Lecture Notes
CS255 - Artificial Intelligence

Rational Agents

1. **Agent** is an entity that perceives P and acts A, hence is a function or a mapping $f : P \to A$.
2. P={abilities, goals, stimuli, past knowledge/experiences}.
3. **Rational action** is one that maximises the expected value of performance measure given the percept sequence.

Dimension	Possible values
Modularity	flat, modular, hierarchical
Planning horizon	static, (in)finite/indefinite stage
Representation	states, features, relations
Compute limits	perfect/bounded rationality
Learning	knowledge given, knowledge learned
Sensing uncertain.	fully observable, partially observable
Effect uncertainty	deterministic, stochastic
Preference	goals, complex preferences
Number of agents	single agent, multiple agents
Interaction	offline, online

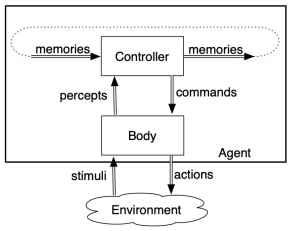
4. **Modularity**: one level of abstraction is **flat**, separate modules understood separately is **modular**, modules recursively decomposed into other modules is **hierarchical**.
5. **Planning horizon**: **Static** (non-planning) world doesn't change, if agent reasons about a fixed finite number of steps, then **finite**, if finite but not predetermined, then **indefinite**, if agent goes forever, then **infinite** stages.
6. **Representation**: **State** is one way the world could be, **features** (propositions) describe a state, **relations** are associations between distinct objects, or **individuals**.
7. **Computational limits**: In **perfect rationality**, agent determines best action without a resource bottleneck, if it's **bounded**, then have to act according to limitations.
8. **Uncertainty** has **sensing** and **effect** dimensions. Agent can know what is true (**no uncert.**) or $\exists$ set of possible states (**disjunctive uncert.**), or there is probability distribution over the worlds (**probabilistic uncert**).
   - definitive predictions: your laptop will run out of power
   - disjunctions: charge for 30 minutes or you will run out of power
   - probabilities: probability you will run out of power is 0.01 if you charge for 30 minutes and 0.8 otherwise

   Uncertainty can view all world states (**fully-observable**) or only some states given agent's stimuli (**partially-obs**).

   Deterministic agent knows result state given the action and the current state, **Stochastic** doesn't.
9. **Preferences** (goal) can either be a specific achievement, or **complex** (involves tradeoffs), where order (**ordinal**) or absolute values (**cardinal**) of the parameters matter
10. **Interaction**: agents can reason before (**offline**), using knowledge base or between (**online**) receiving information and acting. Can be **multiagent** - agent reasons strategically about reasoning of other agents, else **single agent**.
11. **Representation**: rich, compact, natural, maintainable.
12. **Optimal** solution is the best, **satisficing** is good enough, **approximately optimal** is close the best theoretically possible, and **probable** solution is a likely one. **Common-sense reasoning**: making conclusions about the unstated assumptions.
13. Can quantify the **value of information**; **anytime algorithm** can provide a solution at any time, performs better given more time.
14. **Symbol** is a meaningful physical pattern that can be manipulated by the **symbol system**. **PSS Hypothesis**: it is sufficient for GAI. Knowledge/symbol level is in terms of knowledge/reasoning it has/does.

Agent Architect., Hierarchical Control

1. Agent interacts with environment through its body made up of **sensors** that interpret stimuli (*percepts*) and **actuators** that perform actions (*commands*). **Controller** receives percepts from the body.

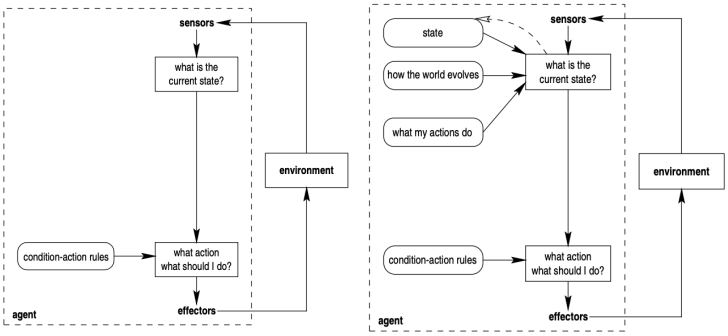


2. Let t be set of time points. **Percept/Command trace** (PT/CT) is sequence of all percepts/commands ever received/output by the controller. **Transduction** is a function $f : PT \to CT$ and is **causal** (controller) if all CT up to time t depend only on percepts up to t.
3. Agent only has access to what is remembered, so memory of **belief state** encodes all of the agent's history that it has access to at time t.
4. Controler has belief state `remember(belief_state,percept)` func that returns next belief state, command `do(memory, percept)` func that returns the command for the agent, and `higherPercept(memory,percept,command)` that uses hierarchy of lower-level percepts and past memories.

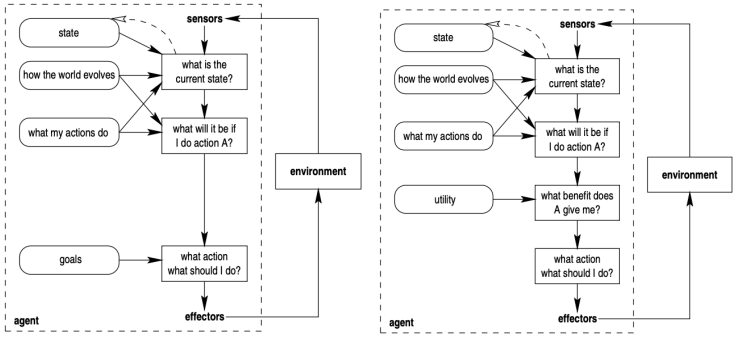
5. Hierarchical system (controller) architecture comprises layers, and is faster, more flexible than serial approach.

6. **Purely reactive** agent doesn't have memory, **dead reckoning** agent doesn't perceive the world. Both aren't useful in complex worlds, so need to create a rational $f : P \to A$ mapping or a trigger to action lookup table approximation

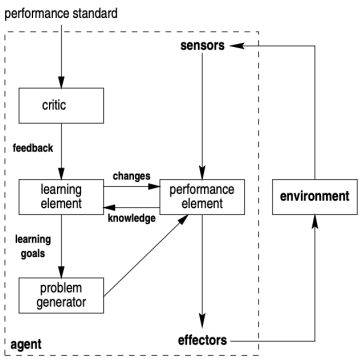
7. **Reflex Agents** (*left*) follow condition-action rules, but assume there is a correct decision to make. **Model-based reflex** agents (*right*) have state that holds the knowledge about updates in the world, works if ∃ correct decision.



8. **Goal-based** agents (*left*): environment state too broad, need a goal and effects of actions to narrow the context down. But usually too many goals, need to prioritise using **utility function** (*right*), can measure importance of uncertain achievements against their likelihood.



9. **Learning Agents** are used when the information about the environment is incomplete. Percepts improve agent's ability to act in the future as they're fed into the **Performance** element (PE) to produce actions (for non-learning agents it's the whole agent), **learning** element takes knowledge of PE and feedback from a **critic** (performance measure/standard), and suggests improvements to the PE. Finally, **problem generator** suggests actions in pursuit of new informative experiences, without it PE would do what it thinks is best, settling for less exploration.



Uninformed Search

1. **Problem-solving agent** is a goal-based agent that determines sequences of actions that lead to desirable states. Need to formulate 1. **Goal** given in the curr. situation, 2. **Problem** comprising permissible actions (operators) and states to consider, then 3. **Search** the sequence of actions and 4. **Execute** them. Can be online or offline.

2. It's *flat*, represented with *states*, is in *indefinite stage*, *fully observable*, *deterministic*, prefers *goals*, is *single-agent*, *knowledge is given*, has *perfect rationality* and runs *offline*.

Offline Problem-Solving Agent

```
def problem_solving_agent(p) -> action:
  p, s, g = inp(), {}, null #percept/action_seq/goal
  state ← Update-State(state, p)
  if not s:
    g ← Formulate-Goal(state)
    problem ← Formulate-Problem(state,p)
    s ← Search(problem)
  action ← Recommendation(s,state)
  s ← Remainder(s,state)
  return action
```

3. **Single-state problem** is *deterministic* and *fully observable*, **Multiple-state** problem is *deterministic* but *partially observable*, agent manipulates sets of possible states. **Contingency problem** is *stochastic* and *partially observable*: don't know curr state or result of action, so use sensors and solution is a **tree** of contingency branches, often **interleave** search and execution. **Exploration** problem is online, so no action plan (travelling without a map).

4. **State-space** problems comprise a set of states, *start states*, set of actions, **action function** producing new states given state curr state and action, a set of goal states (`goal(s)`) and a **criterion** that specifies the quality of an acceptable solution. Can get complex, so may **abstract**.

5. **State-space digraph** $G = (N, A)$ (nodes and arcs). Node n_2 is a *neighbour* of n_1 if $\langle n_1, n_2 \rangle \in A$. Given a set of start and goal nodes, a solution is a path $\langle n_{\text{start}}, .., n_{\text{goal}} \rangle$.

