|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Agent Model**  Perceives its environment through sensors and acting upon that environment through actuators   * Percepts – Sensors * Actions – Actuators * Environment * Performance Measure   **Rational Agent**  Select actions that **maximises its (expected) utility**  Percepts, Env, Action Space 🡪 Action Selected  i.e., Specified by an agent function  **Rationality**  What is rational at a given time depends on PEAS:   * **P**erformance measure * Prior **E**nvironment knowledge * **A**ctions/Actuators * Percept sequence to date (**S**ensors)   Limitations:   * Percepts may not provide all the required info (Rationality omniscience) * Actual outcome of actions may not be as expected (Rationality clairvoyant)   **Environment Types**   * **Fully observable (vs. partially observable):**  An agent's sensors give it access to the complete state of the environment at each point in time. * **Deterministic (vs. stochastic):**  The next state of the environment is completely determined by the current state and the action executed by the agent.   + If the environment is deterministic except for the actions of other agents, then the environment is **strategic** * **Episodic (vs. sequential):**  The agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action), and the choice of action in each episode depends only on the episode itself. * **Static (vs. dynamic):** The environment is unchanged while an agent is deliberating.   + (The environment is **semi-dynamic** if the environment itself does not change with the passage of time, but the agent's performance score does) * **Discrete (vs. continuous):**  A limited number of distinct, clearly defined percepts and actions. * **Single agent (vs. multiagent):**  An agent operating by itself in an environment   **Search Problem Formulation**   * **State space**, e.g. At(Arad), At(Bucharest) * **Initial state**, e.g. At(Arad) * **Actions**, set of actions given a specific state   + **Transition model** e.g., Result(At(Arad),Go(Zerind)) → At(Zerind)   + **Path cost** (additive), e.g., sum of distances, number of actions, etc * **Goal test**, can be   + Explicit, e.g. At(Bucharest)   + Implicit, e.g. checkmate(x)   **Search Problem Solution**   * A **solution** is a sequence of actions from the initial state to a goal state   (E.g., Arad → Sibiu → Fagarus → Bucharest)   * An **optimal solution** is a solution with the lowest path cost | **General Search**   * **Root** = Initial State, **Leaves** = Generated State * **State** is a repr of a physical configuration * **Node** is a data structure constituting **part of a search tree** (Comprises of state, parent, child, action path-cost, depth) * **Expand** function creates new nodes   + Uses Actions and Transition Model to create corresponding states   **Search Strategies**  Defined by picking the order of node expansion  Evaluated through:   * **Completeness** - find a solution if one exists? * **Optimality** - least-cost solution? * **Time** **complexity** - number of nodes generated/expanded * **Space** **complexity** - maximum number of nodes in memory   Time and space complexity measured in terms of:   * **b** – max branching factor * **d** – depth of least-cost solution * **m** – max depth of state space   A picture containing chart  Description automatically generated  **General Tree Search**  Diagram  Description automatically generated  Problem: Repeated states (Redundant paths can cause a tractable problem to become intractable)  **A picture containing text  Description automatically generatedGeneral Graph Search**  **Types of Searches**   * **Uninformed Search** No extra info about states beyond that in the problem definition * **Informed Search** Uses problem-specific knowledge beyond the definition of the problem itself * **Adversarial Search** Used in multi-agent environment where the agent needs to consider the actions of other agents and how they affect its performance | **Breadth-First Search**  General idea: Expand shallowest unexpanded node  Implementation: Use First-In First-Out (FIFO) queue   * **Completeness:** Yes (if b is finite) * **Optimality:** Yes (if cost=1 per step; Not optimal in general) * **Time** **complexity:** * **Space** **complexity:**, all node in memory   Problems: Memory Requirements & Execution Time  **Uniform Cost Search**  General idea: Expand unexpanded node n with the lowest path cost g(n)  Implementation: Using a priority queue ordered by path cost g   * **Completeness:** Yes (if step cost > ε, some positive constant) * **Optimality:** Yes * **Time** **complexity:** , where C\* = cost of optimal solution * **Space** **complexity:**   Problems: Possible redundant searches  **Depth-First Search**  General idea: Expand deepest unexpanded node  Implementation: Use