Agent - environment framework

① what is AI ?

↳ building intelligent mechanisms that can solve problems often requiring humans

↳ spectra of weak ⟷ strong AI

narrow, solves fewer problems

dynamic & able to solve wide range of problems

② Agent - environment framework

1) AI as search problems

↳ we want systems that can model a real problem & find optimal solutions

↳ one way of modelling this is literally as a search problem

- model all possible solutions
- define some way to measure how good a solution is
- use some algorithm to find the optimal

2) Agent - environment framework

↳ there are many types of problems with different properties
- simple tasks (eg. classification, prediction)
- games (eg. sudoku)

in the most general case, our intelligent agents uses some information to make choices that lead to some result / consequence

⇒ natural way to model choices & consequences is as actions and states

↳ one general (and thus powerful) way to design AI solutions is the agent - environment framework.

- frame the problem (& its constraints) as some environment an agent interacts with, with outcomes of interactions as states

- design how the agent chooses between states to search through the state space & find an optimal solution

③ The rational agent

1) components of an agent

1. percepts : snapshot of an environment at a given time step, perceived through a set of sensors. A tuple. $P_t = (\ldots, \ldots)$
2. percept history : set of all past percepts. $P = \{P_1, P_2 \ldots P_t\}$
3. Actions : set of Actions, A that an agent can take at a given time step, enacted via actuators.
4. agent function : function mapping percept history to an action.

$$f: P \rightarrow a$$

2) rationality

Rational agent : For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built in knowledge the agent has.

3) types of agents

1. simple reflex agents

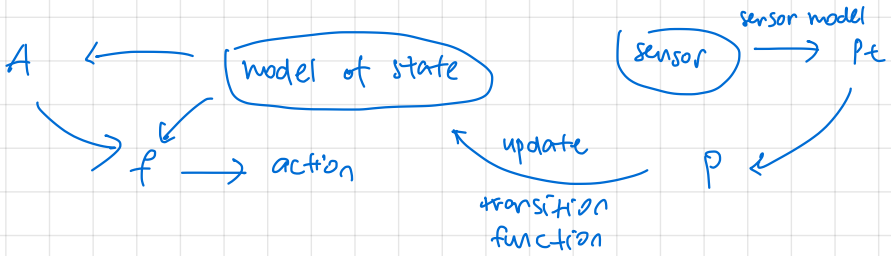
↳ select actions based on current percept, ignoring percept history. Often modelled w rules & if else statements.

function SIMPLE-REFLEX-AGENT(*percept*) **returns** an action
persistent: *rules*, a set of condition–action rules

state ← INTERPRET-INPUT(*percept*)
rule ← RULE-MATCH(*state*, *rules*)
action ← *rule*.ACTION
return *action*

2. model-based reflex agent

↳ makes decisions based on some internalised model that keeps track of how the environment is evolving. passive and acts only upon percept.



function MODEL-BASED-REFLEX-AGENT(*percept*) **returns** an action
  persistent: *state*, the agent's current conception of the world state
    transition_model, a description of how the next state depends on the current state and action
    sensor_model, a description of how the current world state is reflected in the agent's percepts
    rules, a set of condition–action rules
    action, the most recent action, initially none

state ← UPDATE-STATE(*state*, *action*, *percept*, *transition_model*, *sensor_model*)
rule ← RULE-MATCH(*state*, *rules*)
action ← *rule*.ACTION
return *action*

3. goal-based / utility-based agents

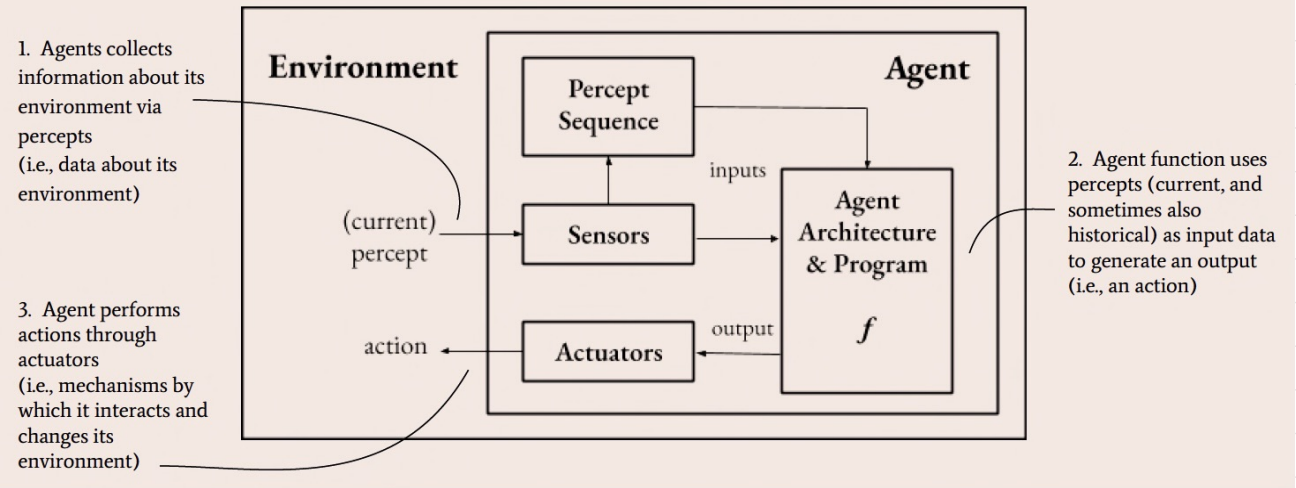
↳ given state & action representations and goal / utility definitions

↳ agent determines sequence of actions necessary to reach goals (binary) or maximise utility (expected) (continuous)

4. learning agents

↳ agents that learn how to optimise performance

- Rational Agent Structure



④ problem environments

↳ when we model problems as an environment to interact with, we have to think of how we might want to do so, since an agent interacts with the environment we define

1. **fully vs partially observable** : whether percepts are full or partial snapshots
   - fully ↓ omniscient
   - partially ↓ not all information can be sensed

2. **deterministic vs. stochastic** : certainty with which actions modify states
   - deterministic ↓ $\text{state}_0, \text{action} \longrightarrow \text{state}_1$, always
   - stochastic ↓ $\text{state}_0, \text{action} \rightarrow \begin{cases} \text{state}_1 \\ \text{state}_2 \\ \text{state}_3 \\ \vdots \end{cases}$ in some p

3. **episodic vs. sequential** : nature of actions — to what extent do past actions restrict / expand current / future actions

   ie. does it depend on percept history or percept?
   - percept history eg. chess
   - percept eg. classification

4. **single vs. multiagent** : cooperative vs. competitive?

5. **static vs. dynamic** : can the environment change while the agent is deliberating?

6. **discrete vs. continuous** : the distinction applies to the state of the environment, the way time is handled and to the percepts and actions of an agent.

   eg. chess (discrete no. of states & actions) vs. driving (continuous locations / velocities — states & degrees of movement)

7. **known vs unknown** : how much does the designer know, or is learning required? Does the agent have enough information to optimise? eg. stochastic but known probabilities
   ⇒ known

AI in simple environments as graph search problems

① abstract search problem formulation

1) problem-solving agents

↳ when the correct action to take is not immediately obvious, an agent may need to plan ahead: to consider a sequence of actions that form a path to a goal state

↳ such an agent is called a problem-solving agent, and the look-ahead process is a (search)

↳ a sequence of actions that reach a goal is a (solution)

in a fully observable, deterministic, known environment, the solution is a fixed sequence of actions

2) abstract search space in simple environments

↳ we build a basic problem-solving agent for a simple environment: one that is fully observable, deterministic, static and known.

↳ A solution is a fixed sequence of actions, that once found, will always be a solution.

↳ we use the following components to model a (search problem)

state an abstract data type that describes an instance of the environment

state space the set of all possible states the environment can be in

initial state the initial state that the agent starts in. atomic & indivisible.

goal state there can be multiple goal states. Any state that is a goal.

goal test isGoal: state → 1/0. function that returns 1 if state is a goal state, else 0

actions fn. Actions: state → {$a_1, a_2 \dots$}. function that returns set of possible actions at a given state. set must be finite.

action cost fn. cost: state$_1$, action, state$_2$ → cost. function that returns cost of taking action at state$_1$ to reach state$_2$.

transition model transition: state, action → new state. function that returns the state transitioned to, when action is applied at original state

solution a sequence of actions form a **path**, and a solution is a path from the initial state to a goal state. An **optimal** solution has the lowest path cost among all solutions.

---

② skeleton of a search algorithm

1) properties of a search algorithm for planning-agents

↳ a search algorithm takes in a search problem and returns a solution or an indication of failure.

↳ for simplicity, we consider algorithms that superimpose a **search tree** or computational graph over the state-space graph.

search tree The search tree describes the paths between states in the state space. The search tree consists of nodes, and may have multiple paths to (and thus multiple nodes of) any given state, but each node in the tree has a unique path to the root.

⇒ search tree is a graph modelling sequence of actions and results of these actions, storing metadata (eg. cost) in nodes

---

node a node in the computation graph is represented by a data structure with four components:

- node.state : the state to which the node corresponds
- node.parent : the predecessor node. following the parent pointers allow us to recover the states and actions along the path to that node. Doing this from the goal node gives solution.
- node.action : the action applied to the parent node to generate this node
- node.pathcost : the total cost of the path from the initial node to this node. we use g(node) to represent this.