6. **Uninformed tree search** is usually **DFS** or **BFS**, where queue is called **frontier**. Need to analyse completeness, optimality, time and space complexity. To account for loops, keep a frontier of explored paths from the start node. The way it's expanded defines the search strategy.

7. **Graph Search Algorithm**

```
graph, start_nodes, isGoal(n)::bool, frontier={⟨s⟩}
while frontier: # unvisited if graph, all if tree
  select and remove path ⟨n_0, .., n_k⟩ from frontier
  if goal(n_k): return ⟨n_0, .., n_k⟩
  for n in n_k.neighbours: frontier.add(⟨n_0, .., n_k, n⟩)
```

8. **Lowest-cost-first (LCF)** graph search selects the path with the lowest cumulative cost$(\langle n_0, \ldots, n_k\rangle) = \sum_{i=1}^{k} \text{cost}(n_{i-1}, n_i)$ from the frontier. If costs are equal, nodes are expanded in BFS (alphabetic) order.

Informed Search

1. Improve offline search efficiency and reliability using problem specific knowledge. Uninformed usually inefficient.

2. **Heuristic Search**: use an efficient heuristic function `h(n)` (can be extended to paths), or the estimate of the optimal solution to guide the search. `h(n)` is an **underestimate** if $\nexists$ path from n to a goal with cost $<$ `h(n)`. An **admissible heuristic** is a nonnegative `h(n)` that never overestimates the cost to reach the goal, (so is $\leq$ goal).

3. **Consistent heuristic**: is one satisfying **monotone restriction** $h(n) \leq \text{cost}(n, n') + h(n')$ for any arc $\langle n, n'\rangle$ and ensures that first path found to a node is lowest-cost one.

4. **Best-first Search** treats frontier as a PQ ordered by heuristic $h(n)$. In DFS, it would select the node appearing closes to the goal, in **Greedy best-first search** - the path p with lowest $h(p)$. For branching factor b, path length n, *Time*: O(b^n), *Space*: O(b^n), not complete nor optimal.

5. **A* Search** uses path cost `cost(p)` and heuristics `h(p)`, resulting in $f(p) = \text{cost}(p) + h(p)$. Frontier is PQ ordered by $f(p)$. Unlike Best-FS, considers cost from the very start. A^* is **admissible** (if a solution exists, it returns an optimal one) if b finite, $\forall e \in E : \text{cost}(e) > 0$ and $h(n)$ is admissible. *Time*: exp(relative error in $h \times \text{len}(p_{\text{opt}})$), *Space*: exponential, as it keeps all nodes in memory.

Strategy	Frontier Selection	Complete	Space
Breadth-first	First node added	Yes, fewest arcs	Exp
Depth-first	Last node added	No	Linear
Lowest-cost-first	Minimal $cost(p)$	Yes, least cost	Exp
Heuristic depth-first	Local min $h(p)$	No	Linear
Greedy best-first	Global min $h(p)$	No	Exp
A^*	Minimal $f(p)$	Yes, least cost	Exp

6. **Cycle pruning**: paths $\langle n_0, .., n_k, n\rangle$ where $n \in \langle n_0, .., n_k\rangle$ are not added to the frontier. Can check in O(1) with hashmap (DFS) or O(len(p)) for path p else.

7. **Multiple-Path pruning**: prune a path to node n when search has already found a path to n. Maintain a **closed list** at the end of explored paths: for path $\langle n_0, ..n_k\rangle$, if n_k in the closed list, the path is discarded. Not yet admissible
   - ensure this does not happen — make sure that the shortest path to a node is found first
   - remove all paths from the frontier that use the longer path, i.e., if there is a path $p = \langle s, \ldots, n, \ldots, m\rangle$ on the frontier, and a path p' to n is found with a lower cost than the portion of p from s to n then p can be removed from the frontier
   - change the initial segment of the paths on the frontier to use the shorter path, i.e., if there is a path $p = \langle s, \ldots, n, \ldots, m\rangle$ on the frontier, and a path p' to n is found with a lower cost than the portion of p from s to n then p' can replace the initial part of p to n.

8. **Direction of Search** is symmetric: given **forward** or **backward branching factor** or the number of arcs (out of)/into a node b_{out}, b_{in} and search complexity b^n, should use $\min(b_{out}, b_{in})$ (sometimes more efficient to search from goal to start). Backwards graph not available if dynamically constructed, but runs in O$(2b^{d/2})$ - exp faster.

9. **Bidirectional search**: forward and backward search simultaneously, but need to ensure they meet.

   Island Driven Search: find places where the two must meet (pass through) between start and goal s, g. May need problem-specific knowledge, and hard to guarantee optimality, but $mb^{k \div m} \ll b^k$, subproblem islands solved using hierarchy of abstractions: for table dist(n) from n to g:

$$dist(n) = \begin{cases} 0 & \text{if } is\_goal(n) \\ \min_{\langle n,m\rangle \in A}(|\langle n, m\rangle| + dist(m)) & \text{otherwise} \end{cases}$$

   This can be used locally to determine what to do, defining a **policy** of which arc to take from a given node. DP requires a lot of space, `dist()` is continuously recomputed.

10. **Bounded DFS** doesn't expand paths exceeding the bound. **Iterative-deepening search** performs Bounded DFS at each bound starting at 0 until the goal is found, mimics BFS, but uses linear space, although has asymptotic overhead of $\sum^{k}$`cost(k)`$\times b \div (b-1)$ at depth k.(??)

11. **Depth-first Branch-and-Bound** DFS with heuristic that finds optimal solution in linear space. Set bound to estimate of cost(p_{opt}), prune if finds path p s.t. cost(p) + $h(p) \geq$ bound. If non-pruned path p' reached g, set bound=cost(p'). Space nonlinear in Multi-path pruning. Can initialise heuristic to ∞ or maximum depth.

12. **Effective branching factor** b^* for A^* tree-search expanding N nodes and solution depth d, is the branching factor of a uniform tree of depth d containing N nodes: $N = 1 + (b^*)^1 + .. + (b^*)^d$. E.g. $d = 5, N = 52 \rightarrow b^* = 1.91$. Experimentally, heuristics with lower effective branching factors are better. h_2 **dominates** h_1 if $\forall n : h_2(n) \geq h_1(n)$.

13. Generally, A^* tree-search expands fewer nodes using **dominating heuristic** (all with $f(n) <$ (f^*=cost(p_{opt})), or all nodes with $h(n) < f^* - g(n)$ where $g(n)$=cost(p). So, non-dominant h_1 may expand more nodes, hence provided being admissible, is worse than dominant.

14. Can derive admissible heuristics from the **exact** solution cost of a **relaxed** (fewer restrictions on operators) version of the problem. Generate several admissible heuristics, and choose dominant $h(n) = \max(h_1(n), .., h_m(n))$. Can also set cost of **subproblem** = lower bound on cost of complete problem.

15. Can store exact solution costs for each possible simplified (sub)problem (heuristic h_{DB}) and construct **Pattern Database** by searching backwards from goal using dynamic programming, finally choosing dominant heuristic.

16. Can create **disjoint pattern DB** by dividing up the problem s.t. moves only affect a single subproblem, as otherwise different databases wouldn't be compatible.

17. Other approaches of deriving heuristics include **statistical** (run search over training problems and gather stats) that expands fewer nodes but *isn't admissible*, or select **features** of state that contribute to heuristic.

Constraint Satisfaction Problems

1. **CSP** is a set of variables $\{V_1, .., V_n\}$ each with domain $V_i \subseteq D_{V_i}$ and **hard constraints** (legal combinations) on its subsets. A solution is a variable value assignment satisfying all constraints. Can be optimisation problem with function of cost for each assignment, and the solution being an assignment minimising such cost.

2. **Finite** CSP domains have n variables of domain size d and produce $O(d^n)$ complete assignments (e.g. Boolean SAT). **Infinite** can't be enumerated, need **constraint language** e.g . $I_a > J_a$, solvable with linear constraints, undecidable with nonlinear ones (interval scheduling). **Discrete**↑. **Continuous** solvable in P-time by linear programming.

3. **Constraints** can be **unary**: single var ($X \neq 5$), **binary**: var pairs ($X \neq Y$), **high-order**: 3 or more vars, and **preferences/soft** - maximising the assignment's adherence to which decreases the cost in optimisation.

4. **Generate-and-Test algorithm** (brute force): generate **assignment space** $D = D_{V_1} \times .. \times D_{V_n}$ (set of total assignments), and test each one against the constraints. Prod of D of size d has d^n possible assignments - inefficient.

5. **Backtracking Search (BS) (uninformed)**

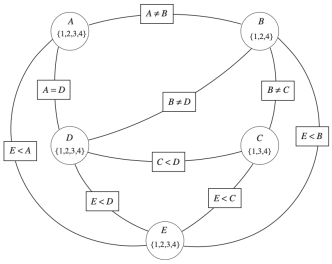
```
def BS(csp) → solution/failure:
  return Recursive-BS({}, csp)
def Recursive-BS(assign, csp) → sol./fail:
  if assign is complete: return assign
  var←Get-Unassigned-Var(vars[csp], assign, csp)
  for val in Order-Domain-Vals(var, assign, csp):
    if val adheres to assign given Constraints[csp]:
      add {var=val} to assign
      result ← Recursive-BS(assign, csp)
      if result ≠ failure: return result
      remove {var=val} from assign
  return failure
```

Backtracking algorithms: systematically explore D one variable at a time (assignments are commutative), evaluate immediately and prune if unsatisfying, every solution appears at depth n, so use DFS with branching factor $b = (n - l) \cdot d$ at depth l, so $n! \cdot d^n$ leaves.

