maximal expected reward that can be achieved from state s in MDP M. The policy that achieves this reward is the optimal policy.

Optimal Policy
Let $M = \langle S, A, T, \text{reward}, s_I, \gamma \rangle$ be an MDP.
A policy π is an optimal policy if $\pi(s) \in \arg\max_{a \in A(s)} q^*(s, a)$ for all $s \in S$ and the expected reward of π in M is $v^*(s_I)$.

Value Functions
Let π be a policy for MDP $M = \langle S, A, T, \text{reward}, s_I, \gamma \rangle$.
The state-value $v^\pi(s)$ of $s \in S$ under π is defined as
$v^\pi(s) := q^\pi(s, \pi(s))$,
and the action- or Q-value $q^\pi(s, a)$ of s and a under π is defined as
$q^\pi(s, a) := \text{reward}(s, a) + \Sigma_{s' \in S} T(s, a, s') \cdot \gamma \cdot v^\pi(s')$.
The state value $v^\pi(s)$ describes the expected reward of applying π in MDP M, starting from s. It's the same as the Bellman equation if the policy is optimal.

Policy Iteration
- Policy evaluation: compute $V_i^\pi(s) = \text{reward}(s, \pi(s)) +$

$\Sigma_{s' \in S} T(s, \pi(s), s') \cdot \gamma \cdot V_{i-1}^{\pi}(s')$
Assess current policy. - Policy improvement: use v^{π} to determine a better policy. Convergence: policy iteration is optimal if threshold ϵ is small enough. Pick greedy action so that policy improves. In policy iteration, we calculate the state values for all possible actions in each state in order to compute the greedy policy for each iteration. For greedy policy: if only state values are available, you need to perform a "one step lookahead" for each action that gives you the action value of the action from the state values, and the greedy policy is then the policy that always picks one of the actions with maximal action value in each state.

Value Iteration

1. Starts with arbitrary $\hat{v}^0(s)$ for all $s \in S$.
2. Bases estimate $\hat{v}^{i+1}$ on values of estimate $\hat{v}^i$ by treating the Bellman equation as an update rule on all states: $\hat{v}^{i+1}(s) := \max_{a \in A(s)} \text{reward}(s,a) + \Sigma_{s' \in S} T(s,a,s') \cdot \gamma \cdot \hat{v}^i(s')$
3. Converges to the state-values of the optimal policy.
4. The residual of step $i+1$ is

$$\epsilon^{i+1} := \max_{s \in S} \hat{v}^{i+1}(s) - \hat{v}^i(s)$$

5. Terminates when the maximal change is smaller than the threshold ϵ. Convergence: optimal if threshold ϵ is small enough.

Greedy Action

The set of greedy actions in s with respect to v is $A_v(s) := \arg\max_{a \in A(s)} \text{reward}(s,a) + \Sigma_{s' \in S} T(s,a,s') \cdot \gamma \cdot v(s')$
A policy π_v with $\pi_v(s) \in A_v(s)$ is a greedy policy w.r.t. v.
A greedy policy with respect to v^* is an optimal policy.

Value Iteration vs. Policy Iteration:

- *Value Iteration:* searches directly over state-values and the optimal policy is induced by the final state-values. Does both policy iteration steps in one. Only does a single iteration of policy evaluation at each step.
- *Policy Iteration:* searches over policies π by evaluating the state-values under π, until π is an optimal policy. Computing the state values of all states for a given policy π (i.e., computing v^{π}) is called *policy evaluation*. Resulting policy π maps each state to the action (out of the possible ones for that state) which yields the highest value for $v(s)$. If only one action possible then map to that one (trivial).

In policy iteration, you always have a fixed policy, i.e., you always take the action that your policy dictates. In value iteration, you simply take the maximum of all actions because you don't have a policy evaluation step, but instead do everything in a single step. They converge to the same result, given enough time and iterations.

Exercise Sheet

An optimal agent that acts in an uncertain environment is not guaranteed to get the expected reward of the optimal policy nor it is guaranteed to get a higher reward than a suboptimal agent, but in the long run it gets the expected reward of the optimal policy on average.

