AI-ML Video Notes.

Rational : Maximally achieving pre-defined goals.
Being rational means maximising your expected utility.

Reflex Agent :

- Chooses action based on current percept (& maybe memory), also it doesn't consider the future conseqs of their actions.

Search Problems :

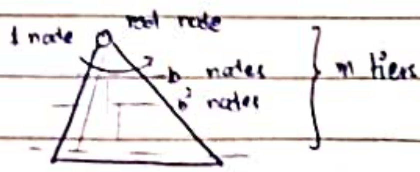
⇒ A search problem usually consists of : a state space & a successor function (with actions, costs), a start state & a goal state.

A solⁿ is essentially a sequence of steps which transforms the start state to a goal state.

State Space Graph : It is a mathematical repⁿ of a search problem, with nodes representing configurations. Arcs representing successors. Each state occurs only once! And building the entire graph is very expensive memory wise.

General Tree Search Algorithm :

```
function TREE-SEARCH(problem, strategy) returns a solⁿ, or a failure
    initialize the search tree using the initial state of the problem.
    loop do :
        if there are no candidates for expansion ⇒ return failure.
        choose a leaf node for expansion according to strategy.
        if node contains a goal state then return the corr. solⁿ
        else expand the node & add the resulting nodes to the search tree
    end loop
```



Properties : (for DFS)

(i) It'll stop at the left most solⁿ.
In the worst case, it'll explore the entire tree.

b : branch factor
m : max^m depth

(ii) How much space does the fringe take ?

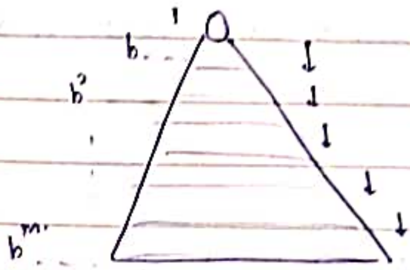
No. of nodes = $1 + b + \dots b^m$: $O(b^m)$

⇒ Only has siblings from on path to root, so $\underline{O(bm)}$

⇒ Is the algo complete? (or can we find a sol^n wherever it is in the tree)

Ans: No as m could be infinite

Breadth First Search:



⇒ It is gonna process all sol^{n}s above the shallowest sol^n.

⇒ Search time in the worst case: $\underline{O(b^m)}$

How much space does the fringe take?

⇒ Roughly the last tier so $\underline{O(b^s)}$

⇒ If m is finite then we can find a sol^n.

Uniform Cost Search: Special Case of A^* (Prove it)

Search Heuristics:

⇒ Heuristic is a function that maps states to real no.s & estimates how close we are ~~to~~ to the goal.

⇒ Designed for a particular search problem.

Greedy Search: (Best First Search)

Strategy: We expand a node that we think is the closest to the goal state.

Heuristic: We estimate the distance to the nearest goal for each state.

Combining UCS & Greedy:

Uniform Cost Search orders by the path cost or backward cost $g(n)$.

→ A^* Search orders by the sum: $f(n) = g(n) + h(n)$.

↳ We only stop when we dequeue a goal.

Admissibility:
They slow down the bad plans but never outweigh true costs.

A heuristic is admissible (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

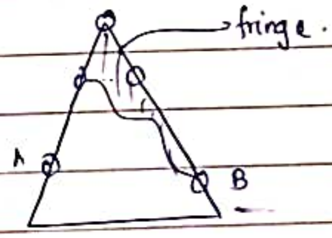
where $h^*(n)$ is the true cost to the nearest goal.

Optimality of A^* Search:

Assume that:

⇒ A is an optimal goal node & B is a suboptimal goal node.

⇒ h is admissible



Claim:

A will exit the fringe before

Proof:

⇒ Imagine B is on the fringe.

⇒ Now there will be some ancestor n of A that is on the fringe too (may even be A)

⇒ n will be expanded before B.

1. $f(n)$ is less than or equal to $f(A)$. → $f(n) = g(n) + h(n)$ and $f(n) \leq g(A)$. cuz of admissibility

2. $f(A) < f(B)$ → $g(A) < g(B)$, $f(A) < f(B)$ since A is goal.