---

expanding a node we expand a node by considering the available actions for that state, using the transition function to see where those actions lead, and generating a new node for each of the resulting states note that expand ≠ will put into frontier. just means will look at it to (possibly conditionally) add to frontier

So, takes up memory regardless of added or not

---

reached we say that any state that has a node generated for it has been reached (whether or not it has been expanded)

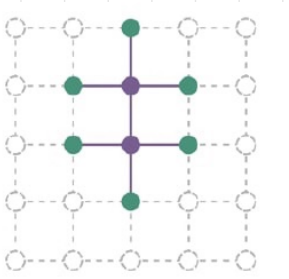
---

explored we say a node has been explored if it has been goal tested and expanded.

frontier as we expand new nodes, we need to systematically keep track of nodes that have been generated but not explored or expanded. we use a queue of sorts to keep track of these nodes.

↳ note that the frontier separates 3 regions of the state space graph: an interior region where every state has been explored, itself, where every state has been reached but not explored (test & expand) and an exterior region of states that have not been reached.

↳ a frontier should be a queue of some kind (eg. FIFO, LIFO (stack), PQ), able to:
- isEmpty(), pop(), top(), add()



2) characterising search algorithms

completeness An algorithm is complete if it will find a solution when one exists and correctly report failure when one does not.

↳ infinite space → algo can eventually reach every state. if complete can't no solution, will never terminate.

↳ finite space → if no ∞ loops in search tree, will be complete

space and time complexity in many AI problems, the graph is represented only implicitly by the initial state, actions, transition model, etc.

↳ for an implicit computation graph, complexity can instead be measured in terms of **optimal depth** d, **maximum depth** m or **branching factor** b

- optimal depth d: number of actions in an optimal solution
- maximum depth m: maximum number of actions in any path
- branching factor b: number of successors of a given node upon expansion

optimal a search algorithm is optimal if it always returns a solution with the lowest cost if it exists

3) tree-search, graph search and redundant paths

repeated state a state that is explored despite having been reached.

cycle A special case of a redundant path that terminates at its root.

redundant path a path (sequence of actions) is redundant if there is another way to reach the same terminal node at lower cost

→ dealing w/ redundant paths

- check for redundant paths
  - keep track of all previously reached states, explore iff. state is not reached or lower path cost has been found (i.e. set to explore)
    - → graph search
  - keep track of all previously reached states, explore iff state is not reached
    - → limited graph search
      - adds to reached on pop
  - check for cycles but not redundant paths in general
- do not check for redundant paths, allowing repetition
  - → tree search
    - note that without redundancy tracking, algorithms are not complete in general.

4) deterministic search and completeness

claim: any deterministic algorithm will search the entire state space in the worst case

idea: suppose there is a deterministic sequence of exploration, which must be finite since the search space is finite and infinite cycles cannot occur. Then we can always find a topological sort of the computational graph and assign a goal to it

0. set g_1 to the goal, and let it search. Let U_1 be the set of unsearched nodes after finding g_1.
1. pick some node g_2 in U_1 and search for it. Because the algorithm is deterministic, at least one more node will be explored. $U_2 \subseteq U_1$.
2. repeat until n, where $|U_n| = 1$. Set the last node as goal, and every node would have been explored.

Uninformed Search

① breadth-first search

implementation	tree search	graph search	limited graph
time complexity	$1+b+b^2 \dots O(b^d)$	no unnecessary repeats. $O(V+E) \le 1+b+b^2 \dots O(b^d)$	$O(V)$
space complexity	$O(b^d)$ all leaves	reached table. $O(V) \le 1+1+b^2 \dots = O(b^d)$	
completeness	1. no cycles 2. finite search space & branching	iff. search space & branching finite	
path-cost optimal	iff. all actions have same cost		

(late goal test) is Goal tested on pop()

↳ suppose goal at depth d. Then would have generated

$1 + b + b^2 \dots + b^d + (b^d - 1)\,b$

(generate, generate ...; all but last; generates b children)

(early goal test) is Goal tested before push()

↳ goal at depth d. Then would generate

$1 + b + b^2 \dots b^{d-1} + b^d$

(all b^d generated by d-1 layers. Last one is goal)

(graph search implementation)

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*
  node ← NODE(*problem*.INITIAL)
  if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
  frontier ← a FIFO queue, with *node* as an element
  reached ← {*problem*.INITIAL} // reached table
  while not IS-EMPTY(*frontier*) **do**
    node ← POP(*frontier*) // pop, can do late goal test here
    for each *child* **in** EXPAND(*problem*, *node*) **do** // generate neighbor nodes
      s ← *child*.STATE
      if *problem*.IS-GOAL(*s*) **then return** *child* // early goal test
      if *s* is not in *reached* **then** // if state not reached
        add *s* to *reached* // update reached
        add *child* to *frontier* // add to frontier
  return *failure*

(graph / graph limited search)

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

---

② uniform cost search

↳ while breadth first search spreads out in waves of uniform depth, Dijkstra or uniform cost search spreads out in waves of uniform path cost

implementation	tree search	graph search	limited graph
time complexity	$O(b^{1+\frac{C^*}{\varepsilon}})$	no unnecessary repeats. $O(V+E) \le O(b^{1+\frac{C^*}{\varepsilon}})$	$O(V)$
space complexity	$O(b^d)$ all at level	reached table. $O(V) \le O(b^{1+\frac{C^*}{\varepsilon}})$	
completeness	1. no cycles 2. finite search space & branching	iff. search space & branching finite	
path-cost optimal	iff. all action have some non-negative cost		not in general

time complexity

1. suppose optimal cost to reach goal is C^*
2. suppose smallest action cost is $\varepsilon > 0$.
3. Then we know that there at most C^*/ε edges from root to the goal.
4. we also do late goal test. so complexity is $O(b^{1+C^*/\varepsilon})$

why do we do late goal test ?

⇓

to ensure optimality. Suppose we didn't, then there might be a smaller thing in the PQ !

space complexity

graph based implementation

```
1: F(Frontier) ← PriorityQueue(u)
2: E(Explored) ← {u}
3: ĝ[u] ← 0
4: while F is not empty do
5:     u ← F.pop()
6:     if GoalTest(u) then                    // late goal test
7:         return path(u)                     ↪ early goal test loses optimality
8:     E.add(u)
9:     for all children v of u do
10:        if v not in E then
11:            if v in F then
12:                ĝ[v] = min(ĝ[v], ĝ[u] + c(u, v))   // update estimate if in frontier
13:            else
14:                F.push(v)
15:                ĝ[v] = ĝ[u] + c(u, v)              // else push and update
16: return Failure
```

graph, limited graph don't have

proof of optimality

idea = PQ & updates / new push to PQ ensures that closest upon goal test

0. claim: when we pop u from frontier, we have found the lowest path cost to it

0-1 notation. $g^*(n)$ = min. cost from root to n.
$g(n)$ = min cost from root to n, so far, when n is in frontier
$g_{pop}(n)$ = min cost from root to n, so far, when n is popped.

1. Let the optimal path from start to goal n be $s_0, s_1 \dots s_n$.

1.1 base step : $g_{pop}(s_0) = 0$, trivial.

1.2 induction hypothesis : $\forall i \in \{0, \dots n\}$, $g_{pop}(s_i) = g^*(s_i)$

1.2.1 Because action cost $\geq \varepsilon > 0$, we know $g^*(s_0) \leq g^*(s_1) \leq \dots g^*(n)$

1.2.2 Also, $g_{pop}(s_i) \geq g^*(s_i)$, since g^* is optimal.

1.2.3 By transitivity, $g_{pop}(n) \geq g^*(n) \geq g^*(s_i)$ for any $i < n$.

1.2.4 Since the frontier is a priority queue, s_i must be popped before s_{i+1}. when we pop s_i, which is connected to s_{i+1}, $g(s_{i+1})$ is updated as

$$
\begin{aligned}
g(n) &= \min\left(g(s_{n-1}) + c(s_{n-1}, n, \text{action}),\ g(n)\right) \\
&\leq g(s_{n-1}) + c(s_{n-1}, n, \text{action}) \quad \text{(by 1.2.3)} \\
&\leq g^*(s_{n-1}) + c(s_{n-1}, n, \text{action}) \quad \text{(by 1.2.3)} \\
&\leq g^*(n)
\end{aligned}
$$

1.25 By the induction hypothesis, optimal.

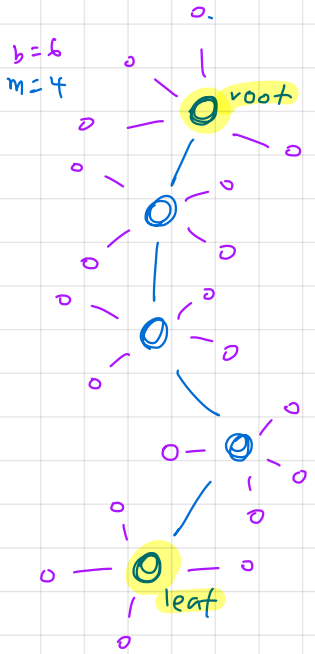
③ space complexity and depth first search

↳ expanding outwards has very high space complexity in general ⇒ time vs. space trade off

implementation	tree search	graph search	limited graph
time complexity	$O(b^m)$	no unnecessary repeats. $O(V+E) \leq O(b^m)$	$O(V)$
space complexity	$O(b \cdot m)$	reached table. $O(V+E) \leq O(b^m)$	$O(V)$
completeness	yes if finite & no cycles	yes if finite space & actions	
path-cost optimal	no in general. returns first solution found.		

time complexity $O(b^m)$, m is max depth

space complexity $O(bm)$, linear path of branches.
can be $O(m)$ if back tracking used, beyond just stack



back tracking search

↳ only 1 successor generated at a time, rather than all successors. Each partially expanded node remembers which successor to generate next

↳ successors are also generated by modifying the current state rather than allocating new memory for a new state. ⇒ $O(bm)$ states ⇒ $O(m)$ actions, 1 state

---

④ depth limited search

↳ to keep DFS from wandering down an infinite path, we can use depth-limited search, a version of DFS that takes a depth-limit l and treates all nodes at depth l as if they had no children.

implementation	tree search	graph search	limited graph
time complexity	$O(b^l)$	no repeats within bound depth. $O(V'+E') \leq O(b^l)$.	
space complexity	$O(bl)$	hash table of bound depth. $O(V'+E') \leq O(b^l)$	
completeness	yes iff. has goal & $l \geq d$, actions are finite, no cycles	yes iff. has goal & $l \geq d$, finite actions	
path-cost optimal	no in general. returns first solution found.		

