|  |
| --- |
| **`Optimisation:** Greedy Algo: Makes best choice every step, only works on matroids, (shortest path, MST, activity selection). P, NP: A Problem is P if there is a deterministic algo that solves in polynomial time (i.e. (O(nk)). Called feasible or tractable. A problem is NP if there is a non-deterministic algo that solves in poly time. Called infeasible or intractable. NP Complete problems are a class of problems which are polynomially equivalent to each other. NP Hard are at least as hard as NP complete. NP Hard has: No known poly-time algo, only algo’s take exponential time, could solve all in poly time if you can solve 1 in poly time. K-OPT: Taking groups of k vertices and exchanging them if it improves the outcome, when no more optimizations exist it is k-optimal. Gradient-Based Search: A State space graph is a huge graph with each vertex representing a feasible tour for I, 2 vertices are connected if they can be obtained by moving across an edge (legal move). Finding vertex with lowest cost is goal. S(I) is vast, GBS is based on deciding whether to move from current T to T’. Hill Climb: Generate all neighbours of T, move to the first one with lower cost. Choosing best move is greedy iterative improvement. Local Optima: Hill climb will finish at local optimum of state-space. Only 1 local-optima is the global. 3 methods of escaping local optima: Simulated Annealing: tries to avoid local by taking “backward” moves. If T’ ≤ T, accept, if T’ > T, accept with probability p . Reduce p as time goes on. Highly problem specific. Tabu Search: Maintain tabu list containing last h vertices visited. Then: select best neighbour of T, if T’ not on list, accept, update list. Very aggressive. Problem specific, expensive. Genetic Algorithms: Maintain population of solutions. Expected to be distributed across search space. At each gen, population n is used to create n new solutions, best n of those survive to the next gen. Extremely successful with fine tuning. |
| **Agents:** An Agent: Perceives its environment through sensors and acts on its environment through effectors. Rational Agents: Try to do the “right thing” wrt a set of goals. Can be specified by performance measure. Rational action maximises expected value. Simple Reflex Agents: Choose action using condition-action rules “if-then”. No history stored. Model-based Reflex Agents: Stores memory, understanding of the effects of actions. Person signal to bus, know that bus will stop, therefore change lanes. Goal-based Agents: Understanding the effects of actions in relation to the goal. Planning is fundamental. Utility-based Agents: Utility defined as something to be maximised. Agent tries to maximise expected utility. |
| **Uninformed Search Algorithms:** Finding a sequence of actions that changes the world from its current state to a desired goal state is a search problem. Comparing Searches: Completeness: is it guaranteed to find a solution? Optimality: guaranteed to find the optimal solution? Time Complexity: How long does it take? Space Complexity: How much memory is needed? Terms: *b = max branching factor, m = max depth of search space, d = depth of least-cost* solution. Breadth-first Search: Expand shallowest node next, complete level before moving down. Space is the issue. Depth-first Search: Expand deepest node next, follow path to terminus, then backtrack to last choice and try alterative, Space is advantage, other metrics are big disadvantages. Depth-limited Search: Depth-first with a cut-off depth, terminate paths at depth L. Apply depth-first to infinite spaces, good if L is good. Iterative Deepening DF Search: Repeated depth-limited, increasing cut-off, check deeper and deeper, iteratively increasing L. for typical b, last tree layer dominates space, worth it for space complexity. ID allows system to adapt to resource limits, “anytime algorithm”. Bi-Directional Search: Search from both ends concurrently, usually expands fewer nodes than unidirectional (2b(d/2) << bd). However, may be many goal states to start from, formalising backward steps Is hard, backward branching factor may be bigger than b, cost of checking when converging may be high.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Algorithm** | **Complete** | **Optimal** | **Time** | **Space** | | **Breadth-first** | Yes if b is finite | Yes, for constant step-costs | O(bd) | O(bd) | | **Depth-first** | No, fails in infinite-depth spaces | No, hits any solution first | O(bm) | O(bm) | | **Depth-Limited** | No | No | O(bl) | O(bl) | | **Iterative Deepening DF** | Yes | Yes, for constant step-costs | O(bd) | O(bd) | | **Bi-Directional** | Yes if b is finite and if both directions use bfs | Yes, if all step costs are identical and if both directions use bfs | O(bd/2) | O(bd/2) |   Summary: Iterative Deepening offers: completeness and optimality of breadth-first, and the space advantage of depth-first. |
