1. **Agent-environment**

|  |
| --- |
| Rational agents |
| For each possible percept sequence, a rational agent should select an action that is expected to maximise its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.   * **Percept**: tuple of sensor inputs at time step * **Percept** history: set of past percepts * **Actions**: set of actions, enacted by actuators * **Agent** **function**: {p1 …} -> a |
| Problem environments |
| * Fully vs. partially observable * Deterministic vs. stochastic * Episodic vs. sequential * Single vs. multi- agent * Static vs. dynamic * Discrete vs. continuous * Known vs. unknown |

1. **Path search problems**

|  |
| --- |
| Definitions |
| * **State**: ADT describing instance of environment * **Goal test**: state -> 1 or 0 * **Action function**: state -> {a1, a2 …} * **Action cost function**: state1, action, state2 -> c * **Transition model**: state, action -> new state * **Solution**: sequence of *actions* that form a *path*, and a solution is a path from the initial state to a goal state. Optimal solution has lowest path cost among *all* solutions. * **Search** **tree**: graph describing the paths between states in the state space traversed by the algorithm. Search tree consists of nodes. Each node in the tree has a unique path to the root. * **Node**: ADT with node.state, node.parent, node.action, node.pathcost * **Expanding**: passing node through action function and transition model to generate a new node for each of the resulting states. Expand != will be in frontier! * **Reached**: *state* is *reached* iff. has had a node generated for it * **Explored**: node is *explored* iff. goal tested *and* expanded. * **Frontier**: queue that tracks nodes *to be explored*. Separates 3 regions in state space, explored, reached but not explored, not reached. Q contains reached but not explored. |
| Path-search problem formulation |
| * State ADT * Goal test, initial/goal state * Action function * Transition model |
| Definitions for search algorithms |
| * **Completeness**: iff. it will find a solution when one exists and correctly report failure when one does not. Note if infinite & no solution -> cannot be complete * **Optimal**: iff. always returns a solution with the lowest path cost if it exists * Optimal depth d, maximum depth m, branching factor b * **Repeated state**: state that is explored/expanded despite having been reached. * **Redundant path**: path is redundant if there is another way to reach the same terminal node at lower cost * **Graph search**: check for redundant paths. Keep track of all previously reached states, explore iff. state is not reached OR lower path cost has been found. * **Limited graph search**: keep track of all previously reached states, explore iff. state has not been reached. Adds to reached on pop. * **Tree search**: do not check for redundant paths, allowing repetition. |

1. **Uninformed search**

|  |  |
| --- | --- |
| BFS | |
| Time | * Tree: 1 + b + b2 … = O(bd) * Graph: O(V + E) = O(bd) * Limited graph: O(V) = O(bd) |
| Space | * Tree: all leaves, O(bd) * (Limited) graph: reached table, O(V) = O(bd) |
| Complete | * Tree: no cycles, finite space and b   (Limited) graph: finite space and b |
| Optimal | Iff. all actions have same cost |
| * Late vs. early goal test: late tests on pop(). Suppose goal at depth d. Then generated 1 + b + b2 ... bd + (bd – 1)b, since all but last generates b children. Testing before push() will only generate 1 + … bd goal at worst. | |
| UCS | |
| Time | * Tree: O(b1 + c\*/e) * Graph: O(V + E) = O(b1 + c\*/e) * Limited graph: O(V) = O(b1 + c\*/e) |
| Space | * Tree: all leaves, O(b1 + c\*/e) * (Limited) graph: reached table, O(V) = O(b1 + c\*/e) |
| Complete | * Tree: no cycles, finite space and b   (Limited) graph: finite space and b |
| Optimal | Iff. all actions have non-negative cost. |
| * Time & space complexity: suppose optimal cost to goal is c\*, then we know that there are at most c\*/e edges to goal. Also do late goal test for optimality, so that PQ can do arrangement. | |
| DFS | |
| Time | * Tree: linear chain O(bm) * Graph: O(V + E) = O(bm) * Limited graph: O(V) = O(bm) |
| Space | * Tree: linear path of branches, O(bm) * (Limited) graph: reached table, O(V) = O(bm) |
| Complete | * Tree: no cycles, finite space and b   (Limited) graph: finite space and b |
| Optimal | No in general, returns first solution found. |
| DLS – DFS with depth restriction | |
| Time | * Tree: 1 +b … = O(bm) * Graph: O(V + E) = O(bm) * Limited graph: O(V) = O(bm) |
| Space | * Tree: linear path of branches, O(bl) * (Limited) graph: reached table, O(V’) = O(bl), vertices at limited d. |
| Complete | Iff. has goal & l >= d, finite actions. Cycles can be eventually cut off.  Often choose l by diameter of state space. |
| Optimal | No in general, returns first solution found. |
| IDS | |
| Time | * Tree: 1 + db + (d-1)b2 … bd = O(bd) * Graph: O(V + E) = O(bd) * Limited graph: O(V) = O(bd) |
| Space | * Tree: linear path of branches, O(bl) = O(bd) * (Limited) graph: reached table, O(V’) = O(bl) = O(bd) , vertices at limited d. |
| Complete | Finite space and b. Cycles can be eventually cut off. |
| Optimal | Iff. all actions have same cost |
| * Repeated computation: nodes at bottom are generated once, while children of root at depth i, d - i times. Note that root trivially never repeated. * Memory vs. time: IDS does BFS in a DFS way, trading some time (repeated computation) for space. | |