↳ we can use l to catch most cycles — since longer cycles are handled by the depth limit. we often choose l by the diameter of the state space graph

---

⑤ Iterative deepening search : fusion of BFS & DFS

↳ iterative deepening search solves the problem of picking a good value for l by trying all values, until a solution is found

implementation	tree search	graph search	limited graph
time complexity	$(d)\,b + (d-1)\,b^2 \ldots = O(b^d)$. if graph, $O(V'+E')$ / iteration, but still $O(b^d)$		
space complexity	$O(bl) = O(bd)$	$O(V'+E') \le 1 + b + \ldots b^l = O(b^l) = O(b^d)$	$O(V') \le O(b^d)$
completeness	yes iff. finite search space / actions or solution exists		
path-cost optimal	yes iff. all step costs are the same		

because depth limit cuts off ∞ cycles

time complexity $O(b^d)$ or $O(b^m)$

↳ in IDS, nodes at bottom are generated once, while children of root m times. root not regenerated, since provided. So no. of generated nodes, it have solution,

$$(d)\,b^1 + (d-1)\,b^2 + (d-2)\,b^3 + \ldots (1)\,b^d = O(b^d)$$

memory vs. time

BFS is memory expensive due to having to store a full layer of nodes at any time. DFS saves on that by storing any given path, but is not optimal.

⇒ IDS does DFS in a "BFS way", but repeats nodes. so we sacrifice time for memory

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution node or *failure*
  for *depth* = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* ≠ *cutoff* **then return** *result*

function DEPTH-LIMITED-SEARCH(*problem*, *ℓ*) **returns** a node or *failure* or *cutoff*
  frontier ← a LIFO queue (stack) with NODE(*problem*.INITIAL) as an element // stack
  result ← *failure*
  while not IS-EMPTY(*frontier*) **do**
    node ← POP(*frontier*)
    if *problem*.IS-GOAL(*node*.STATE) **then return** *node* // test on pop, late goal test
    if DEPTH(*node*) > *ℓ* **then** } DFS don't have
      result ← *cutoff*
    else if not IS-CYCLE(*node*) **do** // optional cycle check
      for each *child* **in** EXPAND(*problem*, *node*) **do**
        add *child* to *frontier* // explicit stack rather than recursion stack
  return *result*

heuristics

① informed search & heuristics

↳ uninformed search algorithms find solutions by systematically exploring the search space, achieving completeness and optimality under certain conditions.

↳ can we use domain knowledge to direct the search effort? i.e. where in the search contour should we focus our efforts?

⇒ **heuristic** estimated cost of the cheapest path from the current state at node n to a goal state. denoted $h(n)$.

② properties of heuristics

admissibility

↳ motivation: if we use a heuristic, we want it to always estimate less than the true, so that we will eventually still reach the goal node and maintain completeness

denote $h^*(n)$ the true lowest cost from current state to a goal state.

$h(n)$ is admissible $\iff \forall n \; h(n) \leq h^*(n)$

- n = goal $\Rightarrow h(n) = 0$
- $n \neq$ goal $\Rightarrow h(n)$ never overestimates

proving admissibility

- induction / other techniques
- counterexamples: at least one $h(i)$ overestimates

---

dominance

$h_1(n)$ dominates $h_2(n) \iff \forall n \; h_1(n) \geq h_2(n)$

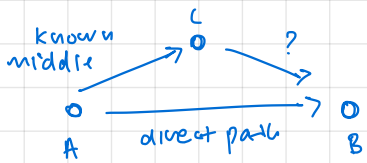
proving dominance

- induction
- counterexamples: $h_1 > h_2$, also $h_2 > h_1$ sometimes
  $\Rightarrow$ does not dominate

consistency

$h(n)$ is consistent $\iff$ $\forall n$, n' that is a child of n generated by action a,
$h(n) \leq h(n') + c(n, a, n')$

↳ idea: a consistent heuristic enforces a constraint of monotonically increasing path costs in its estimates. i.e. if you detour, your estimated detour cost must be at least as much as the direct path.



consistent = then estimate of direct to B is at least as much a less direct route

$h(A) \leq c(A, C) + h(C)$

1. if $h(n)$ is consistent, then $f(n) = g(n) + h(n)$ is non-decreasing along any path

let n' be a successor of n. Then:

$f(n') = g(n') + h(n')$

$= g(n) + d(n, n') + h(n')$ by definition of $g(n')$

$\geq g(n) + h(n)$ by definition of consistency

$= f(n) \implies f(n') \geq f(n)$

we assume transition cost ≥ 0.
A non-consistent heuristic

2. $h(n)$ is consistent $\implies$ $h(n)$ is admissible

1. base case: $k = 1$, node is 1 step from G.

1.1 $h(n) \leq c(n, a, G) + h(G) = h^*(n)$ since $h(G) = 0$ as h is consistent.

1.2 so $h(n) \leq h^*(n)$ and n is admissible.

2. backward induction. suppose our assumption holds for every node $k-1$ steps from G.

2.1 so $h(n-k+1) \leq h^*(n-k+1)$.

2.2 $h(n-k) \leq c(n-k, n-k+1, a) + h(n-k+1)$ by consistency of $(n-k)$

$\leq c(n-k, n-k+1, a) + h^*(n-k+1)$ by admissibility of $(n-k+1)$

$= h^*(n-k)$ by definition

2.3 so $h(n-k) \leq h^*(n-k)$.

③ aggregating heuristics

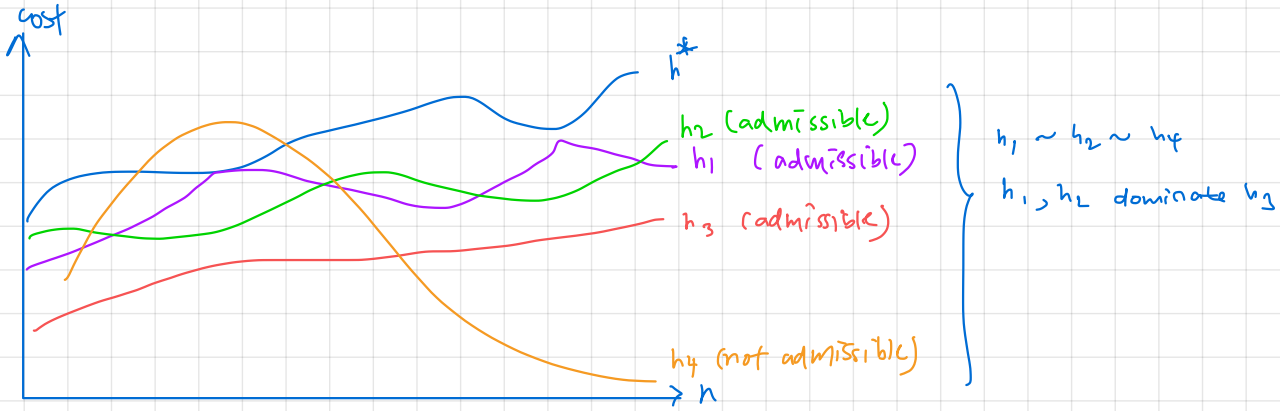
↳ we can aggregate heuristics by adding, min, max etc.
↳ interesting properties arise

min / max 2 heuristics.

both admissible ⟶ max / min admissible
1 admissible, 1 not ⟶ min admissible, max not
neither admissible ⟶ neither min / max always admissible

admissibility and dominance

if h_1 dominates h_2 and h_1 and h_2 are admissible, h_1 is at least as close to h^* as h_2 (use in A* also more efficient)



---

④ creating admissible heuristics

↳ we want to identify heuristics that underestimate the true cost, but one that is easy to compute
⇒ idea: make the game easier!

eg. can only move in straight lines (manhattan) ≤ vs. anyhow (euclidean)

properties of relaxed problems note that relax must strictly be easier

1. any optimal solution in the original problem is also a solution in the relaxed problem, but not necessarily the other way around
2. the cost of an optimal solution in the relaxed problem is an admissable heuristic for the original problem
3. Relax less is better (closer to h^*)

⑤ preservation of consistency

1. suppose we add new edges to the transition graph. If h were initially consistent, would it still be consistent?

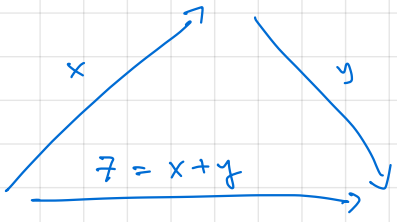
↳ initial consistency implies heuristic always considers a detour to be at least as costly as a direct route

↳ addition of new detour would not fundamentally change how the heuristic operates to estimate

⇒ yes, would still be consistent

Notes

1. we can use triangle inequality to derive heuristics



$\|z\| = \|x+y\| \leq \|x\| + \|y\|$

2. a consistent heuristic is admissible iff. $h(G) = 0$
   eg. consider consistent & admissible h.

   then set $h_1 = h + 1$. ⇒ consistent, but not admissible

heuristic-informed search

① best first search

↳ idea: uninformed search algorithms strictly use path cost (or rather, edge weight) to rank nodes to explore, so they systematically explore contours (IDS, BFS, UCS).