6. What variable should be assigned next? **Minimum remaining values (MRV)**: choose the variable with fewest legal values (fail first). **Degree heuristics**: choose the variable with the most constraints on remaining variables. In what order should its values be tried? **Least constraining value (LCV)**: given a var, choose the value that rules out the fewest values in remaining vars.

7. **CSP as graph search**: suppose node N is assignment of values to variables: $X_1 = v_1, .., X_k = v_k$, then select Y not yet assigned in N, for each value $y_i \in \text{dom}(Y) : X_1 = v_1, .., X_k = v_k, Y = y_i$ is N's neighbour if consistent with constraints. Start with empty assign., goal is a satisfying total assignment.

8. **Constraint network**: circular node for variables (and their domain), rectangular node for each constraint, both connected by an arc $\langle X, c\rangle$ if constraint c involves X.



9. **Domain Consistency**: prune domains as much as possible before selecting values. If constraint c has scope $\{X\}$, then arc $\langle X, c\rangle$ is **domain consistent** if each value of X satisfies constraint c.

10. **Arc consistent** $\langle X, c\rangle$ for $c \in \{X, Y_1, .., Y_k\}$ if $\forall x \in D_X$: $\exists y_1, .., y_k$ where $y_i \in D_{Y_i}$ s.t. $c(X{=}x, Y_1{=}y_1, .., Y_k{=}y_k)$ is satisfied. Every value of X must have ≥ 1 way to pick values for Y_i to satisfy c.

    $\langle X, r(X, \overline{Y})\rangle$ is arc consistent if $\forall x \in D_X : \exists \overline{y} \in D_Y$ s.t. constraint $r(x, \overline{y})$ is satisfied. All vals in D_X with no corresponding val in $D_{\overline{Y}}$) are deleted to make it consistent.

11. **Domain splitting** (case analysis): split a domain, recursively solve each half. Check all **dependent** (not from scratch) arc consistencies at each domain reduction. There are n/c subproblems each taking $O(d^c)$ to solve, hence overall $O(d^c n/c \cdot)$ - linear in n.

12. **Theorem**: if constraint graph has no loops (tree), CSP can be solved in $O(n \cdot d^2)$, compared to general $O(d^n)$.

13. **Tree-Structured CSP algorithm**: Choose a variable as root, hierarchically order vars from root to leaves. For j in range$(n, 2, -1)$ remove inconsistent domain elements for $\langle \text{Parent}(X_j), X_j\rangle$ (reverse order ensures deleted values don't influence consistency of processed arcs). For j from 1 to n, assign X_j consistently with Parent(X_j).

14. **Nearly Tree-Structured** CSP's: initialise a variable, prune it's neighbours domains until it's a tree. **Cutset Conditioning**: init (in all ways) a set of vars s.t. the remaining constraint graph is a tree. Takes $O(d^c \cdot (n-c) \cdot d^2)$ for cutset size c, which is very fast for small c.

15. **Variable Elimination**: eliminate variables one by one, passing their constraints to their neighbours. **Algorithm**: if only 1 var, return intersection of the unary constraints that contain it, else select var X, join constraints in which X appears, forming constraint R_1, project it onto its variables except X, forming R_2. Replace all constraints in which X appears by R_2, recursively solve the simplified problem forming R_3, etc. Return R_1 joined with R_3. If the final single var has no values, the network was inconsistent. Elimination follows **elimination ordering**.

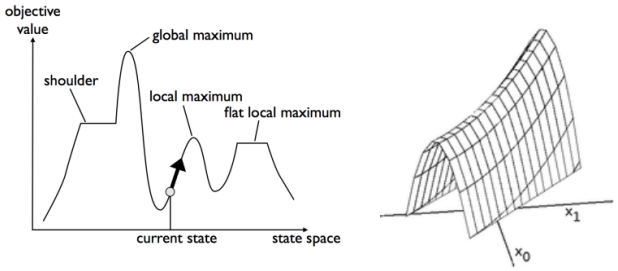
Local Search

1. **Iterative Improvement Algorithm**: don't keep track of paths (memory efficient, often constant space) - goal state is the solution, so iteratively check neighbours. Useful for optimisation problems: state space = set of complete configurations, goal is some subset (min)maximising some objective function (e.g. Hamiltonian path).

2. **Hill-climbing (greedy local search)**

```
def Hill-Climbing(problem) → solution state:
  curr ← Make-Node(init-state[problem])
  loop do:
    next ← highest-valued successor of curr
    if Value[next] < Value[curr]: return curr
    curr ← next
```

3. Hill-climbing problems: local maxima can be suboptimal, ridges may lead to oscillation and no progress.



Plateaux are flat, so search is random (**shoulder** if progression possible), limit sideways moves, else can get stuck.

4. **Greedy descent** (local search for CSP): find an assignment with zero **conflicts** (unsatisfied constraints). Heuristic to be minimised is the number of conflicts.

5. **Random walk**: sometimes choose random variable-value pair or participating in most/any/none conflicts.

6. **Small/unordered domains** → neighbour assignment corresponds to choosing another value for one of the vars. **Large/ordered dom.**: neighbours adjacent ("ab", "a"). **Continuous domains**: gradient descent changes each variable according to gradient of heuristic function in that direction: X_i goes from v_i to $v_i - \eta \frac{\delta h}{\delta X_i}$ with step size η.

7. **Randomised Greedy Descent** allows **random steps** (walk): move to random neighbour, **random restart**: reassign random values to all variables k times **k-restarts**. Subset of **Stochastic (random) local search**.

8. Measure stochastic algorithm performance: plot runtime and proportion of runs solved within that runtime, measure how much time is available or how important is to find solution to decide which algorithm is the best.

9. **Simulated Annealing**: pick variable and value at random, if improvement, use it, else use it probabilistically based on temperature T: move from current assignment n to n' with probability $e^{-(h(n')-h(n))/T}$

10. **Tabu list** of the k last assignments prevents cycling: don't choose ones in it. Total assignment = **individual**.

11. **Parallel Search**: Maintain population of k individuals, at every stage update each one, report any solutions. Like k-restarts, but uses k times the minimum number of steps.

12. **Beam Search**: like parallel search, but choose the k best of all neighbours. When $k = 1$, it's greedy descent, when $k = \infty$ it's BFS. Value of k limits space and parallelism.

13. **Stochastic Beam Search**: choose the k next-generation individuals probabilistically, proportional to heuristics: usually **Boltzmann/Gibbs distribution** $e^{-h(n)/T}$.

14. **Genetic Algorithms (GA)**: related to stochastic beam search, but successor states obtained from *two* parents. Start with **population** of k randomly generated **individuals** (states), represented as a string over a finite alphabet.

15. Each individual is evaluated by **fitness function** (better state → higher value) used to find reproduction probabilities: $e^{-h(n)/T}$ or tournament selection (winner), according to which individual pairs are then chosen. Below-threshold values are **culled** (discarded). For each pair, choose random **crossover** (position in the string), generate offspring by crossing over parent strings. $\underline{12} + 34 \rightarrow \underline{1}4, 3\underline{2}$.

16. GA's take decreasingly large steps (simulated **annealing**) as population converges. Can also have random mutation with small probability . GA's advantage is the ability to crossover large blocks that evolved independently. Need to choose representation corresponding to meaningful components of solution. Useful for optimisation problems.

Adversarial Search

1. **Adversarial search**, or competitive multi-agent **games** where opponents introduce uncertainty (opponent trying to make the best move, randomness, or insufficient time to decide on exact consequences of actions), need to deal with **contingencies** when searching for the solution. Inefficiency is heavily penalised.

2. **Game** has *initial state*, set of operators (successor functions) deciding legal moves and resulting states, *terminal test*: check if game over (in *terminal state*), and *utility function*, or numeric value for terminal states (e.g. chess: {win, lose, draw} = {+1, -1, 0}. Can build **game tree**.

Perfect Decisions

1. Game with 2 (Min, Max) players. Max moves first, needs to form a **strategy** (correct Max's move for each possible Min's move) to win anything Min tries to do. A **Ply** is a single move by one of the player.

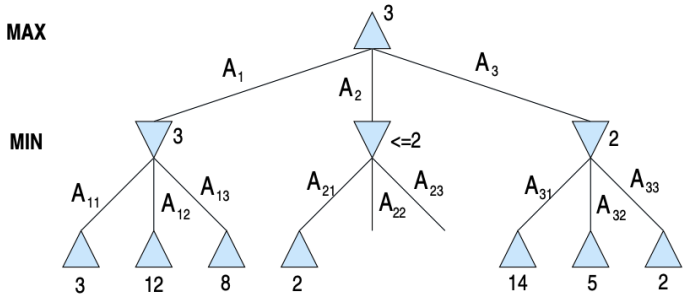
2. **Minimax** gives optimal strategy for Max. **Minimax value** of a state is Max's utility of being in that state given both players play optimally. MinimaxValue(n) =

$$\begin{cases} \text{Utility}(n) & \text{if } n \text{ is a terminal} \\ \alpha = \max_{s \in \text{Successors}(n)} \text{MinimaxValue}(s) & \text{if } n \text{ is a Max node} \\ \beta = \min_{s \in \text{Successors}(n)} \text{MinimaxValue}(s) & \text{if } n \text{ is a Min node} \end{cases}$$

   Minimax obtains best achievable payoff against best play.