| **Informed Search Algorithms:** Selects nodes on basis of estimate of distance to goal state, requires either heuristic rules or evaluation function. Greedy Search: Always selects the unvisited node with the smallest estimated distance to goal. Heuristic h(n) is the estimate of the cost of getting from n to the goal. A\* Search: use estimate of total path-cost as heuristic: f(n) = g(n) + h(n). g(n) = actual cost from start to n, h(n) = estimated cost from n to goal, f(n) = estimated total cost from start to goal via n. A\* Optimality: optimal under 2 conditions: heuristic must be admissible, costs along path must be monotonic. A heuristic h is admissible iff h(n) ≤ h\*(n), for all n. h\*(n) is the actual-path cost from n to the goal. Ie it must never over-estimate the cost. A heuristic h is monotonic iff h(n) ≤ c(n,a,n’) + h(n’) for all n,a,n’. n’ is a successor to n by action a, n to the goal “directly” should be no more than n to the goal via any successor n’. Pathmax modification: f(n’) = max(g(n’) + h(n’), f(n)).   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **Algorithm** | **Complete** | **Optimal** | **Time** | **Space** | | **Greedy Search** | Not always | No | O(bm), highly dependent on the heuristic’s performance | O(bm) | | **A\* Search** | Yes, unless x is infinite | Yes | O(x), x is the number of nodes n with f(n) ≤ f\* | O(x) | | **SMA\*** | Yes, if reachable in memory | Yes | O(x), x being number of nodes n with f(n) ≤ f\* | All of it |   Assessing Heuristics: Straight-line distance is obvious for travel, admissible. Heuristic Quality: The qualify of a heuristic is expressed as its effective branching factor (b\*). A good heuristic has b\* close to 1, choose between a heuristic with b\* closer to 1. Heuristic Dominance:­ h2 dominates h1 iff they are both admissible and h2(n) >= h1(n) for all nodes n. The dominating heuristic will usually visit fewer nodes. Always favourable. If neither dominates, use both: h(n) = max(h1(n),h2(n)). Deriving Heuristics: Given a problem p, a relaxed version p’ of p is derived by reducing restrictions on operators. The cost of an exact solution to p’ is often a good heuristic to use for p (never overestimates). Relax problems by eliminating one or more restrictions on operations. Memory bounded A\*: Limiting factor of A\* is space availability. IDA\* imposes an f-cost cut-off. Searches all nodes n such that f(n) ≤ k, increasing k as search continues. How much to increase k? doesn’t use all available space, only information communicated between iterations is f-cost limit. Simplified Memory-Bounded A\*: SMA\* expands the most promising node until memory is full, then drops nodes to continue search. When a node x is dropped, the f-cost of x is backed-up in x’s parent node. If at some point later in the search, all other nodes have higher estimates than the dropped sub-tree, it is re-generated. |
| **Game Playing:** Approaches to Uncertainty: Contingency planning: build all possibilities into the plan, causes large tree, only guarantees if number of contingencies is tractable. Interleaving/adaptive planning: alternate between planning, acting and sensing, requires extra work during execution, unsuitable for offline planning. Strategy Learning: Learn strategies that can be applied in any situation from examples, must decide on parameterisation, state-evaluation, suitable examples to study etc. Minimax Algorithm: 2 player game between MIN and MAX, Assume zero-sum game, good for MIN is bad for MAX. Assume utility function returns r ∈ R, ∞ if s is a win for MAX, positive if s is good for MAX, 0 if it is even, negative if s is good for MIN, -∞ if s is a win for MIN. Whenever MAX has the move in position s, they choose the move that maximises utility assuming MIN chooses optimally. Search depth = ply, searching 2 levels down = 2 ply. Minimax Performance: Complete: Yes for a finite tree, Optimal: Yes, against an optimal opponent, Time: O(bm), all nodes examined, Space: O(bm), depth-limited search. Standard approach is to apply a cut-off test and an evaluation function so that we can estimate the desirability of a position. Evaluation