

## CSPs

### CSP Formulation

- State representation
  - Variables:  $X = \{x_1, \dots, x_n\}$
  - Domains:  $D = \{d_1, \dots, d_k\}$
  - 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:  $C = \{c_1, \dots, c_m\}$ 
    - Defined via a constraint language (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  $(y_1, \dots, y_n)$  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
```

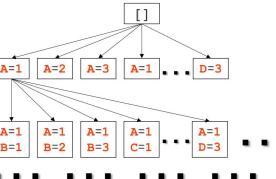
### Search tree size

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

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

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_domain_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  $\Rightarrow$  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 don't 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  $D_a$

#### 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 at most  $d$  values to delete where  $d$  is domain size

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

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

#### Adversarial search

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

#### 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  $utility(MAX) = -utility(MIN)$

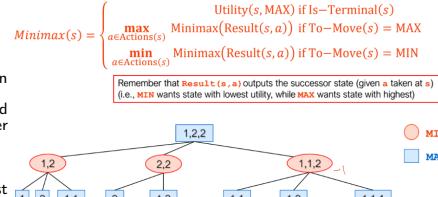
**Strategies: optimal decisions via minimax**

**Minimax(s) =**

$$\max_{a \in \text{Actions}(s)} \min_{a' \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) \text{ if To-Move}(s) = \text{MAX}$$

$$\min_{a \in \text{Actions}(s)} \max_{a' \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) \text{ if To-Move}(s) = \text{MIN}$$

Remember that  $\text{Result}(s, a)$  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

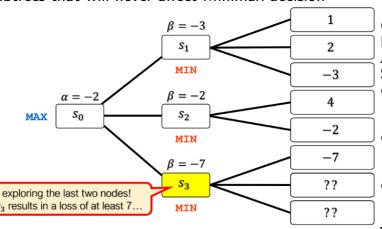
Game trees are massive, impossible to expand entire tree, must find ways to shrink search tree

#### $\alpha - \beta$ pruning

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

$\alpha$  bounds MAX's values,  $\beta$  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
```

```
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  $O(b^{3m/4})$

#### $\alpha - \beta$ 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).

#### Properties

- $O(2^n)$  time complexity
- $O(n)$  space complexity

#### Theorem proving methods

##### Validity & Satisfiability

- Validity
  - 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

##### 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

##### Inference rules

- And-Elimination:  $a \wedge b \models a, a \wedge b \models b$

- Modus Ponens:  $a \wedge (a \Rightarrow b) \models b$

- Logical equivalences:  $(a \vee b) \models \neg(\neg a \wedge \neg b)$

#### 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 knowledge base is domain-specific content

##### KB agent function

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

```
    TELL(KB, MAKE_PERCEP_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  $\alpha$  (a sentence) if  $\alpha$  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  $\alpha$

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,  $\beta = (q \text{ is odd}) \vee q = 2$

## 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  $\alpha$ , i.e. determine if  $KB \models \alpha$ .

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

#### Soundness & Completeness

- $KB \vdash_A \alpha$ 
  - Means "sentence  $\alpha$  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  $\alpha$ , KB entails  $\alpha$  if whenever KB true,  $\alpha$  true

function TT-ENTAILS?(KB,  $\alpha$ ) returns true or false

inputs: KB, the knowledge base, a sentence in propositional logic  
 $\alpha$ , the query, a sentence in propositional logic

symbols  $\leftarrow$  a list of the proposition symbols in KB and  $\alpha$

return TT-CHECK-ALL(KB,  $\alpha$ , symbols, {})

function TT-CHECK-ALL(KB,  $\alpha$ , symbols, model) returns true or false

if EMPTY?(symbols) then

  if PL-TRUE?(KB, model) then return PL-TRUE?( $\alpha$ , model)  
  else return true // when KB is false, always return true

else

$P \leftarrow \text{FIRST}(symbols)$

  rest  $\leftarrow \text{REST}(symbols)$

  return (TT-CHECK-ALL(KB,  $\alpha$ , rest, model)  $\cup \{P = \text{true}\}$ )

  and

  TT-CHECK-ALL(KB,  $\alpha$ , rest, model)  $\cup \{P = \text{false}\}$ )

#### Properties

- $O(2^n)$  time complexity

#### Tautology

$(KB \models \alpha) \iff (KB \models \neg \alpha)$  (deduction theorem)

#### Unsatisfiability

$(KB \models \alpha) \iff (KB \models \neg \alpha)$  is unsatisfiable

\* Definition of proof by contradiction

#### Inference rules

- And-Elimination:  $a \wedge b \models a, a \wedge b \models b$

- Modus Ponens:  $a \wedge (a \Rightarrow b) \models b$

- Logical equivalences:  $(a \vee b) \models \neg(\neg a \wedge \neg b)$