3. **Minimax algorithm**: 1. generate complete game tree, 2. rate terminal states with utility function, 3. iteratively use those utilities to find utilities of nodes one level up until the root. 4. Max chooses the highest utility move. *Complete*, *optimal*, Time: O(b^d) (slow), Space: O(bd) (DFS). Can extend to > 2 players, use *utility vector*.

4. **Alpha-Beta Pruning**: prune non-influential branches in a complete search tree. For candidate node n the player might move to, if $\exists$ a better choice m either at parent of n or above, then n won't be reached, prune it. α/β is the best choice for Max/Min.



5. More efficient to examine best successors first, taking O($b^{b/2}$) instead of O($b^{3b/4}$) if random.

Imperfect Decisions

1. Alpha-Beta still needs to search to the terminal states for some of the tree, better use **heuristic evaluation function** to get a value for states and a **cutoff test** to decide when to stop going down the tree.

2. **Evaluation function** gives an estimate of expected utility for a given position, should order terminal states per utility function. Most calculate **features** of a state, defining their equivalence classes. Can combine features (f) using weighted (w) linear function $w_1 f_1 + .. + w_n f_n$. If features are not independent, use a nonlinear function.

3. Simple cutoff tests: checking if fixed depth is reached, or **Iterative deepening** (continue until out of time). Both are unreliable (bc of approximation in eval func).

4. Hence, use **Quiescent Search**: only apply eval func to **quiescent** (those not changing soon) positions, meanwhile, expand nonquiescent positions - restricting to certain types of moves to quickly resolve pos. uncertainties.

5. **Horizon problem**: when have unavoidable damaging move from opponent, a *fixed-depth* search is fooled into viewing stalling moves as avoidance. Can fix using:

6. **Singular extension search**: choosing a move clearly better than all others. **Forward pruning**: immediate pruning of moves from discarded future nodes, but only safe in very deep search trees; if two symmetric or equiv moves - consider one of them.

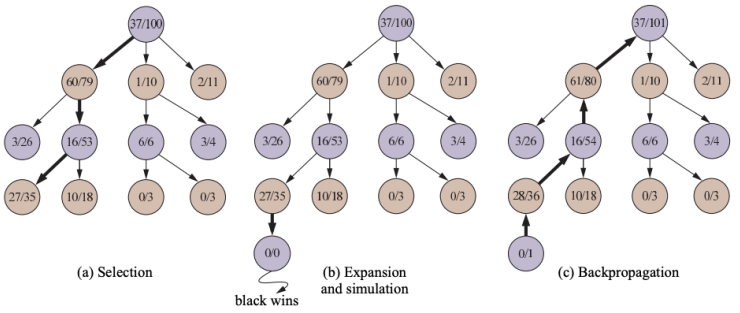
7. In **games with chance**, can't construct complete game tree, so include **chance nodes**, labelled with *result* and *probability*. Their **expected value**, Exp-minimax(n) =

$$\begin{cases} \text{Utility}(n) & \text{if } n \text{ is terminal node} \\ \max_{s \in \text{Successors}(n)} \text{Exp-minimax}(s) & \text{if } n \text{ is a Max node} \\ \min_{s \in \text{Successors}(n)} \text{Exp-minimax}(s) & \text{if } n \text{ is a Min node} \\ \sum_{s \in \text{Successors}(n)} P(s) \cdot \text{Exp-minimax}(s) & \text{if } n \text{ is a chance node} \end{cases}$$

8. Exp-minimax considers all n distinct outcomes of each chance node, so O($b^m n^m$). Can prune chance node without considering its children if put bounds on utility function (chance node is an average, within known bounds).

9. **Monte Carlo Tree Search (MCTS)** addresses branching factor and difficulty of defining eval function in alpha-beta tree search. Estimate value of a state from average utility over **playouts** (simulations) of complete games starting from the state. A **playout policy** determines which moves to make during a playout: learn from self-play or game-specific heuristics (e.g. chess capture moves). Pure MCTS: do N simulations. Use of UCT balances **exploration** of states with few playouts and **exploitation** of those that have done well.

10. MCTS maintains and grows a search tree on each iteration using: **selection**: start at root, choose a move using selection policy (UCT); **expansion**: generate a new child of selected node; **simulation**: playout from newly generated child node (determine outcome, don't record these moves), and **back-propagation**: use result of playout to update search tree up to the root. Repeat for fixed num of iters or until out of time, return move with highest num of playouts (2/3 worse than 65/100 because of uncertainty).



11. **Upper Confidence Bounds (UCT)** is an effective selection policy using upper confidence bound formula:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(Parent(n))}{N(n)}}$$

where $U(n)/N(n)$ is total utility/number of playouts through n, $Parent(n)$ is n's parent in the tree, C is constant balancing exploitation and exploration. Playout computed in O(height) as only one move at choice point.

Planning with Certainty

1. **Knowledge Base (KB)** is database of facts/beliefs, or a set of sentences in formal knowledge representation language (list, array, db etc.). **Inference engine** is a mechanism for reasoning about such beliefs.

2. KB allows building agents **declaratively**: **Tell** (*observe & remember*) it what to do (add sentence). Agent can **Ask** (*decide*), or query itself what to do, and answers follow from KB through inference, following previous *Tells*.

3. **Knowledge-based KB-agents** can reason with **inference** and **knowledge**, can accept new **goals**, update knowledge to adapt to environmental change and **infer** unseen properties of the world from perceptions, so are a better and more flexible solution for partially observable/-dynamic environments than simple search.

4. KB-agents characterised by 3 levels: **knowledge** (epistemological): what is known, regardless of implementation; **logical**: knowledge encoded in formal sentences; **implementation**: data structures/algorithms in KB.

5. KB may contain initial background knowledge. Logical level hidden in Tell, Ask - both are internal to KB.

Simple knowledge-based agent

```
def KB-Agent(percept) → action:
  t ← 0 # time counter
  Tell(KB, Make-Percept-Sentence(percept,t))
  action ← Ask(KB, Make-Action-Tell(KB,
      Make-Action-Sentence(action,t))
  t ← t+1
  return action # logically reasoned action
```

6. **Coercion**: when no certainly safe moves available, reduce uncertainty in the world by forcing (risk) the agent into a known state.

Logic

1. Logics are formal languages for representing info s.t conclusions can be drawn. **Syntax** defines sentences, and **Semantics** - their meanings, or a mapping of sentences to facts (i.e. define truth). Facts *follow* facts.

2. KB **entails** sentence α, or $KB \models \alpha$ iff α is true in all worlds where KB is true. Inference producedure generating only entailed sentences is **sound** or **truth preserving**. Semantics help extract **proof theory** of language (what reasoning steps are sound).

3. **Inference** $KB \vdash_i \alpha$ means sentence α can be derived from KB by procedure i. Procedure i is **Sound** when $KB \vdash_i \alpha \rightarrow KB \models \alpha$, **Complete** if $KB \models \alpha \rightarrow KB \vdash_i \alpha$.

Searching vs Planning

1. Using simple search on real-world problems is infeasible - too many options (unconstrained branching) and hard to apply heuristics. So, use **planning systems**: represent states and goals as sets of sentences, and actions as descriptions of preconditions and effects to allow for direct connections between states and actions.

2. **Divide-and-conquer** by **subgoaling**: planner considering easier problems, then combining solutions. Little interaction between subplans, else cost of combining is too high (useless for 8-puzzle to consider each tile separately).

3. Planning relaxes requirement for sequential construction of solutions, so can add obvious or important actions where needed to reduce branching factor. Order of planning and execution are independent. Planning does better than search in the real-world scenarios.

	Search	Planning
States	data structures	logical sentences
Actions	code	preconditions and outcomes
Goal	code	logical sentence (conjunction)
Plan	sequence from S_0	constraints on actions

4.

5. Update knowledge base, if not already executing a plan, generate a goal and a plan to achieve it. If goal infeasible or achieved, set action to NoOp, once the plan is ready, execute to completion. Minimal interaction w environment.

Simple Planning Agent

```
def Simple-Planning-Agent(percept) → action:
  p, t ← NoPlan, 0 # plan, time counter
  local G, current; # goal, curr state description
  Tell(KB, Make-Percept-Sentence(percept,t))
  current ← State-Description(KB,t) # where am I
  if p == NoPlan: # no plan yet
    G ← Ask(KB, Make-Goal-Query(t)) # what goal
    p ← Ideal-Planner(current,G,KB) # plan
  if p == NoPlan or p empty: action ← NoOp # nothing
  else: action, p ← First(p), Rest(p)
  Tell(KB,Make-Action-Sentence(action,t)) # step
  t ← t + 1; return action
```

Situation calculus and Strips

1. **Situation calculus** is a way of describing change in first-order logic. The world is a sequence of **situations** (snapshots of state of the world), generated from previous situations by actions, or Result(action,situation).

