CSPs

CSP Formulation

- State representation
  - Variables: `X = {x_1, ... , x_n}`
  - Domains: `D = {d_1, ... , d_k}`
    - Such that `x_i` has domain `d_i`
  - Initial state: all variables unassigned
  - Intermediate state: partial assignment
- Actions, costs, and transition
  - Assignment of values (within domain) to variables
  - Costs are unnecessary
- Goal test
  - Constraints: `C = {c_1, ... , c_m}`
    - Defined via a constraint langauge (algebra, logic, sets)
    - Each `c_i` corresponds to a requirement on some subset of `X`
    - Can be unary (|scope|= 1), binary (|scope|= 2), or global (|scope|> 2)
  - Objective is a complete and consistent assignment
    - Find a legal assignment $(y_i, ..., y_n)$ s.t. $y_i \in D_i\ \forall i \in [1, n)$
    - Complete: all variables assigned values
    - Consistent: all constraints in `C` satisfied

Algo

```
Function CSPSolver(variables, domains, constraints):
    assignments = initial_state            # no assignments made
    while assignments incomplete:
        if no possible assignments left: return failure
        current = assign a value to non-assigned variable
        if current is consistent then: assignments.store(current)
    return assignments
```

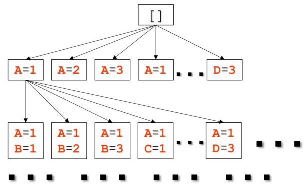
Search tree size

At depth l: $(|X|-l)\cdot|d|$ states

Total number of leaf states: $nm \times (n-1)m \times (n-2)m \times \ldots \times 2m \times m = n!m^n$

where $n = |x|$ and $m = |d|$

Order of variable assignments is not important, just consider assignments to one variable per level (m^n leaves)



Backtracking algorithm

```
def backtrack(csp, assgt) -> bool:
    if assgt is complete:
        return assgt
    var = select_unassigned_variable(csp, assgt)
    for val in order_dom_values(csp, var, assgt):
        if val is consistent with assgt:
            add {var = val} to assgt
            inferences = inference(csp, var, assgt)
            if inferences != failure:
                add inferences to csp
                result = backtrack(csp, assgt)
                if result != failure:
                    return result
                remove inferences from csp
            remove {var = value} from assgt
    return failure
```

Variable-order heuristics

MRV (minimum remaining values)

- Choose variable with fewest legal values
- Most constrained variable
- Sort by domain size, choose variable with smallest domain size
- Places larger subtrees closer to the root so that any invalid state found prunes a larger subtree ⇒ eliminates larger subtrees earlier

Degree Heuristic

- To break ties in MRV heuristics
- Pick unassigned variable with most constraints (highest degree in constraint graph)
- This reduces the branching factor b

Recommended variable selection: MRV, then degree, then random

Value-order Heuristics

LCV (least constraining value)

- Choose the value that rules out the fewest choices from remaining domain values
  - Given assignment of value `v` to variable `x'`, determine set of unassigned variables `U` that share a constraint with `x'` and pick `v` that maximises sum of consistent domain sizes of variables in `U`
- Avoids failure by avoiding empty domains

With variables, fail first. With values, fail last. Cuz need to look at all variables but do to look at all values.

BUT if all solutions required, then value-ordering irrelevant

Forward Checking

Track remaining legal values for unassigned variables, terminate search when any variable has no legal values (based on constraints with recently assigned variable)

Constraint propagation

Inference step to ensure local consistency of all variables

- Traverse constraint graph to ensure variable at each node is consistent, eliminate all values in variable's domain that are not consistent with linked constraints

Node-consistent (vertex-consistent)

- Domain of a variable is consistent with its unary constraints
- Achieved through pre-processing step before backtracking

Arc-consistent (edge-consistent)

- Domain of a variable is consistent with its binary constraints
- The variable's domain value must have a partnering domain value in other variable that will satisfy the binary constraint

X_i is arc-consistent w.r.t $X_j \iff \forall x \in D_i\ \exists y \in D_j$ s.t. binary constraint on arc (X_i, X_j) is satisfied

When checking arc (X_a, X_b), remove values from D_a

- Done during pre-processing or after each variable assignment (expensive)