⇒ can we use heuristics/prior knowledge to attend to better / more likely nodes?

⇒ generalize to [evaluation function] ⇒ PQ ranked by evaluation

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element      (best first search)
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then   } graph implementation
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

---

② greedy best first search

↳ UCS, but instead of path cost, greedy use heuristic. $f(n) = h(n)$.

⇒ always expands node that estimated to be closest to goal

implementation	tree search	graph search	limited graph
time complexity	$1+b+\dots = O(b^d)$	$O(V+E) \le O(b^d)$	$O(V) \le O(b^d)$
space complexity	leaves. $O(b^d)$	hash table. $O(V) \le O(b^d)$	
completeness	not in general. heuristic on cycle trivial cycle. $\Longleftrightarrow$	yes iff. finite space & actions	
path-cost optimal	not in general, since does not take actual cost into account		

③ A* search

↳ idea : include both information about path cost & estimated cost to goal.

$$f(n) = g(n) + h(n)$$

implementation	tree search	graph search	limited graph
time complexity	$1 + b^1 + b^2 \dots b^d = O(b^d)$	$O(V+E) \leq O(b^d)$	$O(V) \leq O(b^d)$
space complexity	leaves . $O(b^d)$	hash table . $O(V) \leq 1 + b^1 \dots b^d = O(b^d)$	
completeness	Yes iff - finite space & action, no cycle	Yes iff - finite state space & actions	
path - cost optimal	yes iff - $h(n)$ is admissible		Yes iff . $h(n)$ consistent

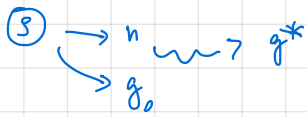
proof of optimality under admissibility

↳ consistency ⟶ admissability

tree & graph based search will expand a node ← long as its not at goal, updating $g(n)$ explicitly (graph, by BST or lazy adding) or implicitly (tree, by allowing revisiting) ⇒ will find optimal

claim : any intermediate node on an optimal path must be expanded before a suboptimal goal node.

0. suppose suboptimal g_0 expanded before node n on optimal path to g^*, at a given node s.



1. then $f(g_0) \leq f(n)$ since g_0 popped first

2. Also, since g_0 is suboptimal we know that $f(g^*) = g(g^*) + \cancel{h(g^*)}^{0} < f(g_0)$

3. Also, $f(g^*) = g(g^*) = g(n) + d(n, g^*)$. By transitivity, $g(n) + d(n, g^*) < f(g_0)$

4. since h is admissible, $h(n) \leq d(n, g^*)$. so $g(n) + h(n) = f(n) \leq g(n) + d(n, g^*) < f(g_0)$

5. so we have $f(g_0) \leq f(n)$ and $f(n) < f(g_0)$ simultaneously. contradiction.

proof of optimality under consistency

⇒ tree, graph & limited graph

limited graph search does not revisit nodes, ever. so for it to be optimal, once popped from frontier, optimal path to node must have been found.

suppose you at s, and you know A and B, and A is more promising than B. 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.

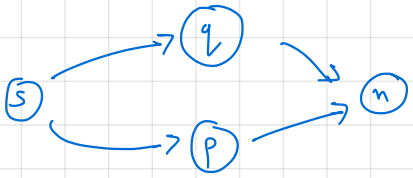
1. consistency ⇒ $f(n) = h(n) + g(n)$ is non-decreasing along any path

2. A* explores nodes in non-decreasing order of f. n expanded before n', then we know $f(n) < f(n')$.

3. suppose p is a node on optimal path before n, q is on suboptimal before n. If limited graph not optimal, then must reach n via q first. so q pop first. By 2, $f(q) < f(n) < f(p)$.

4. But because of optimality & cost $\geq \varepsilon > 0$, we know $f(n) > f(p) > f(q)$. contradiction.

④ Effect of heuristic accuracy on performance



↳ efficiency of A* depends on the accuracy of h

⇒ if $h_1 \geq h_2 \ \forall n$, then h_1 dominates h_2.

if h_1 admissible, then h_1 closer to h^*, and at least as efficient as h_2.

Goal search : hill climbing

① path search, goal search and local search

1) path vs. goal search

↳ so far, we have formulated AI tasks as a search problem, by modelling states as nodes and traversing state space graphs in search algorithms

↳ but built into search algorithms are the notion of a sequence a steps (intermediate states) — what if we relax this requirement? what if we only need some goal state?

2) local search

local search algorithms that operate by searching from a start state to neighbouring states, without keeping track of the paths nor the set of states that have been reached.

advantages

1. they use very little memory
2. they often find reasonable solutions in large or ∞ state spaces for which systematic algorithms are unsuitable

disadvantages

↳ they are not systematic. There could be redundancies or might never explore a portion of the search space

3) formulating a local search problem

state an abstract data type that describes an instance of the environment

initial state the initial state that the agent starts in. atomic & indivisible.

successors of a state the new state brought about by an action from the current state

actions fn. Actions : state → $\{a_1, a_2 \dots\}$. function that returns set of possible actions at a given state. set must be finite.

objective function / value value of a state, measured by an objective function reflects the 'quality' of a state. Should be unique for goal states.

transition model transition : state, action → new state. function that returns the state transitioned to, when action is applied at original state

Algorithm & stopping criteria Depends on algorithm

② hill-climbing search

1) hill climbing algorithm

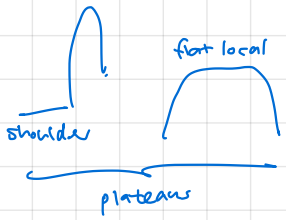
idea algorithm starts at an initial state. It keeps track of one current state and on each iteration moves to the neighboring state with the highest value (direction of steepest ascent)

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  current ← *problem*.INITIAL
  while *true* **do**
    neighbor ← a highest-valued successor state of *current* // find best successor
    if VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current* // if no more better successor, terminate
    current ← *neighbor* // else repeat

2) problems with hill climbing search

local maxima ⇒ always moving in direction of steepest ascent may mean pursuing closer local maxima for further global

plateaus
shoulders
flat local maxima
terminating when all neighbours are equal may lead to early termination



3) hill climbing variants

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 current state. Good for problems w high branching factor

Random restart hill climbing local maxima might not be a solution state. So keep restarting at every failure w a new random initial state, until solution found.

suppose $p_{success}$ of each search $= p$.

Then $E(searches) = 1/p$. $E(computation) = (1)\,E(steps_{success}) + (1/p - 1)\,E(steps_{failure})$

Sideways movement relax check to if value(successor) ≥ value(current). often limit to some n sideways moves.

③ local beam search

1) beam search

idea: keeping one node in memory is a little extreme. Local beam search keeps track of the top k states rather than just one.

1. initialize w/ k states
2. For each k states, generate O(kb) successors.
   - 2.1 choose top k among all successors.
   - 2.2 if any of top k successors are goal, return.
   - 2.3 else, repeat

} better than parallel random restarts, since information is essentially shared by choosing top k among all successors generated

2) stochastic beam search

↳ local beam search can suffer from a lack of diversity among the k states, becoming clustered in a small region of the space ⇒ results in just a slower version of hill climbing

↳ stochastic beam search: samples k from O(kb) successors, with probability proportional to successor's value

---

Notes

1. remember that hill climbing looks for & tests complete solution → eg. graph ↓ random walks !

Systematic goal search : constraint satisfaction

① Defining CSPs

1) CSP idea

↳ in previous techniques, we use atomic states to represent a problem, like snapshots

↳ another way is to represent the search problem using factored representations, that indicate remaining possibilities

⇒ ie. previously state —action()→ actions —transition→ state

now possibilities —try something→ feedback —update→ possibilities

↗ prune search spaces!

any state that does not satisfy a constraint should not be further explored

⇒ how should we formulate this factored representation of the problem?

2) definitions

(X) set of variables $\{X_1, X_2 \dots\}$ that are being solved for

(D) set of domains $\{d_1, d_2 \dots\}$ that each variable can take

(C) set of constraints that specify allowable combinations of values of variables

variable value being solved for

domain values variable can take. can be discrete/continuous, infinite/finite

constraint Describes a necessary relationship, (rel), between a set of variables in its (scope).

eg $C_1 = \{\{X_1, X_2\}, X_1 > X_2\}$ (scope: $\{X_1, X_2\}$; rel: $X_1 > X_2$)

unary |scope| = 1 often by domain

binary |scope| = 2

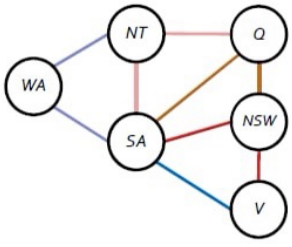
global |scope| > 2

constraint graph constraint graphs represent the constraints in a CSP

↳ simple vertex = variable (Xi)

↳ linking vertex : for constraints □

↳ edge : links all variables in the scope of a constraint (rel)



map coloring constraint graph

T

assignment

- **complete** — assignment where every variable is assigned a value
- **consistent** — assignment that does not violate any assignments
- **solution** — assignment that is complete and consistent
- **partial** — assignment that has some variables unassigned
- **partial solution** — partial assignment that is consistent

3) csp formulation

1. state representation
   - 1.1 set of variables X $\{x_1, x_2 \dots\}$
   - 1.2 domains, D $\{d_1, d_2 \dots\}$
   - 1.3 set of constraints, C $\{(scope, rel) \dots\}$
2. goal test : test that no constraints violated
3. actions : assignment of value to variable