2. **Fluents** (functions, predicates) can change with time given a situation argument At(Agent, S_0 etc). **Eternal or Atemporal** ones don't change, so they don't take situation arguments, e.g. Wall$(0,1)$ - it's always there.

3. **Possibility axioms** of form Precondition $\Rightarrow$ Poss(a, s) describe when it's possible to execute an action. Resulting action defines **effect axioms** of form Poss$(a, s) \Rightarrow$ changes (i.e. executing the action if condition is met). E.g. *Poss(Go(x,y), s) ⇒ At(Agent, y, Result(Go(x,y), s))*.

4. **Situation calculus** turns planning into a logical inference problem with initial state: sentence about S_0: At(Home, S_0)$\wedge\neg$Have(Milk, S_0) and goal state: logical query for suitable situations.: $\exists s$: At(Home, s)$\wedge$Have(Milk, s).

5. Nothing in KB says if something doesn't change: **Frame axioms** capture non-changes due to an action. F fluents and A actions require O(AF) such axioms. **Frame problem** is solved if *representational* (frame axiom proliferation), *inferential* is harder, but avoided by planning.

6. **Successor-state axioms** solve representational frame problem. Each axiom is about a **predicate** P s.t. after an action, P is true and without it it's false. But need to list all such combinations, so use **restricted language**.

7. **Strips**, or Stanford Research Institute Problem Solver is a planner where: states are conjunctions of predicates applied to constants. Goals are conjunctions of literals. Planner asks for a sequence of actions leading to goals.

8. **Closed-world assumption**: planners assume that state descriptions that don't mention a positive literal (non-negated) are false. This assumption can be unreliable.

9. **Operators** comprise *Action*, *Precondition* (conjunction of positive literals) referring to that action, and *Effect* (conj. of free literals). Free literals are those not quantified.

10. **Operator schema** is an operator with variables. Operator is **applicable** in state s if can instantiate each var satisfying precondition in s.

11. In standard search, nodes are concrete world *states*, in planning search - *partial plans*. **Open condition** (OC) is a not yet fulfilled precondition. *Causal link* ($S_i \xrightarrow{c} S_j$), e.g. S_i achieves c for S_j, and *order* ($\prec$) steps with respect to one another, moving to **fully instantiated** plans.

12. Principle of **least commitment**: leave choices as long as possible. Precond **achieved** iff it's the effect of earlier step and no possibly intervening step undoes it. Plan is **complete** iff every precond is achieved, **consistent** iff no contradictions in ordering or binding constraints.

13. Planning: **Progression**: move forward start $\rightarrow$ goal, have high branching factor and search space. **Regression**: move backward start $\leftarrow$ goal , has lower branching factor, but complex conjunctions. **Partial plan** is an incomplete one, some steps not instantiated. **Partial/Total order**: some/all steps ordered with respect to others.

14. **Partial-Order Planner (POP)** is a regression planner to search through plan space. Each iter add a step fulfilling non-achieved preconds, establishing **protected** links (those not breaking others), backtrack if inconsistent. POP is *sound*, *complete*, *systematic* (no repetition).

---

1) Pick step S_{need} with open condition c.
2) Choose step S_{chosen} that fulfils c from operators or plan.
3) Add causal link $S_{\text{chosen}} \xrightarrow{c} S_{\text{need}}$ and $S_{\text{chosen}} \prec S_{\text{need}}$ to ordering. If S_{chosen} is new, add Start $\prec S_{\text{chosen}} \prec$ Finish.
4) If thread/cloberrer to causal link, promote/demote.

15.

Op(Start)	$\rightarrow$ **Op(Drive)** $\rightarrow$	**Op(End)**
Effect: at(Home)	Precond: at(Home)	Precond: at(Office)
Capabilities: —	Effect: at(Office)	Goal: delivered

16. **Clobberer** is a potentially intervening step that destroys (**protected**) causal link condition. Protect them by ensuring threats (clobbering steps) are demoted/promoted, or ordered before/after the causal link.

17. **Sussman anomaly**: Early planners are focused on a single conjunct at a time, so can't guarantee minimal number of operators.

Expanding POP and Hierarchical Decomposition

1. **Problems** with Strips: can't express hierarchical plans, complex conditions, time and resources, hence extend it with **Hierarchical Decomposition** (HD).

2. HD includes nonprimitive and **abstract** operators that can be decomposed into steps that implement them; they are predetermined and stored in library of plans.

3. Plan p **correctly implements** nonprimitive op o if it's complete and consistent plan for achieving effects of o given precond of o. This guarantees nonprimitive operator can be replaced by its decomposition in the plan.

4. **Hierarchical Decomposition HD-POP**

```
def HD-POP(plan,ops,methods) → plan:
  while not Solution(plan):
    S_need, c ← Select-Subgoal(plan)
    Choose-Operator(plan,ops,S_need,c)
    S_nonprim ← Select-Nonprimitive(plan)
    Choose-Decomposition(plan,methods,S_nonprim)
    Resolve-Threats(plan)
  return plan
```

   However, must check that all operators are primitive.

5. Operator decomposition: add *steps/bindings* of method to plan, remove S_{nonprim}/backtrack at contradiction. Follow least commitment ordering: $S_a \prec S_{\text{nonprim}}$ becomes S_a before latest steps of the method, $S_{\text{nonprim}} \prec S_z$ becomes S_z after the *earliest* steps. Resolve any threats. Finally, replace links to nonprim step with those to steps that achieve the precondition.

6. Broaden op descript. **Conditional effects** (CE) of form
   $$\text{Effect} : .. \wedge \neg\text{Clear}(y) \text{ when } y \neq \text{Table}$$
   avoids premature commitment (**effect with condition**). Select-Subgoal's planner needs to consider precond of CE if effect supplies a protected causal link. In Resolve-Threats, any step with effect $[\neg c \text{ when } p]$ poss. threatens link $S_i \xrightarrow{c} S_j$, fix by **confrontation**: ensure p's not true.

7. Allow **negated goals**, consider $\neg p$ matched by initial state not containing p. **Disjunctive preconditions** allow Select-Subgoal to make a nonderterministic choice between disjuncts using principle of least commitment, hence **disjunctive effects** may address coercion.

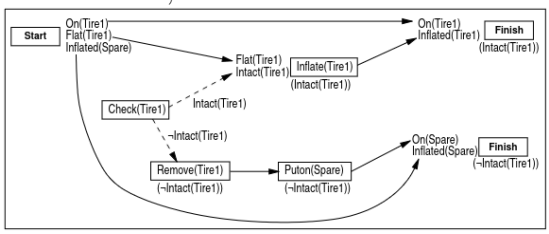
8. Can introduce **universally quantified preconditions** $\forall x \cdot X(x) \Rightarrow Y(x,..)$ and effects. Now have restricted first-order logic, so the world is finite, static and initial state must give **all** objects a type T. Preconditions and effects now have the form $\forall x \cdot T(x) \Rightarrow C(x)$ for condition C. Since the world is finite, static and typed, can expand universal quantification into conjunction.

9. To deal with consumable resources, have numeric **measures** expressed as **measure fluents** e.g. FuelLevel.

10. Plan for scarce resource first, delaying causal links where possible. Pick a plan step, do rough check on resource requirements, resolve threats, hence trying all possible operators. Time is also a resource, but 1. parallel operators cost the maximum, not the sum, 2. it never goes back.

11. Real world deals with incomplete (unknown precond, disjunctive effects), incorrect information and can't list all precond, outcomes of actions (qualificaiotn problem.

12. Fix using **conditional planning** with sensing actions to obtain information, but may create plans for unlikely cases, better use **Monitoring/Replanning**: check progress during execution, replan if necessary, but may fail. The key difference is that *steps have a context*, to check if executable, must insert actions to find info.



13. Sensing action may have any number of outcomes (checking tire for puncture makes it wet), use it in **parameterised plans** where exact actions not known until runtime. Runtime variables are unknown until sensed. To preserve certain facts, use a **maintenance goal** (start$\xrightarrow{c}$goal), can use in loops: while c:.. .

14. **Plan/Action monitoring**: failure means preconditions of the (remaining plan)/(next action) are not met. Replan in both cases. Precond of remaining plan are causal links starting at or before curr step and ending at or after it.

15. **Bounded/Unbounded indeterminacy**: unexpected effects of actions can/can't be enumerated,

16. **Replanning** from scratch is inefficient, rather try to get "back on track" to the original plan, generating "loop until done", introduce **learning** to avoid getting stuck (may keep trying opening a door, but might be locked)

17. Alternatives to replanning: 1. **Coercion**: performing an action to force world into a particular state to reduce uncertainty. 2. **Abstraction**: ignore potentially unknown details of the problem. 3. **Aggregation**: treat a large number of objects as one aggregate predictable object.

18. **Reactive Planning**: abandon domain-independent planning and use domain-specific **procedural knowledge**, or a library of partial plans representing behaviour collection. Select plan from library when agent needs it.