Functions: If we cannot expand to terminal nodes, we expand as far as we can and apply judgement to decide which positions are best, standard approach is linear weighted sum of relevant features. Quality of player depends highly on quality of eval function. Eval function should: Agree with the utility function on terminal states, reflect the probability of winning, be time efficient, to allow max search depth. Cutting off Search: We can cut-off search at fixed depth. Usually must manage time taken per move, use iterative deepening to cut off. Don’t cut uniform across the tree. Quiescence: A quiescent situation is one where values from the eval function are unlikely to change much anytime soon. Using fixed search-depth can mean relying on the evaluations of non-quiescent situations. Avoid by extending the search to the end of a series of events that would change the eval function. Horizon Effect: If we are searching to k ply, something bad that happens on k+1 ply will be invisible. No general solution. Alpha-Beta Pruning: Remove nodes that cannot be better than already seen nodes, enables deeper search. Good move ordering improves pruning. Perfect ordering can double our search depth. |
| **Sequential Decision Problems:** A problem where the utility obtained by an agent depends on a sequence of decisions. SDPs typically include utilities, uncertainty, sensing issues etc, which inhibit use of previously listed search algos. Policies: A policy is a set of state-action rules, for each state, which action to take? Providing a policy turns a utility-based agent into a simple reflex agent. Optimal policy depends on many factors. Depends on transition model, less certain actions imply more conservative policy. Depends on terminal utilities: bigger discrepancy between the two implies more conservative policy. Depends on step-cost, lower step-cost implies more conservative policy. Given a policy, we can determine the agent’s utilities if it follows that policy, vice versa, can determine best policy given utilities for each state. Bellman Equation: The utility of a state is specified by the bellman equation:  Ui = Ri + maxa ∑j MaijUj. Maij is the probability that doing action a in state i leaves the agent in state j (transition model). ∑j MaijUj is the weighted sum of all possible outcomes of doing action a in state i. maxa ∑j MaijUj is the expected outcome of the best action to do in state i. Ui is the cost of boing in state i, plus the optimal cost from then on. This equation underpins both SDP algorithms, cannot be solved directly due to mutual dependence and non-linearity. Value Iteration: Determine true utility of each state, then determine the optimal action in each state by action determination. To determine utility, use iterative approximation algo: start with arbitrary utility, update to make locally consistent with Bellman, repeat until utility is close enough. We can derive the optimal policy without knowing exact utilities by calculating the policy loss at each iteration by using the current value of U to derive “current policy” c, compare this with the optimal policy. Policy Iteration: Start with arbitrary policy pi, compute the utilities U of pi, by value determination, update pi according to U, by action determination, repeat until no change in pi. At each iteration, derive utilities from current policy, then check each state to see if action is optimal. If updated, iterate again. Utilities over time are discounted at rate of 1/y-1, where y is the position of state in sequence. |
| **Learning Agents:** 4 Components of learning agents: Performance element, chooses actions that are known to offer good outcomes. Learning element, improves the performance element, requires feedback on agent performance. Critic element, provides feedback, compares outcomes with objective performance standard. Problem generator, generates new experience, requires exploration – trying unknown actions which may be sub-optimal. Learning Element: has 2 separate goals. Improve outcome and time performance of performance element. 