---

② solving csps : backtracking algorithm

some intuition

↳ naive algorithm : much like BFS, branch and try to assign. Problem : a lot of redundant computation for goal search, since order (path) does not matter to solution ⇒ n variables, $\max |d| = m$ ⇒ $n! \cdot m^n$ leaves

at depth l : $(n-l) \cdot m$ branches, since can choose any of the remaining $n-l$ unassigned variables, each is at most domain size of m

then total no. of states $= nm \times (n-1)m \times (n-2)m \dots$

$= n! \cdot m^n$

↳ instead: we use DFS-like algorithm that dynamically updates branches, backtracks and updates knowledge, pruning as you go (kinda like memoising DP) ⇒ m^n leaves

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, assignment)
      if inferences ≠ failure then
        add inferences to csp
        result ← BACKTRACK(csp, assignment)
        if result ≠ failure then return result
        remove inferences from csp
      remove {var = value} from assignment
  return failure
```

1. Determine the variable to assign to
2. Determine the value to assign
3. Trying to determine if the chosen assignment will lead to a terminal state — and also update domains downstream

Continues recursively as long as the ***assignment*** is ***viable***

General purpose heuristics for 1, 2 and 3 can lead to improved search efficiency

order doesn't matter. At depth l, we follow a pre-determined sequence of variables.

↳ So at depth l, branching factor is m (max domain size)

⇒ at most m^n to n states

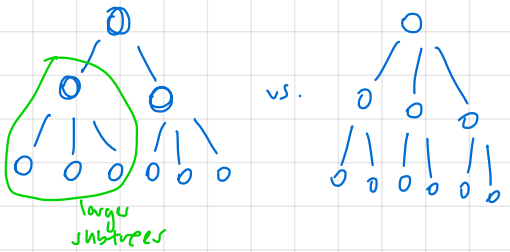
---

③ variable ordering

1) minimum remaining values heuristic

↳ when going down from root to leaf, if we can choose variables w/ less valid values (smaller domain), we can eliminate larger subtrees earlier

↳ visual proof: $|D_{X_1}| = 3$, $|D_{X_2}| = 2$



2) degree heuristic

↳ idea: choose variables w/ most constraints first, then when you propagate constraints down, downstream branching factor smaller (since limited by constraints imposed)

↳ heuristically less useful than MRV, but a useful tie breaker

④ value ordering

i) least constraining value heuristic

↳ idea: when we chose sequence of variables, we wanted most constraining (highest degree) since that would constrain downstream choices & thus reduce branching factor.

But when we choose order of values, we want to find solution as soon as possible, and avoid being blocked by "unnecessary" constraints

⇒ we choose the least constraining value to avoid failure / keep options open downstream → increase no. of downstream options that lead to success

↳ observe that if all solutions needed, then variable ordering does not matter

↳ how? some sort of look ahead on effects on domain size of next variable is needed

---

⑤ inference : constraint propagation

i) notions of consistency

↳ an atomic-state search algorithm makes progress in only one way — by expanding a node to visit successors. This manner of progress could result in "redundant" computation since we do not constrain future values as we set values & explore

↳ constraint propagation in solving CSPs is precisely that — every time we assign a value to a given variable, we go deeper down the search tree only w valid values

the key idea is **local consistency** = in the constraint graph, the process of enforcing local consistency [w domain & rel. constraints] causes inconsistent (ie. violating) possible assignments to be eliminated

node consistency a variable (node in constraint graph) is node-consistent if its domain is consistent w unary constraints (ie. domain is valid).
Normally done at preprocessing step. Trivial.

arc consistency a variable is arc consistent if every value in its domain satisfies the variables binary / global constraints.

X_i is arc consistent w X_j ⟺ for every value in D_i, there is some value in D_j that satisfies binary constraint on the arc (X_i, X_j)

every binary constraint has two arcs! (ie. check both ways)

↳ a graph is arc-consistent if every variable (node) is arc consistent

⇒ eliminate invalid downstream values, & if in process realise this assignment will hit dead end, immediately back track

↓

how do we propagate these constraints (set by assigning value "now") downstream? ⇒ (inference algorithms)

3) inference algorithms

input: factored representation of (X), (D), (C), variable & assignment

output: updated factored representation to (D) + flag whether to terminate (if assignment found to not be viable from constraint propagation to downstream variables)

(forward checking)

1. given (partial) assignment & constraints, update domains of all unassigned variables to remaining legal value
2. if any downstream domain becomes empty → return terminate flag

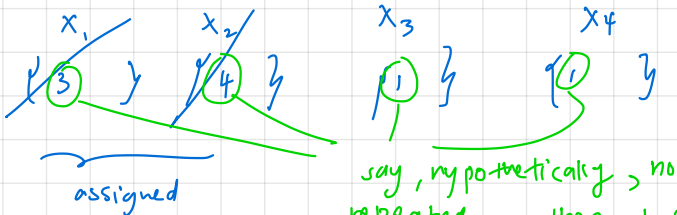
} (step to forward by update)

(complexity) need to check n nodes, all neighbours of each node & update its domain accordingly

↳ binary constraint graph → $2 \cdot {}^{n}C_2 = n(n-1)$ checks ⇒ $O(n^2)$
↳ each node, at most d values to check through ⇒ $O(d^2)$
→ $O(n^2 d^2)$ complexity

⇒ problem: does not provide early detection for all variables and assignments, since it uses domain size without considering how those domains relate to each other

eg.



say, hypothetically, no values can be repeated — then 1 are both legal. But both cannot only have 1 left ⇒ forward checking still has redundancies

AC3

↳ idea: check for arc consistency systematically

1. maintain a queue of arcs (one direction) to check for arc consistency in that direction. X_i wrt. X_j
2. if D_i is updated, then we queue all arcs (X_k, X_i) where X_k is a neighbour of X_i, since the value in D_i that made X_k consistent wrt. X_i may no longer be there
3. If D_i is revised down to nothing, return failure.
4. Return true if arc-queue is empty

---

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in *csp*
  while *queue* is not empty **do**
    (X_i, X_j) ← POP(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to *queue*
  return *true*

Initialise a queue containing all arcs (both directions for each binary constraint)

Each time a variable Xi's domain is updated add all arcs corresponding to binary constraints with other variable (not Xi) as target (except the one that just caused the revision)

function REVISE(*csp*, X_i, X_j) **returns** true iff we revise the domain of X_i
  revised ← *false*
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      revised ← *true*
  return *revised*

Eliminating domain values of the target variable X_i relative to the other variable X_j in the binary constraint

45

---

complexity need to check n nodes, all neighbours of each node & update its domain accordingly

↳ binary constraint graph → $2 \cdot {}^{n}C_2 = n(n-1)$ arcs ⇒ $O(n^2)$
↳ each node, at most d values to check through ⇒ $O(d^2)$
↳ each arc can be inserted at most d times (incremental reduction) ⇒ $O(d)$

→ $O(n^2 \cdot d \cdot d^2)$

⑥ complexity of backtracking search

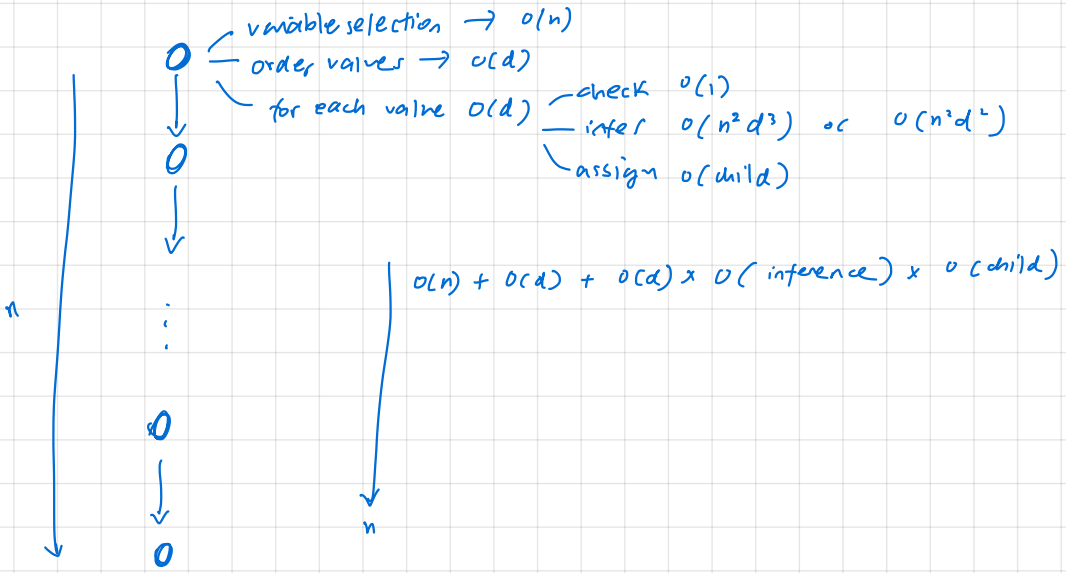
```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, assignment)
      if inferences ≠ failure then
        add inferences to csp
        result ← BACKTRACK(csp, assignment)
        if result ≠ failure then return result
        remove inferences from csp
      remove {var = value} from assignment
  return failure
```

1. Determine the variable to assign to
2. Determine the value to assign
3. Trying to determine if the chosen assignment will lead to a terminal state

Continues recursively as long as the ***assignment*** is ***viable***

General purpose heuristics for 1, 2 and 3 can lead to improved search efficiency



Adversarial search

① Formulating games

1) modelling games

↳ there are three main ways we can model games — as an economy, a part of a non-deterministic environment, and explicitly adversarial agents. They serve different purposes.

↳ for adversarial games, it makes most sense to use the third case

2) definitions and game formulation

perfect information games games that are fully observable

zero-sum games games where what is good for one player is just as bad for the other: there is no "win-win" outcome

↳ also known as "constant sum" games → $u(p_1) + u(p_2) = k$. completely adversarial games.

state abstract data structure describing a state

initial state S_0 state describing how the game is set up at the start

to-move (s) function that checks and returns whose move to make at state s

action (s) gives set of legal moves at state s action : s → {a ...}

result (s, a) Function. Transition model which defines the resulting state from taking action a at state s.

is-terminal (s) function. Returns true if state s is one where the game is over.

utility (s, p) function. Defines & returns a numeric value to player p when the game ends in terminal state (s).

game tree As with graph search, we can impose a search tree to determine what move to make. We define a complete game tree as a search tree that follows every sequence of moves all the way to a terminal state.

↳ note that the game tree can be unbounded

strategy set of moves P_1 will play at every node of the game tree that P_2 plays. state, P_2 move → P_1 move.

winning strategy A strategy s_1^* is a winning strategy if for all strategy s_2 by P_2, the game ends w P_1 winning.

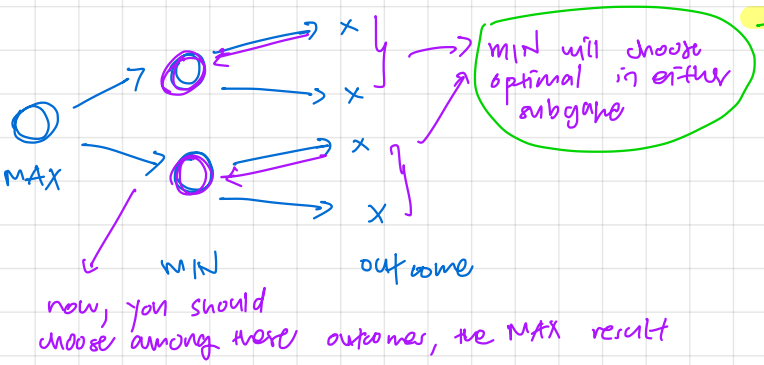
non-losing strategy A strategy t_1^* is non-losing if for any strategy s_2 by P_2, the game ends w either a tie or win for P_1.

3) optimality via backward induction (subgame nash equilibria)

↳ how do we play a "perfect" / "optimal" game?

↳ suppose we have perfect information in a deterministic game. Idea: look ahead to see what opponent would optimally do, and based on that decide what you would do in that "subgame".

↳ then backward propagate that result, and keep going back (backward induction)



→ find dominant (strongly/weakly) strategy at every subgame

---

① minimax

↳ minimax is precisely based on the idea of backward induction

↳ but differently, it is also a greedy algorithm

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

if terminal, evaluate

if max player, choose max among what min could do

if min player, choose min among what max might do

considers players implicitly

start w MAX player