Reinforcement Learning

Agents has no model, but intelligent behavior possible by learning from experience. More problems can be solved, also those that do not have a model

RL Agent

Has no model or only partial model, improves policy indirectly by planning on learned model (model-based RL) and directly by learning state or action values or policy (model-free Rl). Assuming sufficient resources: an rl agent can always act optimaly, it must act suboptimally sometimes to explore its environment. A planning agent can use its knowledge of the model to always act optimally.

Model-based passive RL

Aim to evaluate a given policy, but transition function T and rewards are not known: no way to compute how good actions are, but rather try out actions and learn Phase 1: Learn an approximate model based on experience 1. Interact with the environment by executing a given policy π.
2. Count outcomes s' for each $\langle s, \pi(s) \rangle$ (denote counts with $N(s,a,s')$).
3. Approximate $T(s,a,s') = \frac{N(s,a,s')}{\Sigma_{s'' \in S} N(s,a,s'')}$.
4. Determine all reward(s,a,s') (no need to approximate since these are deterministic!).

Phase 2: Solve the learned model as if it were exact For example, use iterative policy evaluation. Such an agent evaluates a policy by solving a learned model (e.g., with policy evaluation), and does not use a given model to solve an MDP nor it requires an appropriate learning rate for convergence of state-value, nor it can be used to compute an optimal policy in an MDP.

Monte-Carlo Policy Evaluation

1. Simulate from $s_0 := s_I$ until the terminal state s_n is reached.
2. Observe visited states $s_1, \ldots, s_n$ and obtained rewards $r_0, \ldots, r_{n-1}$.
3. Increment visit counter $N(s_i)$ for all s_i.
4. Update state-value estimates $\hat{v}_{\pi}(s_i)$ of all s_i to the average of obtained rewards: $\hat{v}_{\pi}(s_i) \leftarrow \hat{v}_{\pi}(s_i) + \frac{\Sigma_{j=i}^{n-1} \gamma^{j-i} \cdot r_j - \hat{v}_{\pi}(s_i)}{N(s_i)}$
5. repeat MC policy evaluation converges to true state-values under π due to the law of large numbers

Passive Rl with Temporal-Difference Learning

1. Execute action $\pi(s)$ in s.
2. Observe successor s' and reward r.
3. Update $\hat{v}_{\pi}(s)$ such that the difference between estimates for s and s' better matches the expectation: $\hat{v}_{\pi}(s) \leftarrow \hat{v}_{\pi}(s) + \alpha \cdot (r + \gamma \hat{v}_{\pi}(s') - \hat{v}_{\pi}(s))$ (requires $\hat{v}_{\pi}(s) = 0$ for terminal states s).
4. Repeat. Updates are immediate, no terminal states required, but often slow convergence. Computes correct state-values for evaluated policy and does not need to reach an optimal state to perform an update (updates after each step). TD-Learning can be considered to be the passive analogue to SARSA if we were to fix the policy for SARSA.

Passive TD and MC policy evaluation

Both converge to $v^{\pi}(s_I)$. MC policy evaluation does not use state-values of successor to compute estimates (instead uses observed future reward in current iteration) and passive TD learning does not perform an update only after reaching a terminal state (true for MC policy evaluation), but instead it performs an update whenever it reaches a state. We could also use Monte-Carlo policy evaluation to obtain utility estimates $\hat{v}_{\pi}(s)$ if there are terminal states in the MDP.

Active Reinforcement Learning

no given policy, but active search for one. No model so we need to explore to find food actions. In large state spaces focus on promising areas of state space (exploit). Exploration function in active Rl can be modeled as MDP with a single non-terminal state. Every action has a fixed stochastic reward.

Model-based Active Reinforcement Learning

Follows the same idea as model-based passive RL (phase 1 and 2):
Converges to the true model in the limit if the exploration function explores forever.
If the learned model is sufficiently close to the true model, model-based active RL learns the optimal policy. Can be used to compute an optimal policy in an MDP.