1. **Informed search**

|  |  |
| --- | --- |
| Definition | |
| * Estimated cost of the *cheapest path* from the *current state* at node n to the *nearest* goal state. | |
| Properties | |
| * **Admissibility**: ∀n, h(n) <= h\*(n). note h(goal) = 0. Prove by induction or counter example. * **Dominance**: ∀n h1(n) >= h2(n). Prove by induction or two-way counter example. * **Consistency**: ∀n, n’ that is a child of n generated by action a, h(n) <= h(n’) + c(n, a, n’). A consistent heuristic *enforces* the constraint of monotonically increasing path costs in its estimates. Heuristic considers a detour to be at least as costly as a direct route (ie. uses information about state that signals “progress towards goal”). * **H(n) consistent, g(n) + h(n) is non-decreasing along any path**. **Consistent -> admissible**, by backward induction. | |
| Creating admissible heuristics | |
| * Relax game * Optimal solution main game is a solution to relaxed game. Optimal cost in relaxed game is admissible heuristic for main game. | |
| Greedy best first search | |
| Time | * Tree: 1 + b + b2 … = O(bd) * Graph: O(V + E) = O(bd) * Limited graph: O(V) = O(bd) |
| Space | * Tree: all leaves, O(bd) * (Limited) graph: reached table, O(V) = O(bd) |
| Complete | * Tree: no. Heuristic can cause cycle * (Limited) graph: finite space and b |
| Optimal | No in general since does not take *actual cost* into account. |
| A\* search | |
| Time | * Tree: 1 + b + b2 … = O(bd) * Graph: O(V + E) = O(bd) * Limited graph: O(V) = O(bd) |
| Space | * Tree: all leaves, O(bd) * (Limited) graph: reached table, O(V) = O(bd) |
| Complete | * Tree: no cycles, finite space and b * (Limited) graph: finite space and b |
| Optimal | * Tree & graph: yes if h(n) admissible * Limited graph: h(n) consistent |
| * **Optimality under admissibility**: tree & graph search will expand a node so long as not at goal, updating g(n) explicitly (lazy add to PQ, update PQ) or implicitly (allowing revisits), so will find optimal goal before non-optimal. Optimal goal must be popped before sub-optimal goal. * **Optimality under consistency**: limited graph does not revisit nodes. So consistency ensures once popped from frontier, optimal path to a given node must be found. Suppose you at s, and you know A and B, and A has lower cost. Now we discover a detour to goal from B via C. Because heuristic is consistent, we know that if A < B previously , then A < B + detour to C still. Can proof by contradiction by using property of monotonically increasing f(n) and order of pop by PQ. * **Effect of heuristic on performance**: efficiency of A\* (i.e. nodes explored) depends on h vs. h\* . if admissible and dominates, then closer, and *at least as efficient* as dominated h. | |