```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move

function MAX-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
        if v2 > v then
            v, move ← v2, a
    return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
        if v2 < v then
            v, move ← v2, a
    return v, move
```

alternating max and min

completeness yes, if game tree is finite

optimal yes if MIN player plays optimally

time complexity $O(b^m)$ searches whole tree of depth m

space complexity $O(bm)$ by same reasoning as DFS

problem: game trees are massive and deep

---

(3) heuristic cutoff search & design of utility function

1) type A and B → ignore/ moves that look bad (some threshold) and follow promising lines as far as possible → deep but narrow

considers all moves to a certain depth in the search tree and then uses a heuristic evaluation function to estimate the utility of states at that depth

→ wide but shallow

cutoff test depth == limit or .isTerminal()

2) evaluation functions

$Eval(s) : S \longrightarrow \mathbb{R}$

$$eval(s, p) = \begin{cases} utility(s, p) & \text{if Termin-} \\ utility(loss, p) \leq x \leq utility(win, p) & \end{cases}$$

3) what makes a good evaluation function ?

1. easy to compute in features of the state
2. corr. w/ chances of winning

linear weighted sum: $\sum_i w_i f_i(s)$

non-linear weighted sum: $g(\{f_i(s)\}_{i=1}^{i=n})$

3. relative vs. absolute — if s_1 state were more likely than s_2, we do not need $Eval(s_1) = 2\,Eval(s_2)$ — simply $Eval(s_1) > Eval(s_2)$

---

④ α-β pruning

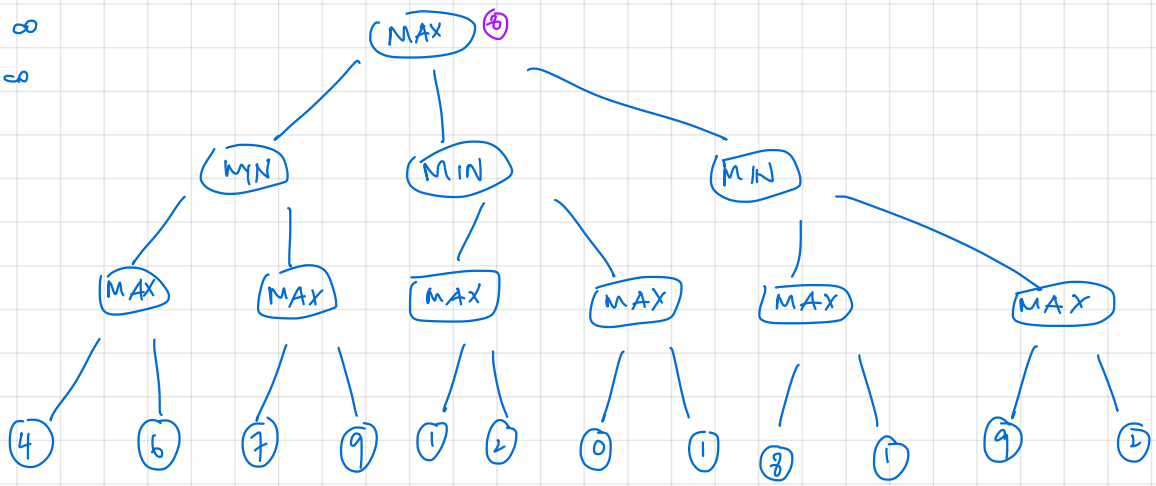
↳ the no. of game states is exponential in depth of tree.

↳ α-β pruning considers carefully what is unnecessary evaluation and blocks them after enough information about a given branch is had

⇒ key idea: minimax backward induction implies other player will choose what's best for them. so we keep track of what options a parent node has as we explore child nodes, and stop when we realize further exploration will not change the outcome

$\alpha = -\infty$

$\beta = +\infty$



(β) bound on what MIN guaranteed above

(α) bound on what MAX guaranteed above

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move
```

i.e. MAX turn

```
function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞  // initialize
  for each a in game.ACTIONS(state) do  // examine each action
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)  // see what the min player would do
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move
```

β passed in

then β is a bound on what MIN player is guaranteed above

update α, best move so far for MAX

if move value ≥ β i.e. worse than current best for MIN player, stop searching (since what prop up, won't matter)

```
function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

α passed in

α is guaranteed at level above. If outcome worse, then doesn't matter, since MAX player won't choose this

move ordering

↳ the effectiveness of α β pruning is highly dependent on the order in which the states are examined, since we cannot prune choices that could be worse

↳ perfect move ordering allows us to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ → branching factor b → $\sqrt{b}$

↳ with random move ordering, roughly $O(b^{3m/4})$

technique

1. draw tree
2. propagate up every time, compare at levels above

⑤ Stochastic games

1) stochastic game tree : weighted graph in edges as probability

2) expecti - minimax : value propagated = $\sum p \times$ value

---

⑥ limitations of game search

1. suboptimal opponent : backward induction assumes opponent plays optimally in looking ahead. Suboptimal player may cause choice to be suboptimal
2. reliance on heuristic design

Logical inference and knowledge agents

① propositional logic

De Morgan's Laws	$\neg(p \vee q) \equiv \neg p \wedge \neg q$	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
Idempotent laws	$p \vee p \equiv p$	$p \wedge p \equiv p$
Associative laws	$(p \vee q) \vee r \equiv p \vee (q \vee r)$	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Commutative laws	$p \vee q \equiv q \vee p$	$p \wedge q \equiv q \wedge p$
Distributive laws	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
Identity laws	$p \vee False \equiv p$	$p \wedge True \equiv p$
Domination laws	$p \wedge False \equiv False$	$p \vee True \equiv True$
Double negation law	$\neg\neg p \equiv p$	
Complement laws	$p \wedge \neg p \equiv False \wedge \neg True \equiv False$	$p \vee \neg p \equiv True \vee \neg False \equiv True$
Absorption laws	$p \vee (p \wedge q) \equiv p$	$p \wedge (p \vee q) \equiv p$
Conditional identities	$p \Rightarrow q \equiv \neg p \vee q$	$p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$

② models & entailment

modelling models are mathematical abstractions, each of which has a truth value for every sentence.

↳ model v corresponds to a set of value assignments

↳ applied to sentence (some literals) α, we say v models α $\longleftrightarrow$ α is true under v

↳ we use $M(\alpha)$ to represent the set of all models where α is true

entailment the idea that a sentence follows logically from another sentence.

↳ sentence $\alpha \models \beta \iff M(\alpha) \subseteq M(\beta)$

i.e. α follows from β iff. for all models where α is true, β is also true

validity A sentence α is valid if it is true for all possible assignments

↳ i.e. tautologies

$\beta \models \alpha \iff \beta \rightarrow \alpha$ is valid

satisfiability A sentence is satisfiable if there exists some assignment such that it is true.

↳ i.e. unsatisfiable sentences are contradictions

$\beta \rightarrow \alpha \iff (\beta \wedge \sim\alpha)$ is unsatisfiable

proof: $KB = True$, thus since $KB \wedge \sim\alpha = False$, $\vee\alpha = False \equiv \neg = True$

knowledge base

↳ how do we model what we know? ⇒ facts! ⇒ represent in logical rules

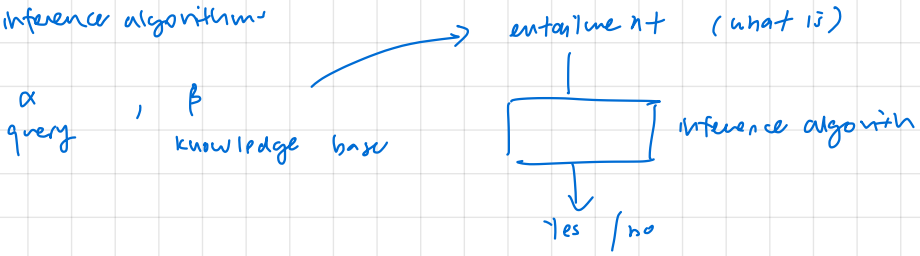
↳ a knowledge base is exactly that — a set of logical rules (that are all known to be simultaneously true) to model what an agent knows

---

③ inference algorithms

logical inference we can apply the definition of entailment to models of knowledge to derive conclusions ie. given knowledge (as rules) can we infer whether some query is true, false or trivially true? (neither)

1) inference algorithms



↳ if an inference algorithm can derive α from some sentence β, we write

$\beta \vdash_i \alpha \iff \alpha$ is derived from β by i

2) properties of inference algorithms

soundness ("truth preserving") inference algorithm A is sound if $\beta \vdash_A \alpha \rightarrow \beta \models \alpha$

↳ in other words, inferences made if it is true

completeness ("truth finding") inference algorithm A is complete if $\beta \models \alpha \rightarrow \beta \vdash_A \alpha$

↳ ie. if α entails β, then inference algorithm will be able to tell
↳ if A is not complete, then A cannot reach all possible conclusions

X: all possible sentences entailed by β

Y: all sentences derived from β using A

complete	sound	
✓	✓	$X = Y$
✓	×	$X \subseteq Y$
×	✓	$Y \subseteq X$

---

④ inference by backtracking

↳ suppose we have a sentence α, and known truths we call the knowledge base KB.

⇒ how do we check if $KB \models \alpha$?