4 issues in design of learning element: components of the performance element to be improved, representation of those components, feedback available, prior information available. Performance Element: many components: a mapping from states to actions, means to infer information from percepts, info about how world evolves, info about effect of actions, utility info about states, goals which will increase utility. Each component might be improved by learning. Feedback Available: Supervised learning corresponds to being taught by an expert. Agent gets input-output pairs of problems and correct answers, agent learns a general rule that passes all these pairs. Reinforcement learning corresponds to learning from experience. Agent learns from the result of an action. Try something new and see if it works better, agent experiments and remembers what worked and what didn’t. Unsupervised learning happens in the absence of feedback. Learns patterns in the input, clusters similar inputs into sets and learns their outputs. Prior Knowledge: tabula rasa: agent starts with empty slate and basic skills OR agent starts with good design. Tabula rasa aims to explore new things, good design aims to exploit what we know. Decision Trees: Can be used as performance elements and can be created by learning elements. Can use greedy approach to find “good” tree. Test attribute that makes the most difference first. Divide each node until there are no longer positive and negative examples at a leaf. Test the tree with test data. Common approach is 90% training data 10% test data. Learning Under Uncertainty: Reasoning under uncertainty comes down to learning the probabilities of events, and how the probabilities are related. The joint probability distribution is the probability for combinations of events occurring. Bayesian networks represent events in a directed acyclic graph, where events are dependent on their parents, and otherwise conditionally independent. Bayesian Networks are a good method to take prior knowledge and compute conditional probabilities to support rational decisions. |
| **Reinforcement Learning:** Agent relies on feedback about its performance to assess its functionality. Basis of Reinforcement learning, use rewards to learn successful agent function. Aspects of RL: unknown environments must be learned if not known. Accessible environments are where the agent’s state can be identified by its percepts, inacessible environments require the agent to remember info about its state and recognise it by other means. Should rewards be given only in terminal states? Should rewards only be given in bulk? All feedback should be utilised. Passive Learning: Given a fixed agent function, learn the utilities of that function in the environment. Active Learning: no fixed function, agent must select action using what has been learned so far. Utility Learning: agent learns state utilities, then selects actions that maximise expected utility. Agent needs to know where actions can lead, must learn model of the environment, can be faster. Q-Learning: agent learns action-value function, the expected utility of taking action in a state. Doesn’t need to know where actions lead, just learns how good they are. Passive learning in known environment: Assume accessible, preselected actions, unknown effects of actions. Aim is to learn the utility function of the environment. The agent executes a set of trials in the environment. In each trial, the agent moves from the start state to a terminal state, the percepts identify the current state and the immediate reward. After a number of trials, we have the utilities of each state. Adaptive Dynamic Programming: ADP tries to learn faster. A passive situation finds state by the probability weighted average of its successors plus its own rewards: Ui = Ri + ∑j MaijUj. ADP estimates Maij from experience. This reduces learning to the value determination process. Process is intractable. Temporal Difference Learning: Exploit constraints between states, without solving for all states simultaneously. The idea is to use the observed transitions to adjust utilities locally to be consistent with Bellman. The update formula is Ui <- Ui + a(Ri + Uj – Ui), where a is the learning rate, higher a = more changes to Ui. Sometimes set a to decrease over time. TDL can be a crude approximation to ADP. Active Learning: In active learning, the agent not only needs to learn utilities, but also select actions. Agent needs to evolve its performance element by exploring its options. For learning utilities; each state maintains an estimated utility per action it can perform. Active agents must select actions that enable good performance and also learning about its environment. Must balance performance and learning. Optimal exploration policy is GLIE, Start whacky get greedier. Fundamental idea is to give weight to actions not tried often, and avoiding actions with low utilities. Regions near start are likely to be explored first. More-distant regions are more sparsely explored, so weight them higher. Q-Learning: means instead of learning the overall utility of State I, learn separately the utility of each action a that is available in i. No longer need to know the transition model. Learning via Q-values is slower. |