SARSA

1. Select action a' in s'.
2. Update if s' was reached with a from s yielding r: $\hat{q}(s,a) \leftarrow \hat{q}(s,a) + \alpha \cdot (r + \gamma \hat{q}(s',a') - \hat{q}(s,a))$.
(Treats $\hat{q}(s,\cdot) = 0$ for terminal states s)
3. Execute a' and repeat. on-policy: it performs updates based on subsequent decisions. True probabilities not known by the agent.

Converges to true state-values if the exploration function is greedy in the limit and explores forever. SARSA with UCB1 exploration learns an optimal policy in the limit.

Q-Learning

1. Select action a in s.
2. Execute a and observe r and s'.
3. Update $\hat{q}(s,a)$: $\hat{q}(s,a) \leftarrow \hat{q}(s,a) + \alpha \cdot (r + \gamma \max_{a'} \hat{q}(s',a') - \hat{q}(s,a))$.
(Treats $\hat{q}(s,\cdot) = 0$ for terminal states s)
4. Repeat. Off-policy: it performs update on best possible subsequent action.

Converges to the true optimal state-action values (Q-values) if the exploration function explores forever, i.e., prob for exploration needs to be larger than 0 at all times. Q-Learning keeps track of the action-values via a table which become infeasible to store all at once for large state spaces, so it can be approximated in environments where the state or action spaces are too large to be handled.

Q-Learning vs SARSA

In Q-learning, we update the Q-values by selecting the maximum Q-value for the next state-action pair, meaning we choose the best action that could have been taken. In SARSA, we update the Q-values using the action that was actually taken in the next state, following the current policy (off-policy vs on-policy). Q-Learning and SARSA are model-free methods, meaning they do not learn the underlying MDP but try to estimate state-action values directly. However, if we know the MDP, then we should perform planning, e.g. via value iteration or policy iteration. Q-Learning can lead the agent to favor shorter, more direct paths towards the goal, even if they are riskier. SARSA is more conservative in its updates and learns a safer policy. This often results in a longer path that avoids the risk of obtaining a negative reward. Both: Model-free (no need to learn transition probs). The smaller α, the closer it converges to optimal Q-values, but the slower it converges.

Approximate Reinforcement Learning

idea: learn on small part of MDP and then generalize the experience to new and similar situations. Does not need to visit all states during learning, nor to store one estimate for each state or state-action pair.

Planning

Model available, no longer learning.

Properties of Environment

Same as CSPs
Objective of the agent: find a plan (sequence of actions) that reaches a goal state from an initial state.

State Spaces with Declarative Representations

Input is a given state space description and there is no problem-specific heuristics. State spaces larger than input, world is described in terms of state values.

Planning Formalism

Is a description language for planning tasks.
Three types: PDDL, STRIPS, SAS^+. All of these can be compiled from one to another. The STRIPS heuristic is safe and goal-aware, but not admissible.

STRIPS

special case of PDDL where preconditions and goals are restricted to conjunctions over positive literals. No parameters, no objects (lifted variables), only state variables. All state variables in V are binary (true or false) and set of states s is represented as s $\subseteq$ V , where s encodes the set of state variables that are true in s goals and preconditions of actions are given as sets of variables that must be true. effects of actions are given as sets of variables that are set to true and set to false, respectively.
4-tuple $\Pi = \langle V, I, G, A \rangle$, where A is a finite set of actions $a = \langle pre, add, del, cost \rangle$ with preconditions $pre(a) \subseteq V$, add effects (or add list) $add(a) \subseteq V$, delete effects (or delete list) $del(a) \subseteq V$, and costs $cost(a) \in 0$ ($cost(a) = 1$ if not specified explicitly).

SAS^+

Similar to STRIPS, state variables not necessarily binary, but with given finite domain. States are assignments to these variables, preconditions and goals are now partial assignments (but still in conjuction), effects are assignments to subset of variables.
5-tuple $\Pi = \langle V, dom, I, G, A \rangle$, where A is a finite set of actions $a = \langle pre, eff, cost \rangle$ with preconditions $pre(a)$, a partial assignment of V to dom, effects $eff(a)$, a partial assignment of V, and cost $cost(a) \in 0$.

Obtaining a Heuristic

Procedure: solve a simplified version of the problem, and use solution of the simplified problem as heuristic for the original problem.