⇒ n expands before B

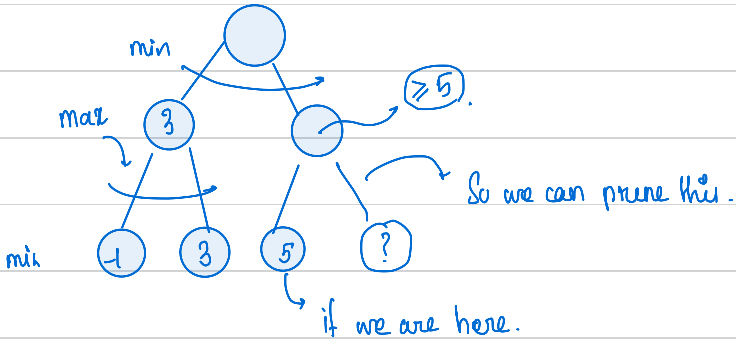
Minimax Algorithm:

Convention: Suppose that we're playing chess then white tries to maximize the score while black tries to minimize it. Leaf nodes ⇒ static evaluation of nodes.

```
minimax ( pos, depth , playerOne )
        if depth == 0 or gameOver in pos :        <------ This is for the leaf nodes.
              return static evaluation of pos

        if playerOne :
              maxEval = -∞
              for each child of pos :
                    eval = minimax( child, depth-1, False)
                    maxEval = max( eval, maxEval)
              return maxEval.

        else :
              minEval = +∞
              for each child of pos :
                    eval = minimax( child, depth-1, True)
                    minEval = min( eval, minEval)
              return minEval
```



maximising Player :

$\alpha = \max(\alpha, \text{eval})$

if $\alpha \geq \beta$

break

minimizing Player :

$\beta = \min(\beta, \text{eval})$

if $\alpha \geq \beta$

break

Constraint Satisfaction Problems : (Part - i)

Map Coloring : We want to color each of the countries but we don't want neighbouring countries to have the same color.

State : Its a black box

Goal test : It is a function that checks if a particular state is a goal state or not.

CSPs :

State → variables X_i with values from domain D.

Goal test → set of constraints specifying allowable combinations of values for subset of variables.

Map Coloring :

Each country → variables.

Domains : D = { color set }

Constraints : Adjacent regions should have different color.

↳ **Explicit** : Constraints in terms of domain

Implicit : Direct inequalities.

Sol^n : Assignment of variables that passes all constraints.

N-Queens :

Variables : Position of queen in each row Q_k

Domains: $\{1, 2, \dots n\} \to$ row numbers

Constraints:

$\forall i, j \quad (Q_i, Q_j) \Rightarrow$ non-threatening. (Implicit)

$(Q_1, Q_2) \in \{ \dots \}$ (Explicit)

Constraint Graphs:

Binary CSP: Each constraint relates at most 2 variables

Nodes are variables & edges are constraints.

Cryptarithmetic Puzzles:

$$\begin{array}{r} {\scriptstyle x_3 \; x_2 \; x_1} \\ TWO \\ +\ TWO \\ \hline FOUR \end{array}$$

Variables: $X_1, X_2, X_3, T, W, O, F, U, R$

Domain: $\{0, 1, \dots 9\}$

Constraints

↳ Graph won't be binary

Waltz Algorithm: (Example of CSP)

Used for the interpretation of line drawings of solid polyhedra as 3D objects.

Approach:

Each intersection is a variable.

Adjacent intersections impose constraints on each other.

Sol'n are physically realizable 3D interpretations.

Types of CSPs:

a) Discrete Variables

finite domain → $d \rightarrow O(d^n)$

eg: Boolean CSPs

infinite domains → integers, strings

eg: Job Scheduling

b) Continous Variables

eg: linear constraints solvable in polynomial time by LP methods.

Types of Constraints:

Unary Constraints: eg: $var \neq x$ ← value.

Binary Constraints: eg: $var_1 \neq var_2$.

Higher ordered: (involving 3 or more variables)

Preference ⇒ x is better than y.

Solving CSPs:

Backtracking Search: [Bad name] ← cuz we backtrack only we don't have more options !!