```
function TT-ENTAILS?(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in KB and α   //get all literals
  return TT-CHECK-ALL(KB, α, symbols, { })

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
  if EMPTY?(symbols) then
    if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
    else return true        // when KB is false, always return true
  else
    P ← FIRST(symbols)
    rest ← REST(symbols)
    return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
            and
            TT-CHECK-ALL(KB, α, rest, model ∪ {P = false}))
```

check for entailment

vacuously true

returns true if sentence KB holds in model

recursively explore models where you branch a literal to true and to false

generates the 2^n assignments

↳ effectively enumerates through a truth table of all literals, finds all models where KB is true, then checks that α is also true

↳ $O(2^n)$ time complexity (if finite literals) and $O(n)$ space complexity due to backtracking

---

⑤ propositional theorem proving

i) conjunctive normal forms

CNF a sentence expressed as a conjuction (ANDs) of disjunctions (ORs) is said to be in conjunctive normal form

conversion to CNF for a given sentence

1. eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
2. convert $\alpha \rightarrow \beta$ to $\sim\alpha \vee \beta$ (implication law)
3. move $(\sim)$ negations into brackets by De morgan's / Double negation / distribute

$$\sim(\alpha \wedge \beta) \equiv \sim\alpha \vee \sim\beta$$
$$\sim(\alpha \vee \beta) \equiv \sim\alpha \wedge \sim\beta$$

every CNF formula can be converted to 3-CNF

$$(l_1 \vee \dots l_q) \Leftrightarrow (y_1 \vee l_1 \vee l_2) \wedge (\sim y_1 \vee l_2 \vee \dots l_q)$$

3 CNF

new variable

2) inference by resolution

Resolution rule / operation

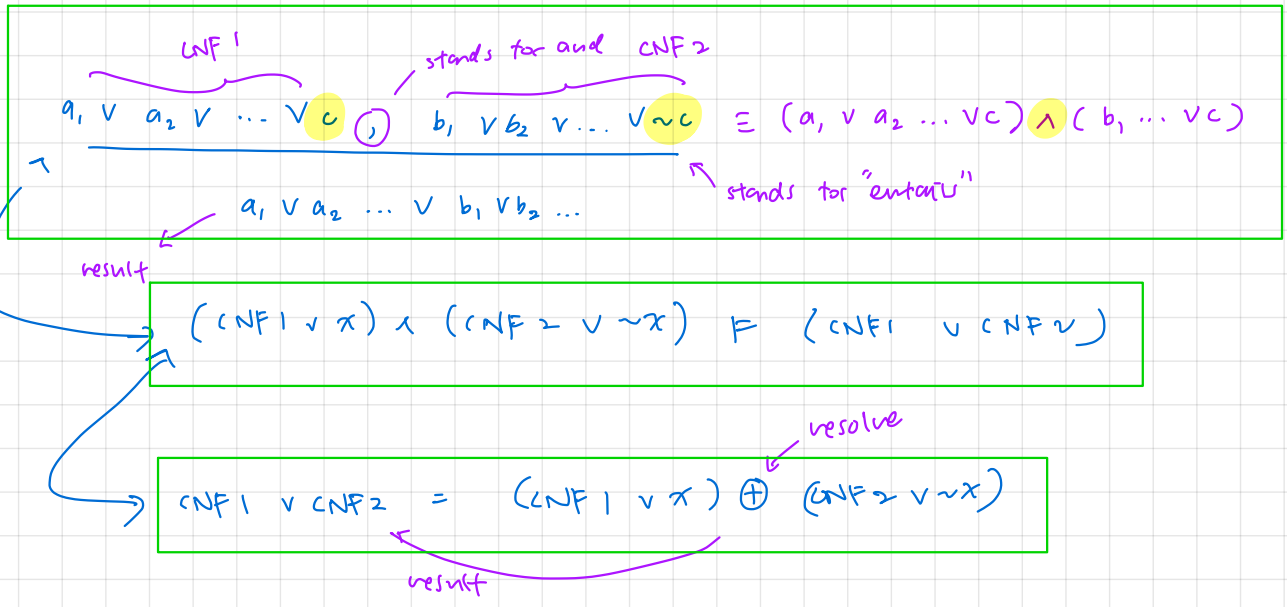
↳ two literals are said to be complements if one is the negation of the other

↳ the resolution ryle exploits the idea that if two disjunctive sentences contain literals that are complementary, they can be eliminated

↳ observe that :

1. if $a_1 \vee a_2 \dots \vee c$ is true, and $b_1 \vee b_2 \dots \vee \sim c$ is true.
2. Then with implication law, $(\sim a_1 \wedge \sim a_2 \dots) \rightarrow c$ and $c \rightarrow (b_1 \vee b_2 \dots)$
3. Then by transitivity, $(\sim a_1 \wedge \sim a_2 \dots) \rightarrow (b_1 \vee b_2 \dots)$
4. Equivalently, $a_1 \vee a_2 \dots \vee b_1 \vee b_2 \dots$

⇒ 5. In other words, if $a_1 \vee a_2 \dots \vee c$ and $b_1 \vee b_2 \dots \vee \sim c$, $a_1 \vee a_2 \dots \vee b_1 \vee b_2 \dots$ is entailed!



Resolution algorithm

↳ with the knowledge that resolutions help us to simplify expressions now can we use it to infer if $KB \models \alpha$?

↳ observe that :

1. $KB \models \alpha \iff KB \rightarrow \alpha$
2. $KB \rightarrow \alpha \iff KB \wedge \sim\alpha$ is not satisfiable

for a given query α, we resolve sentences in $KB \wedge \sim\alpha$ recursively.

→ we end up w an empty expression
$KB \wedge \sim\alpha$ is unsatisfiable
then $KB \models \alpha$
disjunction of no literals holds. A disjunction is only true when at least one literal is true.
$KB \wedge \sim\alpha$ not satisfiable

If we end up w a sentence → then the value in those literals are valid cases where $KB \wedge \sim\alpha$ is true.
Disjunction of some literals are true, so $KB \wedge \sim\alpha$ is satisfiable

- Utilises proof by contradiction – tries to show that KB ∧ ¬α is unsatisfiable

function PL-RESOLUTION(KB, α) **returns** *true* or *false*
inputs: KB, the knowledge base, a sentence in propositional logic
α, the query, a sentence in propositional logic

clauses ← the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
new ← { }
while *true* **do**
  for each pair of clauses C_i, C_j **in** *clauses* **do** ①
    resolvents ← PL-RESOLVE(C_i, C_j) for each pair, resolve
    if *resolvents* contains the empty clause **then return** *true* ②
    new ← *new* ∪ *resolvents*
  if *new* ⊆ *clauses* **then return** *false*
  clauses ← *clauses* ∪ *new*

If cannot be resolved further and not empty clause – cannot infer α

② else add and iterate

if empty clause results, then two sentences cannot coexist

What does an empty clause imply??

③ after one full pass, if resolvents ⊆ clauses, then cannot be resolved anymore → false

④ else do again

Suppose we have a KB as follows:

$$\frac{(x_1 \vee \cdots \vee x_m \vee x) \wedge (y_1 \vee \cdots \vee y_k \vee \neg x)}{(x_1 \vee \cdots \vee x_m \vee y_1 \vee \cdots \vee y_k)}$$

And the algorithm slowly removes literals:

$$\frac{(\cancel{x_1 \vee \cdots \vee x_m} \vee x) \wedge (\cancel{y_1 \vee \cdots \vee y_k} \vee \neg x)}{(\cancel{x_1 \vee \cdots \vee x_m \vee y_1 \vee \cdots \vee y_k})}$$

Eventually, there is nothing in the KB.

KB indicates the disjunction of no literals holds. A disjunction is True only when at least one literal is true. So, whole KB is False here – i.e., the query ¬α is unsatisfiable.

We may infer α (via proof by contradiction).

proof of soundness key idea: only returns true on resolvent being $\emptyset$, and $\emptyset$ indicates not satisfiable

1. The resolution algorithm makes use of the rule that $KB \models \alpha \iff KB \wedge \neg\alpha$ is not satisfiable.
   - 1.1 suppose not. $KB \models \alpha \rightarrow KB \wedge \neg\alpha$ is satisfiable. obvious contradiction.
   - 1.2 Suppose not. $KB \wedge \neg\alpha$ not satisfiable $\rightarrow KB \not\models \alpha$. also obvious contradiction.

2. While pairs of sentences can be resolved, the algorithm will resolve pairs.
   - 2.1 If an empty resolvent results, then the algorithm returns true.
   - 2.2 If an empty resolvent resulted, then the two sentences, which must be true, cannot be true simultaneously, since a disjunction must have at least one true literal to be true.
   - 2.3 Then $KB \wedge \neg\alpha$ is not satisfiable, and $KB \models \alpha$.
   - 2.4 So $KB \vdash_{resolution} \alpha \rightarrow KB \models \alpha$, and resolution is sound.

proof of completeness key idea: if entails, will result in $\emptyset$ by iterative resolution, and return true

0. $KB \models$ resolvent at every step. proof by derivation then induction.
1. suppose not, that there exists an α such that $KB \models \alpha \rightarrow \sim KB \vdash_A \alpha$.
   - 1.1 since $KB \models \alpha \iff M(\alpha) \subseteq M(KB) \iff KB$ true $\rightarrow \alpha$ is true.
   - 1.2 1.3 Then $KB \wedge \neg\alpha$ must not be satisfiable
   - 1.3 let $x_1 \ldots x_n$ be the literals in α. If x_i in $\neg\alpha$, then $\neg x_i$ in KB.