Abstraction

Idea: estimate solution costs by considering a smaller planning task where states are merged to abstract states.
An abstraction of a state space S is defined by: an abstraction function α that determines which states can be distinguished in the abstraction Based on S and α, we compute the abstract state space S_{α} which is "similar" to S but smaller. Idea of the abstraction heuristic h_{α}: Use abstract solution costs (solution costs in S_{α}) as heuristic values for concrete solution costs (solution costs in S). More informed abstraction means higher heuristic value.

Induced Abstraction

The abstraction of S induced by α, denoted as S_{α}, is the state space $S_{\alpha} = \langle S', A, cost, T', s_I', S^{*\prime} \rangle$ with:

- $T' = \{\langle \alpha(s), a, \alpha(t) \rangle \mid \langle s, a, t \rangle \in T\}$
- $s_I' = \alpha(s_I)$
- $S^{*\prime} = \{\alpha(s) \mid s \in S^*\}$

A state in an abstraction is a goal state if the abstraction in which it is contains a goal state. Every plan in the original system is also a plan for the abstraction. In the abstraction we can have much cheaper plans as in the original system. The heuristic value for a concrete state corresponds to the shortest path from the abstract state it is part of to the goal state.
Every abstraction heuristic is admissible and consistent. **Pattern Databases**
Abstraction heuristic used to come up with a good abstraction function α. Has some state variables of the task preserved with perfect precision while others are not preserved at all. Formalized as projections on a pattern P (subset of the variables).

Pattern Database Heuristic

Let P be a subset of the variables of a planing task.
The abstraction heuristic induced by the projection π_P on P is called *pattern database heuristic* (PDB heuristic) with pattern P. Abbreviated notation: h_P for h_{π_P}.

How to compute a pattern database

Compute the syntactic projection of Π to P as $\Pi|_P = \langle P, dom|_P, I|_P, G|_P, \{a|_P \mid a \in A\} \rangle$, where $a|_P = \langle pre(a)|_P, eff(a)|_P, cost(a) \rangle$ (i.e., all variables not in P are ignored everywhere).
Compute the abstraction $S(\Pi|_P)$ from $\Pi|_P$.
Compute and store the pattern database, which maps each abstract state to its goal distance in $S(\Pi|_P)$.
The transition systems S_{π_P} and $S(\Pi|_P)$ are equivalent. $|_P$ means restricted to the variables in the pattern P. Store values of abstract states and not concrete states (way more), and only store distances for abstract states, since concrete states in the same abstract state store the same heuristic value. Compute heuristic for each concrete state, which is equal to cost from abstract state to a terminal state in the abstraction.

Delete Relaxation

Estimate solution costs by considering a simplified planning task where all negative action effects are ignored.

Relaxed Planning Tasks

Relaxation of actions

The relaxation a^+ of STRIPS action a is the action with $pre(a^+) = pre(a)$, $add(a^+) = add(a)$, $cost(a^+) = cost(a)$, and $del(a^+) = \emptyset$.

Relaxation of planning tasks

$\Pi^+ := \langle V, I, G, \{a^+ \mid a \in A\} \rangle$.

Relaxation of action sequences

$\pi^+ := \langle a_1^+, \ldots, a_n^+ \rangle$.
STRIPS planning tasks without delete effects are called relaxed planning tasks. If Π is a STRIPS planning task and π^+ is a plan for Π^+, then π^+ is called a relaxed plan for Π. A plan for the relaxed task is a relaxed plan for the original planning task. $h^+(\Pi)$ denotes the cost of an optimal plan for Π^+, i.e., of an optimal relaxed plan.

Analogously: $h^{+(s)}$ is the cost of the optimal relaxed plan starting in state s (instead of the initial state). h^+ is called the optimal relaxation heuristic. For any action sequence applicable in a task, its relaxation can be applied in the relaxed task. If any action sequence is a plan in the original task, then that sequence is also a plan for the relaxed task, having the same cost. If a pattern P contains no variables, then all states in $\Pi|_P$ (the abstractions) are goal states. Optimal plan of the abstraction can be shorter than the one for the original pan but never more expensive. For general STRIPS planning tasks h^+ is an admissible and consistent heuristic. We always transition to a state with more true values (never lose anything, only gain), as only the positive are kept (negative ignored). If a planning task Π is solvable, then its delete relaxation Π^+ is solvable, opposite does not always hold.