AC-3 Algo

```
function ac3(csp) -> bool:
    queue = queue of all arcs in csp (both dirs)
    while queue:
        (X_i, X_j) = queue.pop()
        if revise(csp, X_i, X_j):
            if len(D_i) == 0:
                return False
            for X_k in X_i.neighbors - {X_j}:
                queue.append((X_k, X_i))
    return True

function revise(csp, X_i, X_j):
    revised = False
    for x in D_i:
        if no value in D_j allows (x, y) to satisfy
        the constraint between X_i and X_j:
            delete x from D_i
            revised = true
    return revised
```

Time complexity of AC-3

With n variables, there are at most $2 \times \binom{n}{2} \in O(n^2)$ directed arcs

Each arc can be reinserted at most d times because X_i has hat most d values to delete where d is domain size

Checking consistency of arc (`revise()`) takes $O(d^2)$ time

Overall time complexity: $O(n^2 \times d \times d^2) = O(n^2 \times d^3)$

Advesarial search

Assume 2 players, zero-sum game. MAX player wants to maximise value (our agent), MIN player wants to minimise value (opponent)

- If MIN does not play optimally, the actual outcome for MAX can only be better and never worse, but the algo may not select the optimal move.

Formulating games

- State representation: as per general search formulation
- `TO-MOVE(s)`: returns `p`, the player to move in state `s`
- `ACTIONS(s)`: legal moves in state `s`
- `RESULT(s, a)`: the transition model, returns resultant immediate state when taking action `a` at state `s`
- `IS-TERMINAL(s)`: returns `True` when game is over, `False` otherwise
- `UTILITY(s, p)`: defines a numeric value (score) for player `p` when the game ends at terminal state `s`

For zero-sum games, at terminal state `s`,

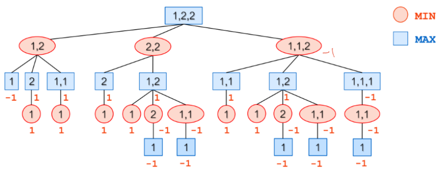
`utility(MAX, s) + utility(MIN, s) = 0`.

Just use `utility(MAX) = -utility(MIN)`

Strategies: optimal decisions via minimax

$$Minimax(s) = \begin{cases} \text{Utility}(s, \text{MAX}) & \text{if Is−Terminal}(s) \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s,a)) & \text{if To−Move}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s,a)) & \text{if To−Move}(s) = \text{MIN} \end{cases}$$

Remember that `Result(s,a)` outputs the successor state (given `a` taken at `s`) (i.e., `MIN` wants state with lowest utility, while `MAX` wants state with highest)



Backwards induction used

Properties

- Complete assuming finite game tree
- Optimal assuming optimal gameplay (opponent plays optimally)
- Time complexity $O(b^m)$
- Space complexity $O(bm)$
- Time polynomial to tree size

Backwards Induction Issue

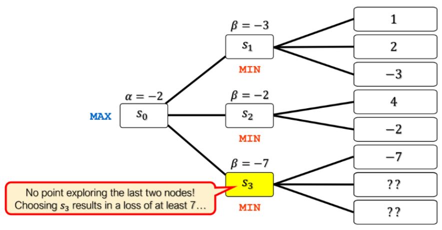
Game trees are massive, impossible to expand entire tree, must find ways to shrink search tree

α - β pruning

General idea: don't explore moves that would never be considered.

α bounds MAX's values, β bounds MIN's values.

Prune subtrees that will never affect Minimax decision



S_3 will be -7 or lower. MAX will not choose it because it already has S_1 or S_2 which are higher than -7.

Minimax algo (a and b are alpha and beta)

```
def minimax(node, depth, isMax, a, b):
    if node is a leaf node:
        return value of node
    if isMax:
        bestVal = -INFINITY
        for each child node:
            value = minimax(node, depth+1, false, a, b)
            bestVal = max(bestVal, value)
            a = max(a, bestVal)
            if b <= a:
                break
        return bestVal
    else:
        bestVal = +INFINITY
        for each child node:
            value = minimax(node, depth+1, true, a, b)
            bestVal = min(bestVal, value)
            beta = min(beta, bestVal)
            if b <= a:
                break
        return bestVal
minimax(0, 0, true, -INFINITY, +INFINITY)
```

Perfect ordering with alpha beta gives time complexity $O(b^{m/2})$ Random ordering gives time complexity $O(b^{3m/4})$

α - β pruning issues:

- Issue: Depth of the tree could be very large
  - Backwards induction only works with terminal states
  - If depth is very deep, it takes very long before we can reach terminal state
- Solution: Heuristic minimax
  - Cutoff test: have a depth limit as to how deep we search for a terminal node
  - Evaluation function: Estimates the expected utility at that state
  - Run MINIMAX until depth d then start using evaluation function to choose nodes
  - Replaces `is_terminal(s)` with `cutoff_test(s, d)`.
  - Replaces `utility(s, p)` with `eval(s, p)`.