Knowledge Representation

1. Since *thinking* is retrieval of and reasoning upon relevant knowledge, need to represent it somehow.

2. Knowledge is a relation defined by **propositional attitude** between **knower** and a **proposition** (John knows Earth is round), but infeasible to represent all true propositions, so use reasoning to bridge this gap. Reasoning at each step + collection of known true propositions are **structural ingredients**.

3. Represent knowledge as logical formulae, hence a proposition (true/false statement) might be: $a \wedge b \wedge c$. Knowledge takes form of **definite clause** $h \leftarrow b$, with *body* b (proposition) and *head* h (literal determined by result of body).

4. The user is responsible for the **indended interpretation** of the symbols and responses - the system doesn't have access to it (all it sees is symbols), so user must **axiomatize** the domain (provide known true clauses).

5. **Expert system**=knowledgee+inference represents & reasons with knowledge of "specialist" interface to get advice and solve problems. Consultation is a goal-tree search.

6. Have vocabulary of set O of names of objects in domain, A of their attributes and V of values those attributes can take. Obj-Attr-Val triples ($o \in O, a \in A, v \in V$).

7. **Working memory WM** stores facts (assertions/propositions) defining initial state of KB and state transition operators. Facts are working memory elements (WME).

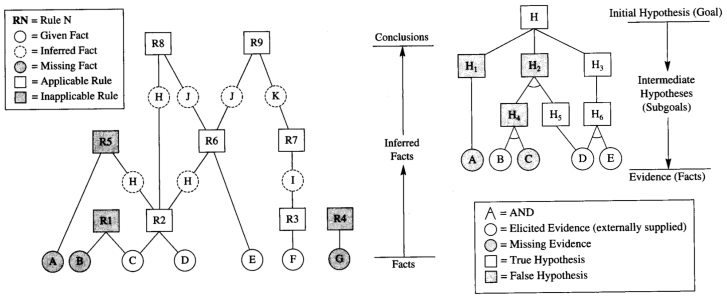
8. **Production rules** have sets of "if P_1 and .. and P_m" conditions and consequent "then Q_1 and ... and Q_n actions.

9. **Recognise-act cycle**: match "if" conditions of rules against elements in working memory, apply the rule (arriving at a solution or adding a new fact) and (optionally) delete from working memory, repeat until no rules or halt.

10. Global/Local inference control is domain (in)/dependent, (hard)/coded in IE/as meta rules. The series of rules that fire is called an **inference chain**.

11. **Forward chaining** (Bottom-up Ground Proof Procedure): data driven, starts from known data, all possible inferences will be made, so sound, deterministic, but performs wasteful computation. *left fig.*

12. **Backward chaining** (Top-down Definite Clause Proof Procedure): goal driven, tries to find evidence to prove the goal, more efficient as unrelated facts won't be inferred. But, non-deterministic, based on choice of production rules, can halt if some elements can't be derived (because query is conjunctive). *right fig.*



13. **Ask-the-user** system has askable clauses, the only way to receive new information is from the expert, but only works well with backward chaining (only ask when necessary), as too many things to ask about otherwise. User and system now have symmetric relationship.

14. Knowledge-level debugging can fix **non-syntactic** errors: ***incorrect answer*** (either some variable or the rule itself is wrong, ask the user), ***non-produced answer*** (no appropriate rule for the var, rule should've fired, so some var isn't true. Solve recursively by finding all true vars not in a rule), ***infinite loop*** (can't happen in forward chaining if refractoriness is applied; convert to digraph and check for cycles, reassess the rules) and ***irrelevant questions*** (needs reassessment of KB).

15. **Conflict resolution**: choosing which rule out of **conflict set**/agenda to fire. Can fire in order of appearance, but rules have strong influence on outcome, so don't guess, use priority, but difficult to define it. So, use **specificity**: fire most specific rule. **Recency**: fire rule that uses data most recently entered the WM, **refractoriness**: rule only allowed to fire once on the same data, preventing inference loops. Domain specific/independent **meta knowledge** is about knowledge, so use that too "if two rules are..then..".

16. WM only modified very slightly in each recognize-act cycle, many rules share conditions, so create offline network from rule antecedents with simple self-contained (*unary*) α nodes (α : age > 27) and variable condition constraints as β (*higher-order*) nodes (β : father $\sim$ name).

17. **RETE algorithm**: during operation of production system pass new/changed WME tokens through network. Those that make it through satisfy the rule, new conflict sets are generated from those in previous cycles, so only small part of WM needs to be matched. If can't move through network, reassess it when WME is modified

18. Rule-based systems provide easy mapping between expert and format of rules and ability to represent and reason with a lot of potentially uncertain knowledge, but may be expensive, have to be supervised and may be brittle.

19. **Assumption-based reasoning** contains **Abduction**: make assumptions to explain observations and **Default reasoning**: assume normality, used to find consequences.

    ABR has a set of closed formula F: **facts**, integrity constraints evaluating to false and H: **possible hypotheses** (assumables). Need to find set of assumables implying the given query. For each var in query select rule resulting in that var being added to WM, replace such vars with condition of the corresponding rule, repeat until all query vars are assumables.

Planning with Uncertainty (Bayesian AI)

Probability

1. Use Probability instead of logic. Set of outcomes must be well defined. *Frequentistic* view: repeatable identical experiments (coin toss), *objectivist*: tendencies of objects to behave in certain ways and *subjectivist*: deg. of belief.

2. **Sample space** Ω is a finite set of mutually exlusive and exhaustive possible outcomes $s_1, .., s_n$ (world states).

3. *Probability measure* (Ω, P) comes from the assignment of $P(s) \in [0,1]$ to each state $s_i \in \Omega$ s.t. $\sum_{s_i \in \Omega} P(s_i) = 1$. **Event** $(E \subseteq \Omega)$ can be impossible: $P(\Phi) = 0$, certain: $P(\Omega) = 1$, or otherwise $P(E) = \sum_{s_j \in E} P(s_j)$.

4. Given prob space (Ω, P), **random variable** X is a function on Ω. **Primitive** proposition: var assignment or comparison. **Proposition** combines primitive propositions with logical connectives (Heads $\wedge$ $X \neq Y$). Probability of proposition α in relation to each possible state s_i is: $P(\alpha) = \sum_{s_j \in \Omega : \alpha = \text{True in } s_i} P(s_i)$.

5. **Expected value** $E[X] = \sum_{s_i \in \Omega} P(s_i) X(s_i)$ is average value of random var. **Variance** $Var[X] = E[(X - E[X])^2]$ measures the spread of the random variable.

6. Random Var X can be discrete, continuos or binary. **Probability distribution (PD)** is function over X that assigns a probability to each possible outcome in domain: $P(\text{heads}) = P(\text{tails}) = 0.5$. If multiple vars then **joint**.

7. **Conditional probability** $P(\alpha|\beta) = P(\alpha \wedge \beta) \div P(\beta)$. Aka **posterior** prob. and $P(\alpha) \equiv P(\alpha|\text{true})$ is **prior** prob. Conditional prob is not a measure of causality.

8. Rand vars X, Y are **independent** if $P(X|Y) = P(X) \wedge P(Y|X) = P(Y)$, then $P(X \wedge Y) = P(X) \times P(Y)$. **Conditionally** indep given Z if $P(X|Y \wedge Z) = P(X|Z)$.

9. **Total probability**: given set of disjoint events A_i that partition sample space Ω, then $P(\Omega) = \sum_i P(A_i) = 1$.

    Total prob. $P(B) = \sum_i P(B \wedge A_i) = \sum_i P(B|A_i)P(A_i)$. **Product rule**: $P(A \wedge B) = P(B|A) \times P(A)$. **Chain rule**: $P(a_1 \wedge ..a_i) = P(a_1) \times P(a_2|a_1) .. \times P(a_i|a_1 \wedge ... \wedge a_{i-1})$.

    Bayes' Rule: $p(A|B) = \dfrac{p(B|A)p(A)}{p(B)} = \dfrac{p(B|A)p(A)}{\sum_{i=1}^{n} p(B|A_i)p(A_i)}$

    $$p(B) = p(B|A)p(A) + p(B|\neg A)p(\neg A)$$

10. Given rand vars A, B with mutually exclusive states $a_1, ..a_n$, $b_1, ..b_m$ and prob distrs $P(A), P(B)$, join PDs $P(A, B)$ will contain m events where A in state a_i, so $p(a_i) = \sum_{j=1}^{m} p(a_i, b_j)$. B is **marginalized** out of $p(A, B)$.

11. Prob density function **PDF** $P : \mathbb{R} \to \mathbb{R}^+$ integrates to 1. $\underline{P(a \leq X \leq b)} = \int_a^b p(X)dX$. $\underline{p(X{=}x)} = \frac{1}{\sigma\sqrt{2\pi}} e^{(-x-\mu)^2/(2\sigma^2)}$

Bayesian AI

1. **Probabilistic inference** procedure with query variable X, evidence variable E and their observed values e, unobserved variables Y, then to reason with uncertainty:

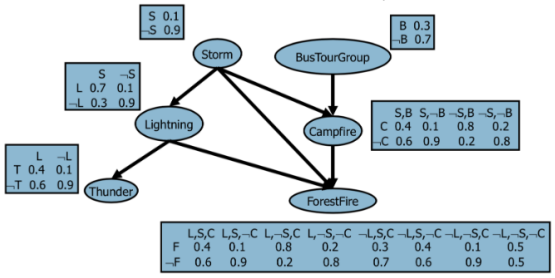
    General inference $p(X|e) = \alpha \cdot p(X, e) = \alpha \sum_y p(X, e, y)$

    Normalisation constant $\alpha = 1 \div [p(e) = \sum_{x,y} p(x, e, y)]$

    Full join probability distribution (PD) is the KB.