Last-In First-Out (LIFO) queue   * **Completeness:** No (if m is infinite) * **Optimality:** No * **Time** **complexity:** , bad if m>d by a lot * **Space** **complexity:**, linear space   **Depth-Limited Search**  General idea:  Depth-First Search with predetermined depth limit  (Nodes at depth have no child nodes & Solves infinite-path problem)   * **Completeness:** No (if ) * **Optimality:** No (if ) * **Time** **complexity:** * **Space** **complexity:**   **Iterative Deepening Search**  General idea: Use increasing Depth-Limited Search (DLS) to find the best depth limit l   * I.e., use DLS with depth limit 1. If no solution, then increase depth limit to 2. So on and so on, until solution is found   Best of both Breadth-First Search and Depth-First Search   * **Completeness:** Yes * **Optimality:** Yes * **Time** **complexity:** * **Space** **complexity:**   **BFS vs DFS**  Use BFS when:   * Optimal solution is important * m is much greater than d   Use DFS when:   * Space is important. DFS: BFS:   **Heuristics**   * The heuristic function h(n) is an estimate of how close a state n is to the goal state * Informed search algorithms use heuristics to solve the search problem   **Greedy Best-First Search**  General idea: Expand the node n with the lowest heuristic h(n)  Implementation: Use a priority queue ordered by heuristic h(n)   * **Completeness:** No (Can get stuck in loops, unless we keep track of repeated nodes) * **Optimality:** No * **Time** **complexity:** * **Space** **complexity:**, all nodes in mem | **UCS vs G-BFS**   * UCS is complete and optimal but may waste search in the wrong direction * Greedy search generally in the correct direction but not complete or optimal * Combine UCS & G-BFS 🡪A\* Search   **A\* Search**  General idea: Expand the node n that has incurred the least cost and is nearest to the goal state  Implementation: Using a priority queue ordered by eval. func. f(n)   * Evaluation function f(n) = g(n) + h(n) * Path cost g(n) = total path cost from start node to node n * Heuristic h(n) = estimated distance from node n to goal state * **Completeness:** Yes (if step cost > ε, some positive constant) * **Optimality:** Yes (If heuristics are admissible/consistent) * **Time** **complexity:** , where C\* = cost of optimal solution * **Space** **complexity:**   Applications:   * Path finding problems * Video games * Resource planning problems * Robot motion planning   **Heuristic Properties**  **Admissibility**: A heuristic h(n) is admissible if . For example:   * = estimated distance from node n to goal state * = true cost from node n to goal state   **Consistent**: A heuristic h(n) is consistent if . For example:   * h(n) = estimated distance from node n to goal state G * h(n’) = estimated distance from node n’ to goal state G * c(n,a,n’) = cost of getting from node n to n’   **Dominance**: A heuristic dominates if , for all n.   * Only if both heuristics are admissible * A more dominant heuristic will be better for search (Potentially explore less branches)   **Designing Heuristics**  Admissible heuristics can be derived from the exact solution cost of a relaxed problem  **Constraint Satisfaction Problems**  **State**   * Defined by variables that take on values from domain   **Goal Test**   * A set of constraints specifying allowable combinations of values for subsets of variables   In contrast to standard search problems   * State is a “black box” - any old data structure that supports goal test, eval, successor   CSP comprises of:   * Finite set of variables * Non-empty domain D of k possible values for each variable Di, where * Finite set of constraints * Each constraint limits the values that variables can take, e.g.,   **Complete**: Every variable is assigned  **Consistent**: Does not violate any constraint  **CSP** **solution**: Complete & Consistent assignment | **Advantages of CSP**   * **Formal representation language** that can be used to formalize many problem type * Represent problem as a CSP and solve with general-purpose solver * Can use **general-purpose solver**, which are more efficient than standard search * Constraints allow us to focus the search to valid branches * Branches that violate constraints are removed * Non-trivial to do this for standard search (need manual selection of actions)   **Constraint Graph**  Nodes = Variables, Edges = Constraints  **Variety of CSPs**  Discrete Variables   * Finite domains: complete assignments for n variables, domain size d * Infinite domains: Integer, Strings, etc.   Continuous Variables   * Time, float, etc.   **Variety of Constraints**   * **Unary**: Involve single variable * **Binary**: Involve pairs of variables * **Higher order**: Involve 3 or more variables * **Preference** (Soft constraints)   **CSPs as Standard Search**   * Can be easily formulated (Initial State, Actions, Path Cost, Goal State) * Sequence of actions do not matter, only the goal state (i.e., solution at depth n, use DFS) * However, there are potentially leaves   **Commutativity**   * CSP variable assignments are commutative: i.e., Regardless of variable assignment order * Only need to consider assignments to a single variable at each level/depth * reduce from leaves to leaves   **Backtracking Search**   * DFS for CSPs with single variable assignment * Backtracking occurs when there are no legal values for a variable * The basic uniformed algorithm for CSPs   **General Purpose Methods**  Can give huge gains in speed:  **Minimum remaining values**   * Choose the variable with the fewest legal values (i.e., the most constrained variable)   **Degree Heuristic**   * + When multiple variables have the same MRV   + Choose the variable with the most constraints on remaining variables   **Least constraining value**   * + Given a variable, choose the least constraining value   + The one that rules out the fewest values in the remaining variables   **Forward Checking**   * + Keep track of remaining legal values for unassigned variables   + Terminate search when any variable has no legal values   + propagates information from assigned to unassigned variables   + Does not provide early detection for all failures   + Need to enforce constraints locally | **Arc Consistency**   * Simplest form of propagation make each arc consistent * is consistent iff. for every value X there is some allowed y * Arc consistency detects failure earlier than forward checking * Can be run as a pre-processor or after each assignment * Ordering of arcs do not matter * Complexity of * : Need to check for all edges, potentially edges * : For each edge, comparing their two domains * : Each variable change reprop-agate to neighbours, max d times   Text  Description automatically generated  **Problem Structure**   * Suppose each subproblem has c variables out of n total * Worst-case solution cost is   **Tree-Structured CSPs**   * if the constraint graph has no loops, the CSP can be solved in time  1. Choose a variable as root, order variables from root to leaves such that every node’s parent precedes it in the ordering 2. For j from n down to 2, apply RemoveInconsistent(Parent,) 3. For j from 1 to n, assign consistently with Parent()   **Nearly Tree-Structured CSPs**   * Conditioning: instantiate a variable, prune its neighbours’ domains * Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree * Cutset size c ⇒ runtime , fast for small c   **Representing Game as Search Problem**   * Initial State * Actions * Terminal Test (Win/Lose/Draw) * Utility Function (Numerical reward for the outcome): E.g., Chess 🡪 +1, 0, -1  Poker 🡪 Cash win or lose * Zero-sum: each player’s utility for a state are equal and opposite   **Minimax**   * Perfect play for deterministic, perfect-information (fully observable) games * Idea: choose moves with highest minimax value * best achievable payoff wrt best play * **Completeness:** Yes, if tree is finite * **Optimality:** Yes, against optimal opponent * **Time** **complexity:** * **Space** **complexity:** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Chart, radar chart  Description automatically generated  Graphical user interface, text, application  Description automatically generated  **Cutting Off Search**   * MinimaxCutoff is identical to MinimaxValue except * Terminal is replaced by Cutoff? * A picture containing chart    Description automatically generatedUtility is replaced by Eval   **α-β Pruning**   * Pruning does not affect final result * Good move ordering improves effectiveness of pruning * With “perfect ordering”, time complexity is O(bm/2) * doubles depth of search * can easily reach depth 8 and play good chess * A simple example of the value of reasoning about which computations are relevant (a form of meta-reasoning)   Chart, radar chart  Description automatically generated  **ExpectiMiniMax (Non-deterministic Games)**   * Accounts for chance nodes * If state is a chance node, return weighted average expected values of child nodes   A picture containing chart  Description automatically generated | **Breadth-First Search**  **A picture containing indoor, device  Description automatically generated**  Text, letter  Description automatically generated  **Uniform-Cost Search**  **A picture containing indoor, different  Description automatically generated**  Text, letter  Description automatically generated  **A picture containing indoor, device  Description automatically generatedDepth-First Search**  **Chart, bubble chart  Description automatically generatedDepth-Limited Search**  **A picture containing different, arranged  Description automatically generatedIterative Deepening Search** |  |  |  |  |