Relaxed planning graphs

Represent which variables in Π^+ can be reached and how.
Graphs with variable layers V_i and action layers A_i alternated. Directed edges: From v_i to a_{i+1} if $v \in pre(a)$ (precondition edges). From a_i to v_i if $v \in add(a)$ (effect edges). From v_i to G_i if $v \in G$ (goal edges). From v_i to v_{i+1} (no-op edges).

Maximum and additive heuristics

h^{max} and h^{add} are the simplest heuristics. Annotates vertices with numerical values. Costs of variable vertices: 0 in layer 0; otherwise, the minimum of the costs of predecessor vertices. Costs of action and goal vertices: maximum (h^{max}) or sum (h^{add}) of predecessor vertex costs; for action vertices a_i, also add $cost(a)$. In the h^{max} heuristic, (for variables), you consider the sequence of actions with the smallest total cost needed to achieve each individual goal state from the initial state, and then you take the maximum of those costs. In the h^{add} heuristic, (for variables) you consider the sequence of actions with the smallest total cost needed to achieve each individual goal state from the initial state, and then you sum up all of those costs. Also add cost of actions that are shared to reach the individual goal states.

Heuristic value Value of the goal vertex in the last layer.

h^{max} vs. h^{add}

both are safe and goal-aware. h^{max} is admissible and consistent; h^{add} is neither (not suited for optimal planing), more informative than h^{max} h^{add} often overestimates the actual costs: use FF-heuristic.

FF Heuristic

Identical to h^{add}, but with additional steps at the end: Mark the goal vertex in the last graph layer. Apply the following marking rules until nothing more to do: Marked action or goal vertex? $\Rightarrow$ mark all predecessors. Marked variable vertex v_i in layer $i \geq 1$? $\Rightarrow$ mark one predecessor with minimal h^{add} value (tie-breaking: prefer variable vertices; otherwise arbitrary). Heuristic value: The actions corresponding to the marked action vertices build a relaxed plan. The cost of this plan is the heuristic value (cost of the actions only). Operators marked at the end are then used to come up with the heuristic. h^{FF} is guaranteed to be no higher than h^{add}. Allows for actions being executed in parallel. h^{add} can be higher or lower than h^*

FF Heuristic: Remarks

like h^{add}, h^{FF} is just safe and goal-aware. Always at least as good as h^{add}.
$h_{\max}(s) \leq h_+(s) \leq h^*(s)$
$h_{\max}(s) \leq h_+(s) \leq h_{\text{FF}}(s) \leq h_{\text{add}}(s)$
h^* and h_{FF}, and h^* and h_{add} are incomparable. $h^*(s)$ is the optimal solution, $h_+(s)$ is the optimal solution for the relaxation task.

Probabilistic Reasoning

Full Joint Distribution

Size with n variables and maximal domain size k is $O(k^n)$. It contains all joint distributions. Do not care about some variables? use marginal distribution: obtain it with marginalization, by summing out irrelevant variables $Y_1, \ldots, Y_m$: $P(X_1, \ldots, X_n) = \Sigma_{y_1} \cdots \Sigma_{y_m} P(X_1, \ldots, X_n, y_1, \ldots, y_m)$
Sufficient for all unconditional queries. **Conditional Probability**
$P(x)$ is called prior probability of x (assumes no additional information). If we learn that variable Y has value y, our belief on x changes. $P(X|Y) = \frac{P(X,Y)}{P(Y)}$
Normalization trick: select entries matching evidence, normalize selected entries (divide each by sum). $P(X|Y) = \frac{P(Y|X) \cdot P(X)}{P(Y)}$

Independence

iff P(x, y) = P(x) · P(y) $\forall x \in \text{dom}(X), \forall y \in \text{dom}(Y)$,denoted with$X \perp\!\!\!\perp Y$.
$X$and$Y$are conditionally independent given$Z \iff \forall x, y, z : P(X = x, Y = y \mid Z = z) = P(X = x \mid Z = z) \cdot P(Y = y \mid Z = z). X \perp\!\!\!\perp Y \mid Z$