   - 1.4 then iterative application of resolution will result in an empty resolvent and return true contradiction. so $KB \models \alpha \rightarrow KB \vdash_A \alpha$

3) 2NF and implication graphs

1. observe that all 2NF forms w x and y can be represented by four implications.

$$x \vee y \equiv \neg x \rightarrow y$$

$$x \vee \neg y \equiv \neg x \rightarrow \neg y$$

$$\neg x \vee y \equiv x \rightarrow y$$

$$\neg x \vee \neg y \equiv x \rightarrow \neg y$$

2. observe also that $x \rightarrow \neg x \rightarrow x$ is a contradiction, since requires $x = \text{True} \wedge x = \text{False}$.

3. So if we construct a graph w literals as nodes and implications as directed edges, we have an implication graph.

3.1 so if we find a cycle (in polynomial time) we have found a contradiction.

---

⑥ Knowledge based agents

1) Knowledge based agents

- use a process of reasoning over an internal representation of knowledge to decide what actions to take

function KB-AGENT(*percept*) **returns** an *action*
  persistent: *KB*, a knowledge base
  t, a counter, initially 0, indicating time

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
  action ← ASK(*KB*, MAKE-ACTION-QUERY(*t*))
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
  t ← *t* + 1
  return *action*

} maintain KB & time step

① convert percept to sentence, TELL (update) KB with it

② at a given time, decide what you might want to do, query KB if conditions are met ASK

eg. is (1,2) safe?

③ TELL KB that action being taken

2) formulation of a knowledge based agent

knowledge base as defined above. sentences known to be true.

MAKE PERCEPT SENTENCE function that converts x into logical statement

CNF conversion

TELL sentence, KB $\rightarrow$ KB. updates KB w/ sentence.

MAKE ACTION QUERY function that generates all queries at time step t to decide what to do next

ASK KB, $\{\alpha\}$ $\rightarrow$ a. function that queries KB w/ all α, and depending on result of inferences, decides what to do

Notes

1. use implications to represent XOR $\Rightarrow$ $x_1 \rightarrow \sim x_2$
   (also useful for cardinality)

2.

Bayesian Networks

① Basics of probability

Random Variable variable representing outcomes

Domain set of values RV can take

Events subset of domain

Axioms

1. $0 \leq P(A) \leq 1$
2. $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$
3. $P(u) = 1 \quad , P(\phi) = 0$

joint probability $(x, y) \in D_x \times D_y \quad , P(x, y) = P(X = x \wedge Y = y)$

conditional probability

1. $P(A \mid B) = \frac{P(A \wedge B)}{P(B)}$ assuming $P(B) > 0$
2. $P(A) = P(A \mid B) P(B) + P(A \mid B') P(B')$

chain rule

$$P(E_1 \wedge E_2 \wedge E_3 \dots) = \prod_{i=1}^{n} P(E_i \mid E_1 \wedge E_2 \dots E_{i-1})$$

↳ derived from inductive application of Bayes' rule

independence A & B are independent $\iff P(A \wedge B) = P(A) \times P(B)$

$\iff P(A \mid B) = P(A)$ knowing B adds no information about A and vice versa

conditional independence

Given event B, event A is conditionally independent of event C if

$P(A \mid B, C) = P(A \mid B)$

$\Rightarrow$ A & C are related by a shared cause B

$\Rightarrow$ if we take it into account, they become unrelated

② uncertainty & inference by enumeration

↳ suppose our agent needs to reason about things, and needs to find the probability that outcome given percepts

ie. P(outcome of action | percepts)

inference by enumeration

↳ then a naive way would be to store the entire joint distribution of all variables, and use chain rule to compute

⇒ n variables, domain size d ⇒ $O(d^n)$ space & time complexity!

using independence

↳ suppose, for simplicity, all variables tracked are independent

↳ then application of chain rule does not need conditional probabilities!

⇒ n variables, domain size d ⇒ $O(nd)$ space & time complexity

↳ how can we do some sort of inference, but use independence to maintain compact representations?

---

③ Inference with Bayesian networks

1) exploiting conditional independence

↳ suppose all n variables share a cause s. Then the full joint distribution

$P(X_1 \wedge X_2 \dots \wedge s) = P(X_1 | s) \cdot P(X_2 | s) \dots \cdot P(s)$ by chain rule.

⇒ then we only need to maintain $d \cdot n + 1$ size table for any computation

2) Bayesian networks

Bayesian networks

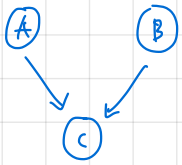
↳ joint distributions represented by a graph

↳ vertices are random variables, an edge from $X \rightarrow Y$ implies X directly influences Y (some corr. assume X causes Y)

↳ Then conditional distribution for each node is $P(X \mid \text{Parents}(X))$

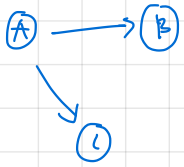
Structures in bayesian networks

independent causes



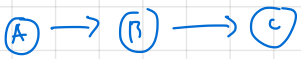
$$P(A \wedge B \wedge C) = P(A) \cdot P(B) \cdot P(C \mid A \wedge B)$$

conditionally independent effects



$$P(A \wedge B \wedge C) = P(A) \cdot P(C \mid A) \cdot P(B \mid A)$$

causal chain



$$P(A \wedge B \wedge C) = P(A) \cdot P(B \mid A) \cdot P(C \mid B)$$

Inference by graph traversal

1. observe that by modelling each node as a distribution conditioned solely on parents, we are implying conditional independence between disconnected nodes

2. To find the joint distribution of some subset of variables, therefore, we simply traverse from root node to target node, along a chain

↓

essentially a realisation of conditional independence and chain rule!

⑭ Inference with Naive Bayes model

↳ suppose we want to infer $P(\text{cause} \mid \underbrace{E_1 \wedge E_2 \dots}_{\text{percepts}})$ i.e. given these observations, we want to find out likelihood that x caused it

↳ additionally suppose all $E_1 \dots E_n$ are conditionally independent on cause.

1. $P(\text{cause} \mid E_1 \wedge E_2 \dots) = \dfrac{P(\text{cause}) \cdot P(E_1 \wedge E_2 \dots \mid \text{cause})}{P(E_1 \wedge E_2 \dots)}$ by conditional prob.

$$= \frac{P(\text{cause})}{P(E_1 \wedge E_2 \dots)} \prod_{i=1}^{n} P(E_i \mid \text{cause})$$

$$= \alpha\, P(\text{cause}) \prod_{i=1}^{n} P(E_i \mid \text{cause}), \quad \alpha = \frac{1}{P(E_1 \wedge E_2 \dots)}$$

2. Now, to compare, the probability of causes given observations, we can do it in relative terms of α!

↳ That is, relative likelihood $\dfrac{P(X \mid E_1 \wedge E_2 \dots)}{P(Y \mid E_1 \wedge E_2 \dots)} = \dfrac{P(X) \prod_{i=1}^{n} P(E_i \mid X)}{P(Y) \prod_{i=1}^{n} P(E_i \mid Y)}$ (the α cancels)

Notes

1. Given 2 random boolean variables A and B, if $P(A \mid B) = 0$ and $P(A \mid \sim B) = 1$, then $P(A) = \sim P(B)$

↳ $P(A) = P(A \wedge B) + P(A \wedge \sim B)$

$= P(B) \cdot P(A \mid B) + P(\sim B) \cdot P(A \mid \sim B)$