| **Logical Agents:** Knowledge based agents separate thinking from knowledge. Agent asks itself questions and tells itself new facts when making decisions.   |  |  |  | | --- | --- | --- | | ¬ not | ∧ and | ∨ or | | ∃ There exists | ∀ for all | → implies | | ⊢ proves | ⟷ iff | ⊨ first order logic | | Inference rules: | | | | Modus ponens | a, a -> B ⊨ B |  | | Modus tollens: | ¬B, a -> B ⊨ ¬a |  | | And-elimination: | a ∧ B ⊨ a |  | | Or-introduction: | a ⊨ a ∨ B |  |   A sentence is valid if it is true in all possible models, AKA tautology. A sentence is satisfiable if it is true in some model. 2 sentences are logically equivalent if true in same set of models. Entailment: An agent needs to ask its database if the current state of a database entails a fact, a ⊨B, a entails B if B follows logically from a. Checking via truth tables. To check whether a ⊨B, Determine which rows give a=true, if B is true in all those rows, then a ⊨B. Inference Systems: An inference system is a set of rules for deriving new sentences that are entailed by existing sentences. Inference systems are only sound if it only does correct derivations. Complete if it does all correct derivations. Resolution: proof by resolution has 3 steps. 1. Convert the agent’s database to conjunctive normal form (CNF). 2. Negate the query and add it to the database. 3. Repeatedly apply the resolution principle to try to demonstrate a contradiction. Proof by contradiction: If KB is assumed true, and if KB ∧ ¬Q is false. Then Q must be true. CNF: A clause is a disjunction of literals, a sentence in CNF is a conjunction of clauses. Every propositional sentence can be converted to logically-equivalent sentence in CNF by a simple recursive procedure. Applied Below.   |  |  | | --- | --- | | S1 ⟷ S2 | (S1 → S2) ∧ (S2 → S1) | | S1 → S2 | ¬S1 ∧ S2 | | ¬(S1 ∨ S2) | ¬S1 ∧ ¬S2 | | ¬(S1 ∧ S2) | ¬S1 ∨ ¬S2 | | S1 ∨ (S2 ∧ S3) | (S1 ∨ S2) ∧ (S1 ∨ S3) | | ¬¬S | S |   Pros and Cons and Prop logic: Pros: Syntax corresponds to facts. Allows partial information. Cons: Limited expressive power, cannot make general statements like pits cause breezes in adjacent squares.  First-Order Logic: many different entities. Objects: people, houses. Predicates: isRed, isRound (bool). Functions: nextDoor, plus, etc, describes properties of objects. Variables: x,y, describing properties of sets of objects. Connectives: as for propositional logic. Equality: for identifying two objects. Quantifiers: for all, there exists. |
| **Planning:** Partial-order planning: In planning search: a node is a partial plan (set of steps), Operator modifies a partial plan by adding a step, changing order of steps, or sets a bound variable. Search proceeds until plan is complete. A partial order planner places an order on steps in its plan only when it is essential to do so. Final result is a directed acyclic graph, where nodes are steps, edges are orderings of steps. Topological Sorting: turns a graph from partial order into a total order that respects the same dependencies. Clobbering: Make sure no step clobbers plan. If a step eliminates possibility of future steps, that is clobbering, the offending step must either be demoted (put before clobbered step), or promoted (put after clobbered step). If neither is possible, must backtrack and make new plan. Conditional Planning: Anticipate problems and build contingencies into the plan. Add observation actions to check preconditions and effects, create sub-plan for each possibility. Expensive because it plans for many unlikely cases. Monitoring and Replanning: Assume normal successful execution. Continually check the progress of the plan, replan when necessary. Monitoring Types: Action monitoring, agent checks if preconditions of next action are met. Plan monitoring, agent checks whether the preconditions of the remaining plan are met. Goal monitoring, Agent checks whether changing its goals is appropriate. An agent can either replan from scratch or replan to restore the expected state so that the rest of the plan can continue. Situated planning: Agent interleaves planning and execution. Execute step in plan, monitor world state, fix plan deficiencies, refine plan in light of new information. |