Bayesian Networks

Graphical model which describes complex joint distributions using local conditional distributions. Consists of a set of nodes, a set of edges between the nodes, and a conditional probability table for each node. Every node has an associated variable and domain. Nodes and edges form directed acyclic graph. An edge from X to Y indicates that Y causes or influences Y. Not every full joint distribution can be described by a bayesian network, but the basic bayesian network with just the chain rule is able to represent any full joint distribution. Size of bayesian is only $O(n \cdot 2^{k+1})$ for n boolean variables with at most k parents, and has same power as full joint distribution. Bayes' net with $k < n$ cannot represent every joint distribution. Size of bayes net is determined by the largest table. k is largest number of parents. For total prob second factor (right of parentheses) must change. **Chain Rule**
$P(A) * P(B) * P(C \mid A, B) * P(D \mid A, B, C) \ldots$ holds for any Bayes' net.

Conditional Probability Table

Every node is conditional over its parent node, not over its successors. Every node has an associated conditional distribution. $P(C \mid A, B)$ means that C has parents A and B.
Probability of full assignment computed as $P(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid \text{parents}(X_i))$

D-Separation

Used to determine if variables X and Y are conditionally independent given variables $Z_1, \ldots, Z_n$. It checks **all undirected** paths between X and Y and analyzes triples of variables (nodes) on the path. A path is blocked if at least one triple is blocked: no information flow. If all undirected paths from X to Y are blocked, the X and Y are conditionally independent given $Z_1, \ldots, Z_n$. One unblocked path -> not independent. Cases:
Causal chain $A \rightarrow B \rightarrow C$ (or $A \leftarrow B \leftarrow C$) where B$\notin$ Z. Unblocked, with B as evidence is blocked (independent).
Common cause: A $\leftarrow B \rightarrow C$where B$\notin$ Z. B node as evidence? if yes then blocked
Common effect: $A \rightarrow B \leftarrow C$where B or one of its descendants is in Z. B node as evidence? unblocked, otherwise blocked.

Inference using Bayesian Network

With Full Joint Distribution: select entries consistent with evidence, sum out hidden variables, normalize. Improvement: instantiate all factors (i.e., all CPTs), and while there are still hidden variables pick a hidden variable V, join all factors mentioning V and then eliminate V. After this step join remaining factors and finally normalize. It marginalizes (eliminates) as soon as possible -> variable elimination. Join step: multiply probabilities, e.g: $P(S \mid F) \cdot P(A \mid S) = P(A, S \mid F)$. Vars that are in common are on left of parentheses, vars that were in left but not in common stay left, all others are evidence -> in join.

Supervised Learning

Assume data and labels are given. In classification there are a finite number of values a variable can take (discrete output value). Data is composed of features and each feature has its domain.

Decision Trees

Trees that consist of non-leaf vertices v with associated feature, leaf nodes with associated class and edges which are labeled with a Boolean value test (e.g. <= 10 or = T). Tests must be mutually exclusive and collectively exhaustive, meaning that every possible value must pass exactly one of the tests in the decision tree. Pick feature that maximizes information gain. Every feature occurs at most once on each root-leaf path in the tree (can appear multiple times in entire tree). Continue until all nodes have associated feature or are leaves. Testing same attribute more than once along a path in a dt is not beneficial (provides no new information that could change the decision). Information gain and structure (form) are learned by decision tree decision tree is a model space. Every data point in training set is classified perfectly.

Entropy

$H(X) := -(\Sigma_{x \in \text{dom}(X)} P(x) \cdot \log_2 P(x))$, where X is a random variable.
Measures the degree of uncertainty of a random variable: ↑ entropy = ↑ remaining difficulty. Information gain is the difference of current and remaining difficulty, to obtain it for a feature subtract probability-weighted entropies from previous difficulty. When entropy close to 1, certainty of equal separation (high uncertainty), when close to 0 all in one class and none in other (low uncertainty). Entropy can be negative, but information gain cannot.