Logical agents

Knowledge-based agents

- Represent agent domain knowledge using logical formulas
- Main components of a logical agent are inference engine and knowledge base. Inference engine consists of domain-independent algorithms while knowedge base is domain-specific content

KB agent function

```
def KB_agent(percept) -> action:
    persistent: KB, t (time)

    TELL(KB, MAKE_PERCEPT_SENTENCE(percept, t))
    action = ASK(KB, MAKE_ACTION_QUERY(t))
    TELL(KB, MAKE_ACTION_SENTENCE(action, t))
    t++
    return action
```

Inference via entailment

Entailment

Modelling: v models α (a sentence) if α is true under v

- v corresponds to one set of value assignments
- v corresponds to one instance of the environment (known part of a state)
- Let $M(\alpha)$ be the set of all models for α

Entailment ($\models$): One thing (right) follows from the other (left)

- $\alpha \models \beta \iff M(\alpha) \subseteq M(\beta)$
- E.g. $\alpha = q$ is prime, $\beta = (q$ is odd$) \vee q = 2$

Inference

Inference is deriving new knowledge from the KB

KB: environment rules/laws and percepts

$$KB \models \alpha \implies M(KB) \subseteq M(\alpha)$$

$$M(KB) \cup M(\alpha) \equiv M(\alpha)$$

$$M(KB) \cap M(\alpha) \equiv M(KB)$$

$$M(KB) \cap M(\neg\alpha) \equiv \emptyset$$

Given KB, try to infer α, i.e. determine if $KB \models \alpha$.

If $KB \models \alpha$, then α can be added to KB since $M(KB) \cap M(\alpha) \equiv M(KB)$

Soundness & Completeness

- $KB \vdash_A \alpha$
  - Means "sentence α is derived from KB by inference algorithm A"
- Soundness
  - A is sound if $KB \vdash_A \alpha \implies KB \models \alpha$
  - A will not infer nonsense.
- Completeness
  - A is complete if $KB \models \alpha \implies KB \vdash_A \alpha$
  - A can infer any sentence that KB entails

Truth table enumeration

Draw truth table of KB and α, KB entails α if whenever KB true, α true

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
  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* }))

Properties

- $O(2^n)$ time complexity
- $O(n)$ space complexity
- Sound and complete

Theorem proving methods

Validity & Satisfiability

- Validity
  - A sentence α is **valid** if it is true for all possible truth value assignments
  - **tautology**
  - $(KB \models \alpha) \iff (KB \implies \alpha)$ (deduction theorem)
- Satisfiability
  - A sentence is **satisfiable** if it is true for some truth value assignment (i.e. a model exists for that sentence)
  - A sentence is **unsatisfiable** if it is true for no truth value assignments
    - **contradictions**
  - $(KB \models \alpha) \iff (KB \wedge \neg\alpha)$ is unsatisfiable
    - Definition of proof by contradiction

Inference rules

- And-Elimination: $a \wedge b \models a,\ a \wedge b \models b$
- Modus Ponens: $a \wedge (a \implies b) \models b$
- Logical equivalences / Modus Tollens: $(a \vee b) \models \neg(\neg a \wedge \neg b)$
- Contrapositive: $(a \implies b) \models (\neg b \implies \neg a)$
- Syllogism: $(a \implies b) \wedge (b \implies c) \models (a \implies c)$

CNF

CNF = conjunction of disjunctive sentences, e.g. $(x_1 \vee x_2) \wedge (x_2 \vee x_3 \vee x_4)$

Conversion to CNF

- Convert:
  - $\alpha \iff \beta$ to $(\alpha \implies \beta) \wedge (\beta \implies \alpha)$
  - $\alpha \implies \beta$ to $\neg\alpha \vee \beta$
- Expand $\neg$ using De Morgan's and double negation
- Distributive law to convert $(\alpha \vee (\beta \wedge \gamma))$ to $(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$

Cardinality Rules

Suppose n variables, and we want k to be true.