1. **Goal search via hill climbing**

|  |
| --- |
| Local search |
| Algorithms that operate by searching from a *start state (complete)* to neighbouring states, without keeping track of paths or set of states that have been reached. |
| Definitions |
| * **State/initial state:** ADT that describes environment. A potential solution. * **Value function:** evaluates “quality” of state in terms of how close it is to be a goal state. Value for a goal state should be unique. * **Successor generation:** state -> states. Given a state, how the next possible solutions should be guessed. * **Algorithm and stopping criteria** |
| Hill-climbing search algorithms |
| * **Hill climbing:** algorithm starts at an initial state. Keeps track of one current state, and generates successor, moving towards a neighbouring state with the *highest value* (direction of steepest ascent). If no neighbours higher, terminates. F = -h(n) for descent. * **Problems with hill climbing**: local maxima, plateaus, shoulders, flat local maxima * **Stochastic hill climbing** – chooses at random among all *uphill* moves * **first choice hill climbing**: implements stochastic hill climbing by generating successors randomly, until one is generated that is better than the current state. * **Random restart hill climbing**: terminal state might be a local maxima and not a goal state. So keep restarting with a random initial state, until solution is found. Use Bernoulli variables to compute. * **Sideways movement:** modification to stopping criteria, continue even if neighbour is equal to current. |
| Local beam search |
| * Initialize k states. For each k states, generate O(kb) successors, choose top k among all kb successors. If any of top k is goal, return. Else keep repeating. * Stochastic beam search: choose top k by sampling with probability proportional to relative valuation. |
| Local search formulation |
| * State (solution structure) * Initial state (initial solution guess) * Successor generation strategy (how to modify current guess, can be random!) * Stopping criteria & climbing algorithm * Evaluation function |

1. **CSPs**

|  |
| --- |
| Idea |
| Model a problem by variables, domain and constraints. Possibilities -> try something -> feedback -> update possibilities. Idea that any state that does not satisfy a constraint should not be further explored. |
| Definitions |
| * **Variable/X:** set of variables to be solved for. * **Domain/D**: set of domains of each variable, values that each variable can take. * **Constraints/C:** set of constraints all variables must conform to. Try to formulate with symbolic logic and mathematical values. * **Assignment**   + Complete – every variable assigned   + Consistent – no constraints violated   + Solution: complete & consistent assignment   + Partial – some variables unassigned   + Partial solution: partial assignment that is consistent (so far) |
| CSP formulation |
| * **State representation** – X, D, C, in MATHEMATICAL/NUMERICAL representations * **Goal test**: test that no constraints violated * **Actions:** assignment of value to variable |
| Backtracking algorithm |
| * **Naive BFS**   + Redundant computation since order is considered. At depth l, (n-l) \* m branches, since can choose any of n-l remaining unassigned variables, each with domain size max of m.   + Then total no. of states = nm + (n-1)m + … = n!mn * **Backtracking**   + Go down the search tree in a DFS way. Order doesn’t matter. So at depth l, branching factor is |d|, and at most mn states. |
| Variable ordering |
| * **Minimum remaining values:** choose variables with least remaining values first. This way, you can prune *larger* subtrees * **Degree heuristic:** variables w/ most constraints first, downstream branching factor smaller (since domain smaller from more constraints) |
| Value ordering |
| * **Least constraining value heuristic:** Choose values that keep options most open, can avoid failure and get to solution faster. Makes no difference if *all* solutions need to be found. |
| Consistency |
| * **Node consistency:** domain is consistent with unary constraints (i.e. valid domain) * **Arc consistency:**    + Xi is arc consistent with Xj <-> for every value in Di, there is *some* value in Dj that satisfies the binary constraint (Xi, Xj).   + Every binary constraint as TWO arcs (both ways) |
| Forward checking |
| * Given partial assignment and constraints, update domains of all *unassigned variables* to remaining legal values. If any downstream domain becomes empty, terminate. * Need to check n nodes, all neighbours of each node and update domains accordingly. Binary constraint graph, NC2 x 2 = n(n-1) checks, O(n2). * Each check from Xi to Xj at most |d| values, nested for loop then O(d2) * **So total O(n2d2)** |
| AC3 |
| * Generate initial 2 x NC2 queue of arcs in both directions. Check for arc consistency and eliminate illegal values for each arc. * If Di is updated, we queue all INCOMING arcs (Xk, Xi), since the value of Di that made Xk consistent wrt to Xi may no longer be there. * If Di empty, failure. Queue empty, success. * Need to check n nodes, all neighbours of each node and update domains accordingly. Binary constraint graph, NC2 x 2 = n(n-1) checks, O(n2). * Each check from Xi to Xj at most |d| values, nested for loop then O(d2) * Each arc enqueued at most n times. * **So total O(n3d2)** |