2. Vars related through **inference chains**. For PD P of random vars $X \in V$ in DAG $G{=}\langle V, E\rangle$, then (G, P) satisfies: **Markov Condition**: nodes are conditionally independent on the set of their non-descendents given their parent. Like chain rule, but: $P(a_1 \wedge ..a_i) = P(a_1) \times P(a_2|a_1) \times P(a_3|a_2) \times .. \times P(a_i|a_{i-1})$

3. **Bayesian Belief Network (BBN)** is a (G, P) satisfying Markov cond, $p(X_1, ..X_n) = \prod_{i=1}^{n} p(X_i|PA_{X_i})$, satisfies it too. Nodes are vars, influencing others down the tree. Using chain rule: $p(X_1, .., X_n) = \prod_{i=1}^{n} P(X_i|\text{Parents}(X_i))$

4. Constructing BBN: by **clarity principle**, omniscient agent must know a variable's value. Define its topology, including nodes (observable vars) and edges (qualitative relationships between them), and define **Conditional Probability Tables (CPTs)** $\{X_i : P(X_i|PA_x)\}$.

5. Each node has $\leq k = \max_i |\text{parents}(n_i)|$ parents, max num entries in CPT is 2^k, max num entries in PD is $2^k n$.

6. **Pearl's Network**: given a node ordering $\{X_1, .., X_n\}$, process them in order: add to network, add arcs from min set of parents s.t. X_i independent from any $X_{j<i}$ and define PA_{X_i}, finally define CPT for X_i. Compact networks are more tractable dense ones don't represent independencies or causal dependencies.

7. Markov condition entails only independencies, not dependencies! Presence of an edge doesn't imply there is a direct dependency, but its absence does imply the opposite.

8. Types of evidence E: *specific* ($E = e_1$), *negative*: ($E \neq e_1$) or *virtual*, or likelihood (def new PD over E).

9. **Probabilistic inference** is either **exact**, where probs are computed exactly (e.g. enumeration, variable elimination), or **approximate**, which produces a range of probs. and eventually converges to the answer.

10. Types of reasoning: ***diagnostic*** (symptom → cause), ***predictive*** (cause → symptom), ***intercausal*** (mutual causes of a common effect; they are independent unless effect is observed) and ***combined*** (query variable is parent and descendent of some other observed vars).

11. Inference by **Enumeration** through every world consistent with the evidence. From 1: y is the set of hidden ($\notin$ evidence,query) vars. E.g. $A, B \in y$, then consider $p(X|E \wedge (A \wedge B)/(A \wedge \neg B)/(\neg A \wedge B)/(\neg A \wedge \neg B))$.

12. **Variable Elimination (VE)** is more efficient than enumeration. **Factor** is function on a set of vars, or its **scope**, then $p(X|Y, Z)$ is a factor f_0 with scope (X, Y, Z). Such factors can be expressed as *arrays* if (co)domain positive, finite; values ordered, unique (truth table but with probabilities).

13. **Conditioning** on observed (e.g. $P(X|Y, Z)$) vars, define a new factor (e.g. $P(X|Y, Z{=}t)$) with new scope being (X, Y), since Z is already known. Keep decreasing factor scope until single probability.

14. If two factors share a scope variable, **multiply** them to produce new factor with scope of the union of the two: $f_0(X, Y) \times f_1(Y, Z) = f_2(X, Y, Z)$, like left join in SQL.

15. Can eliminate (**sum out**) var (e.g. Y) by adding each possible val res $f_1(X, Z){=}f_0(X, Y{=}t, Z) + f_0(X, Y{=}f, Z)$

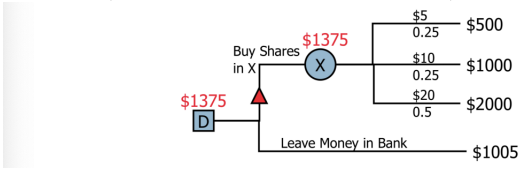
16. Algorithm for solving a BBN query: 1. construct a factor for each cond. prob condition, 2. eliminate each non-query var (if observed, set its val to observed val of each factor where it appears, else sum out), and 3. multipy remaining factors, normalise. Even if factors have same scope, their values are diff as they're constructed using diff evidence.

17. **Expected utility (EU)**: given set of outcomes $\{O_i\}$ of action A, evidence E and utility function $U(O_i|A)$ assigns measure of desirability to each O_i. Assuming PD $P(O_i|A)$:

$$\text{EU}(a) = \sum_i P(o|a) \times U(o), \text{ or } \text{EU}(A|E) = \sum_i P(O_i|A, E) \times U(O_i|A)$$

18. Consider outcomes σ_1, σ_2, then if $\sigma_1 (\succeq / \sim / \succ) \sigma_2$ then σ_1 is weakly preferred/indifferent/strictly preferred to σ_2. These relations are complete and transitive.

19. **Decision trees** ∘ have **chance nodes** (rand vars) with edges being their probability (expected val of util assoc with outcomes); and **decision nodes** □, representing decisions to make, edges being mutually exclusive exhaustive actions (max EU of all alternatives).



20. **Normative theory**: agent wants to maximise EU. Humans don't think that far ahead, so only consider value in current context (how much will I gain right now), called **prospect theory** - given risk tolerance R, have exponential **utility** function $U_R(x) = 1 - e^{-x/R}$.

21. For non-numerical outcomes, assign 0/1 to worst/best outcome, otherwise $p \cdot U(\neg X) + (1 - p) \cdot U(X)$.

22. Decision trees grow exponentially, probs sometimes unavailable, so use **influence diagrams (ID)**. Have observed/unobserved chance •/∘, decision □, utility ◇ nodes.

    •→∘: **probabilistic dependency** (chance child on parent)
    ∘→□: **information link** (decision observes chance)
    □→∘: probabilistic dependency (chance child on decision)
    □→□: **decision ordering** (parent ≺ child)
    •→◇, □→◇: deterministic utility dependency

    Chance nodes satisfy Markov condition. IDs are BBNs with utility and ordered decision nodes.

23. ID with single □: add all evidence, for each action value in □: set □ to that value, calculate posterior probs of nodes $\in PA_{\square}$, calculate and return single max EU of the actions.

24. **Information link** ∘ → □ indicates that chance must be known before a corresponding decision is made. Used to calculate what decision to make given ∘ vals, calculate same as (23) but instead of returning max, return an action → EU **decision table**.

25. Test-Action ID: eval test □ first, include the cost as separate ◇, if □ is test, collect evidence. To evaluate, evaluate Decision network with any avaliable evidence, enter test decision as evidence (if not "yes", use "unknown"), and eval action decision.

Reinforcement Learning

1. Have *prior knowledge*: possible world states or actions, *observations*: curr world state, immediate feedback and *goal*: act to max accumulated reward. At any time agent must either **explore**, gaining knowledge or **exploit** it.

2. RL Model comprises ***state transition*** $P(s'|a,s)$ (probability of getting to state s' if taken action a in state s) and ***reward*** $R(s,a,s')$ (expected reward of transitioning from s to s' using a) functions, then solve using **Markov Decision Process (MDP)**, or learn $Q^*(s,a)$ guiding action.

3. **Horizon**: agents carry out actions forever (infinite), until stopping criteria (indefinite), or in (finite) num steps.

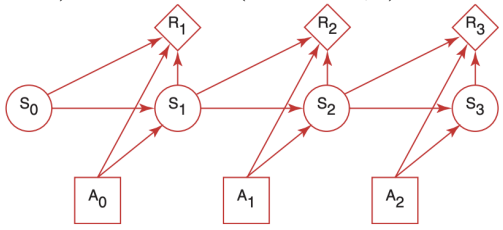
4. Assume flat, explicit states, indef/infinite stage, fully observable, stochastic, complex preferences, single agent, knowledge given, perfect rationality.

5. Given seq of rewards $r_1, r_2, ...$, can find total ($V{=}\sum_{i=1}^{\infty} r_i$) or average ($V{=}\lim_{n\to\infty}(r_1 + .. + r_n)/n$) rewards, but unreliable if seq is infinite, so use **discounted return** $V = r_1 + \gamma r_2 + \gamma^2 r_3 + ..$ with discount factor $0 \le \gamma \le 1$.

6. Since $V_t = r_t + \gamma V_{t+1}$ then $\frac{\text{min reward}}{1-\gamma} \le V_t \le \frac{\text{max reward}}{1-\gamma}$, this removes the motivation to go forever. Can approximate the value of V with: $V = \gamma^k V_{k+1} + (r_1 + ... + \gamma^{k-1} r_k)$.

7. **Markovian assumption**: no information about the past is relevant to the future. S_i is state and A_i is action at time i, then $P(S_{t+1}|S_0, A_0, .., S_t, A_t){=}P(S_{t+1}|S_t, A_t)$. Notably, $P(s'|s,a)$ is prob that agent will be in state s' immediately after doing action a in state s.