- At least k
  - DNF: Connect each possible k-way conjunction via disjunctions
    - E.g. $n = 3, k = 2 : (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$
  - CNF: Connect each possible i-way disjunction via conjunctions, where $i = n - k + 1$
    - E.g. $n = 3, k = 2, i = n - k + 1 = 2$, so we use pairwise OR: $(A \vee B) \wedge (A \vee C) \wedge (B \vee C)$
- At most k
  - CNF: Negation of $k + 1$ ways
    - E.g. $n = 3, k = 1$: 1 + 1 (pairwise) negations of all literals: $(\neg A \vee \neg B) \wedge (\neg A \vee \neg C) \wedge (\neg B \vee \neg C)$
- Exactly k
  - DNF: Choose all combinations of the n variables where k literals are true and $n - k$ are False
    - E.g. $n = 3, k = 2$: $(A \wedge B \wedge \neg C) \vee (A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge C)$
  - CNF: Conjunction of at least k and at most k
    - E.g. $n = 3, k = 2$: $(A \vee B) \wedge (A \vee C) \wedge (B \vee C) \wedge (\neg A \vee \neg B \vee \neg C)$

Resolution

If a literal x appears in R_i and its negation $\neg x$ appears in R_j, where $R_i, R_j \in KB$, then it can be removed from both. Then the remaining literals combine to 1 clause

Resolution algo

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**
      resolvents ← PL-RESOLVE(C_i, C_j)
      if *resolvents* contains the empty clause **then return** *true*
      new ← *new* ∪ *resolvents*
    if *new* ⊆ *clauses* **then return** *false*
    clauses ← *clauses* ∪ *new*

- Make a clause list (copy of KB specified in CNF including negation of query $\neg\alpha$)
- Repeatedly resolve two clauses from clause list (add resolvent to clause list)
- Keep doing this till empty clause found or no more resolutions possible
  - If empty clause then can infer α
  - If no more resolutions and not empty clause then cannot infer α

Uncertainties

Conditional probabilities and Bayes RUle

$Pr[A|B] = \frac{P[A \wedge B]}{P[B]}$

Bayes rule: $Pr[A|B] = \frac{Pr[B|A]Pr[A]}{Pr[B]}$

Chain rule

$$Pr[R_1 \wedge \ldots \wedge R_k] = Pr[(R_1 \wedge \ldots \wedge R_{k-1}) \wedge R_k]$$
$$= Pr[R_k | R_{k-1} \wedge \ldots \wedge R_1] \cdot Pr[R_{k-1} \wedge \ldots \wedge R_1]$$

$$Pr\left[R_1 \wedge R_2 \wedge \ldots \wedge R_k\right] = \prod_{j=1,\ldots,k} Pr\left[R_j | R_1 \wedge \ldots \wedge R_{j-1}\right]$$

$$Pr[A \wedge B \wedge C \wedge D] = Pr[D | C \wedge B \wedge A] \cdot Pr[C \wedge B \wedge A]$$
$$= Pr[D | C \wedge B \wedge A] \cdot Pr[C | B \wedge A] \cdot Pr[B \wedge A]$$
$$= Pr[D | C \wedge B \wedge A] \cdot Pr[C | B \wedge A] \cdot Pr[B | A] \cdot Pr[A]$$

$$A \text{ and } B \text{ are independent } \iff Pr[A \wedge B] = Pr[A] \cdot Pr[B]$$
$$\iff Pr[A|B] = Pr[A]$$

Inference via Bayes' Rule

Infer statement of the form "What is the likelihood of an event α given the probabilities of other events"

$$Pr[A \wedge B | C] = \frac{Pr[A \wedge B \wedge C]}{P[C]} = \frac{Pr[A \wedge B \wedge C]}{Pr[B \wedge C]} \times \frac{Pr[B \wedge C]}{Pr[C]}$$
$$= Pr[A | B \wedge C] \times Pr[B | C]$$

Bayesian Networks

- Vertices are random variables
- An edge from X to Y means X directly influences Y
- Each vertex has a conditional distribution given its parents
- In the simplest case, a conditional distribution can be represented as a conditional probability table (CPT)
- CPTs in the BN are the distribution over Z for each combination of parent values

Markov Blanket

- A variable is conditionally independent of all other nodes in the network, given
  - its parents
  - its children
  - its children's parents