1. **Adversarial search**

|  |
| --- |
| Game formulation |
| * Zero-sum games: games where what is good for one player is just as bad for the other. * Constant-sum games: u(p1) + u(p2) = k always * **State/initial state:** ADT that describes how the game is set up/state of the game. * **To-move(s)**: function that checks whose move * **Action(s)**: function that returns *set* of legal moves at state s * **Result(s, a)**: transition model which returns resulting state * **Is-terminal(s):** returns true if winning condition or maximal depth reached * **Utility(s, p):** evaluates state. Has to be between losing and winning value if non-terminal, and exactly win/loss if terminal. |
| Strategy |
| * Strategy: set of moves p1 will play *at every node* of the game tree that p2 plays. State, p2 move -> p1 move * Winning strategy: for all strategy s2 by p2, the game ends with p1 winning. * Non-losing strategy: for any strategy played by p2, the game ends in tie or win for p1. |
| Backward induction & minimax |
| * **Backward induction:** Look ahead with perfect information, assume optimal plays, find dominant strategy for that player at that node, backpropagate the value up the game tree * **Minimax search:** Search move by backward induction.   + Completeness: yes, if game tree finite   + Optimal: yes, if MIN plays optimally   + Time: O(bm), searching whole tree of depth m   + Space: O(bm) by DFS |
| Heuristic design |
| * **Heuristic design**: fast to compute, linear weighted sum of *features* of *state* * **Heuristic cut-off Minimax search:** Search move by backward induction but cut off at depth m.   + Completeness: if game tree depth <=m   + **Optimal:** if MIN plays optimally and game tree depth <= m   + **Time:** O(bm), searching whole tree of depth m   + **Space:** O(bm) by DFS |
| Alpha beta pruning |
| * **Idea:** keep track of best option seen by grandparent node when we explore at level of child node, eliminating if pointless even if backprop the value. * **Sanity check**: outcome same as minimax * **Alpha:** lower bound (worst case) on MAX choice’s above so far * **Beta:** upper bound (worst case) of nearest parent MIN so far * **How:**    + Draw tree, propagate up every time a subtree is complete. When going back down, compare to levels above to see if it will change outcome.   + If at children of MIN node, if value coming up to child is <= ALPHA, stop exploring other children   + If at children of MAX node, if value coming up to child is >= BETA, stop exploring other children * **Pruning order:** makes a big difference   + Perfect move ordering: O(bm/2)   + Random ordering: O(b3m/4)   + Right-to-left, left-to-right may matter, depending on how children are generated |

1. **Knowledge & logical agents**

|  |
| --- |
| Definitions |
| * **Model:** model corresponds to a *set* of variable value assignments, applied to sentence. V models alpha <-> α is true under v. **M(α) = set of all models that α is true** * **Entailment**: α |= β ⬄ M(α) ⊆ M(β) ⬄ α follows from β ⬄ α 🡪 β * **Validity:** tautologies, sentence always true * **Satisfiability:** there exists some assignment such that sentence is true. Contradictions not satisfiable. |
| Inference |
| * **Inference** * **Soundness:** Inference algorithm is sound if α derived from β (β -> α) and indeed β |= α * **Completeness:** β |= α = (β -> α) -> algorithm ALWAYS derives α from β for all α |
| CNF |
| * **Conversion to CNF**   + a ⬄b to a -> b AND b -> a   + a 🡪 b to ~a OR b   + negations into brackets * **Conversion to 3 CNF (reverse resolution)**   A1 OR … An ⬄ (b1 OR A1 OR A2) AND (~b1 OR A3 …) |
| Resolution |
| 1. Convert to CNF 2. Find matching clauses with ~x and x, conjoin them, put back 3. (CNF1 OR x) and (CNF2 OR ~x) -> CNF1 OR CNF2   **Derivation** by CNF1 or x = ~CNF1 -> x, CNF2 or ~x = x -> CNF2. By transitivity, ~CNF1 -> CNF2 = CNF1 OR CNF2 |
| Resolution algorithm |
| * **Steps:** Convert KB to CNF. Add ~ α to list of clauses. Repeatedly resolve until empty clause (true) or no other way (false). * **Soundness:** KB |= α ⬄ KB -> α ⬄ KB and ~ α not satisfiable. CNF requires 1 literal to be true/satisfiable. Algorithm infers when empty clause from resolution. * **Completeness:** So long as entails, will result in empty clause by iterative resolution, by induction. |
| Knowledge based agent formulation |
| * **Knowledge base:** sentences known to be true * **MAKE\_PERCEPT\_SENTENCE(percept)**: convert percept to sentence * **TELL:** updates,sentence, KB -> KB’ * **MAKE\_ACTION\_QUERY(t):** at time step, generate all queries * **ASK(KB, queries):** returns action to take after querying KB and processing results of queries. |