8. **MDP** is a Markov chain with set A of actions, S states, $P(S_{t+1}|S_t, A_t)$ dynamics, $R(S_t, A_t, S_{t+1})$ reward at time t.



Fully-observable MDP observes S_t when deciding on A_t; in ***Partially observable*** **POMDP** agent has noisy sensor of state, needs to remember its sensing and acting history.

9. **Stationary policy** (strategy) is function $\pi : S \to A$ specifying what action agent will do given a state. Optimal policy is one with max expected discounted reward, always exists for fully-observ MDP. Optimal policy is denoted $*$.

10. $Q^\pi(s,a)$ is expected value of doing action a in state s then following policy π. $V^\pi(s)$ is expected value of following policy π in state s. Def mutually recursively:

$$Q^{(\pi/*)}(s,a) = \sum_{s'} P(s'|a,s)(r(s,a,s') + \gamma V^{(\pi/*)}(s'))$$

$$V^\pi(s) = Q(s,\pi(s)) \quad V^*(s) = \max_a Q(s,a)$$

$$\pi^*(s) = \text{argmax}_a Q(s,a)$$

11. **Value iteration (VI)**: let V_k, Q_k be k-step lookahead value and Q functions. Set V_0 arbitrarily, compute Q_{i+1}, V_{i+1} from V_i. Converges exponentially fast in k to optimal value func, error reduces proportionally to $\frac{\gamma^k}{1-\gamma}$.

12. Better to individually update value functions for each state, optimal if gets stuck: (**asynchronous** VI, **AVI**). Repeat forever: Select state s and action a, then compute:

$$V[s] \leftarrow \max_a \sum_{a'} P(s'|s,a)(R(s,a,s') + \gamma V[s']) \text{ or}$$

$$Q[s,a] \leftarrow \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma \max_{a'} Q[s',a'])$$

13. **Deterministic Reinforcement learning**: flat, explicit states, indef/infinite stage, fully observable, deterministic, complex preferences, single agent, knowledge learned, perfect rationality.

Experimental AVI for Deterministic RL

```
Q[S,A], s ← arbitrary init, observe curr state s
while true:
 select, perform action a, observe reward r, state s'
 Q[s,a] s ← r+γ max_a' Q[s',a'];    s ← s'
```

14. **Nondeterministic RL**: same, but stochastic. For seq of values $v_1, v_2, ..$ whave running esitmate of average of first k vals: $A_k = \frac{v_1 + ... + v_k}{k} = \frac{k-1}{k} A_{k-1} + \frac{1}{k} v_k$, let $\alpha_k = \frac{1}{k}$, then $A_k = A_{k-1} + \alpha_k(v_k - A_{k-1})$. These are called **temporal differences (TD)**. For fixed α, they converge to average if $\sum_{k=1}^{\infty} \alpha_k = \infty$ and $\sum_{k=1}^{\infty} \alpha_k^2 < \infty$.

15. **Q-learning**: store Q[State,Action] and update same as in AVI, but using experience (empirical probs and rewards)

Q-learning

```
Q[S,A], s ← arbitrary init, observe curr state
while true: ⟨s,a,r,s'⟩
 select, perform action a, observe reward r, state s'
 Q[s,a] += α(r + γ max_a' Q[s',a'] − Q[s,a])    s ← s'
```

For state-action-reward-newState tuple $\langle s,a,r,s' \rangle$:

$$Q[s,a] = Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$$

$$Q[s,a] = (1-\alpha)Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'])$$

16. Q-learning always converges to an optimal policy as long as it tries each action in each state enough. Exploit: do a in state s to max $Q[s,a]$, explore: choose another action.

17. Exploration ϵ-**greedy strategy**: choose random action with probability ϵ and choose best action with $1-\epsilon$. Softmax action selection: in state s, choose a with probability

$$\textbf{Softmax } \epsilon = \frac{e^{Q[s,a] \div \tau}}{\sum_a e^{Q[s,a] \div \tau}}$$

where $\tau > 0$ is the **temperature**: good actions are chosen more often than bad ones, τ defines how much difference in Q-values maps to difference in probability ("optimism in face of uncertainty"). Initialise Q to values encouraging exploration, take into account avg, variance ("upper confidence bounds").

18. **On-policy** learning learns value of policy being followed, unlike **off-policy** Q-learning which learns value of an optimal policy no matter what it does. Use $\langle s, a, r, s', a' \rangle$.

SARSA: On-policy Q-learning

```
Q[S,A], s ← arbitrary init, observe curr state
while true: ⟨s, a, r, s', a'⟩
 select, perform action a, observe reward r, state s'
 a' ← select using policy based on Q
 Q[s,a] ← Q[s,a] + α(r + γQ[s',a'] − Q[s,a])
 s, ← s';      a ← a'
```

19. **Model-based** methods (e.g., value iteration or policy iteration) build or use an explicit model of the environment's transition probabilities and reward function to plan ahead, whereas **model-free** methods (e.g., Q-learning or SARSA) learn value functions directly from sampled experience without ever estimating that model.

Multiagent Systems

1. Agents can be **cooperative**: share util function, **competitive** (zero-sum), or inbetween; Select actions autonomously or *self-interested* - maximising their own util.

2. Use game theory. **Perfect Information Game**: agents act sequentially and can observe state before acting. Each agent can use DP or search, MDP or RL with separate Q functions, maximing for themselves.

3. **Normal(strategic) form of a game** includes a finite set I=$\{1, ..n\}$ of agents each with a set of actions $A_{i \in I}$. An action profile σ=$\langle a_1, ..a_n \rangle$ s.t. agent i takes action a_i, and utility function $u(\sigma, i)$ giving EU for i when **all** agents follow σ. Join actions of all agents σ produces an **outcome**.

4. Can use **payoff matrix** defining possible game decision states, some games need strategy or controllers.

5. **Extensive form of a game** with perfect-info game is a decision(game) tree with state nodes and action arcs. Each node labeled with agent(**nature**), has PD over its children; leaves are outcomes with utility for each agent.

6. **Partially-observable(imperfect-information)** game: need not know the state of world to choose an action (e.g. simultaneous action games). Extensive form of the game uses **information sets** of nodes controlled by same agent with same undistinguishable available actions. **Strategy** chooses one action per info set.

7. **Multiagent decision network** comprises decision nodes labeled with agent that completes them, utility node for each agent and parents specifying available info.

8. Reasoning with imperfect info: choose actions stochastically, when can't move away from a decision, it has reached an **equilibrium**. **Strategy** is a PD over actions of the agent. In stochastic strategy no probabilities are 1.

9. **Strategy profile** σ_i is assignment of strategy to each agent i. Also σ_{-i} is set of strategies of other agents, so $\sigma = \sigma_i \sigma_{-i}$. Utility for i of node n controlled by nature is expected value for i of its children c: $u_i(n) = \sum_c P(c) u_i(c)$ where $P(c)$ is prob that nature will choose c.

10. **Nash Equilibrium** σ_i is best response to σ_{-i} if $\forall$ other strategies σ_i' agent i: utility$(\sigma_i \sigma_{-i}, i) \geq$ utility$(\sigma_i' \sigma_{-i}, i)$. No player can, by themselves, switch to another strategy and improve their expected payoff.

11. Nash equilibrium doesn't guarantee maximum payoff to each agent. Compute it by eliminating ***dominated strategies***, find which actions have non-zero probs (***support set***), determine prob for actions in the support set.

12. Strategy s_i **strictly dominates** if utility$(s_i \sigma_{-i}, i) >$ utility$(s_2 \sigma_{-i}, i)$. When actions $a_1, ..a_k$ have same value for agent, randomise the picking. If all probs $\in (0, 1)$, then Nash equilibrium.

13. Each agent maintains PD over actions $P[A]$ and estimate of value of doing A given σ_{-i} - $Q[A]$. Repeat: select and do action a using P, observe payoff. $Q[a] \leftarrow Q[a] + \alpha(\text{payoff} - Q[a])$. Increment prob of best action by δ, decrement probs of other actions.

Revisited:

Partial-Order Planner (POP)

```
def POP(initial,goal,ops) → plan: # ops=operators
  plan ← Make-Minimal-Plan(initial, goal)
  while not Solution(plan):
    S_need, c ← Select-Subgoal(plan)
    Choose-Operator(plan,ops,S_need,c)
    Resolve-Threats(plan)
  return plan
def Choose-Operator(plan,ops,S_need,c):
  step S_add ← ops or Steps(plan) with effect c
  if not S_add: return fail
  add causal link S_add --c--> S_need to Links(plan)
  add ordering S_add ≺ S_need to Orderings(plan)
  if S_add newly added step from ops:
    add S_add to Steps(plan)
    add Start ≺ S_add ≺ Finish to Orderings(plan)
def Resolve-Threats(plan):
  ∀ S_threat threatening link S_i --c--> S_j ∈ Links(plan):
    Demotion: add S_threat ≺ S_j to Orderings(plan) or
    Promotion: add S_j ≺ S_threat to Orderings(plan)
  if not Consistent(plan): fail
```