→ One variable at a time! (Only need to consider assignments to one variable at a time s.t that doesn't break the constraints)

→ Check constraints on the go! (Incremental goal test)

Pseudo code:

function Backtrack (csp) initially assignment is empty.

return recursive Search ({ }, csp)

function recursive-Search (asgn, csp)

if asgn is complete ⇒ passes all constraints.

return asgn

var ← select un-assigned-variable (variables [csp], asgn, csp)

for each value in Order-domain-values (var, asgn, csp) :

if value == consistent given Constraints [csp] :

add { var = value } to asgn

result ← recursive-Search (asgn, csp)

if result ≠ failure

return result

Normal backtracking with this check.

remove { var = value } from asgn

return failure.

cuz of this

Improving :

Ordering : Order of assignment

Filtering : Can we detect a failure early on. Ruling out suspects.

Structure : Can we exploit the problem structure.

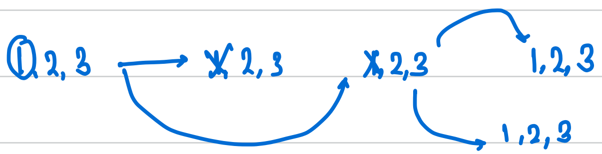
Filtering:

We keep a track of domains for unassigned variables & cross-off the bad options.

Forward Checking: (Its re-filtering of all variables after each assignment)

We cross-off values that could violate a constraint when added to existing assignment

eg:



(Map Coloring).

Essentially forward checking doesn't contribute to early detection of failures which is precisely what we want!! It doesn't care about relⁿ b/w assigned & unassigned variables

Constraint Propagation: Reason from constraint to constraint.

↳ its slow!!

Consistency of a Single Arc:

Defⁿ: An arc $X \rightarrow Y$ is consistent if and only if for every x in the tail, there is some y in the head which could be assigned without violating a constraint.

To make $X \rightarrow Y$ consistent, we need to delete smth from the tail.

A simple form of propagation makes sure that all arcs (or edges) are consistent.

In forward filtering, we are just checking the consistency of all nodes with the new assignment.

Note:

If X loses a value, then neighbours of X need to be re-checked!

Limitation: There can 1, 0 or multiple sol^ns

Enforcing Arc Consistency in a CSP:

$\boxed{T = O(n^2 d^3) \Rightarrow O(n^2 d^2) \text{ can be done}}$

function AC-3 (csp) returns CSP

q → queue of arcs (initially all arcs in CSPs)

while $q \neq \phi$ do:

$(x_i, x_j) \leftarrow$ remove-first (q)

if re-move-inconsistent-values(x_i, x_j) then:

for each x_u in Neighbours$[x_i]$ do:

add (x_i, x_u) to the q.

function re-move-Inconsistent-values (x_i, x_j)

removed ← false

for each x in Domain$[x_i]$

if no value y in Domain$[x_j]$ allows (x, y) to satisfy $x_i \leftrightarrow x_j$

remove x from Domain$[x_i]$

remove ← true

return removed

CSPs Part - II

Ordering:

What's a good place to start?

MRV - Minimum Remaining Variables: Choose the variable with the fewest legal values left in the domain post assignment.

Why min? ⇒ Most constrained variable!! (Hard ones first)

Choose a variable that doesn't cross-off too many things ⇒ Least constraining value.

Picking a value ⇒ choose the easy one.

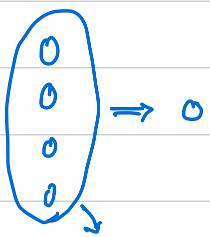
K- Consistency: (Doesn't violate constraints)

1- Consistency: (Node): Each single node's domain has a value which meets the node's unary constraints.

2- Consistency: (Arc): For each pair of nodes, any consistent assignment to one can be extended to the other. (delete from the tail after assignment)

⋮

k- consistency: For each k nodes, any consistent assignment to (k-1) can be extended to the k^{th} node.



can be extended to one more node.

legal move

Strong k-consistency : $\Rightarrow$ also $k-1$ $\Rightarrow$ $k-2$ ... $\Rightarrow$ 1-consistent

Claim : Strong n-consistency means we can solve without backtracking !

Why ?

Choose any assignment to a variable

Choose a new variable

By 2-consistent, the given choice is consistent with the first

Bottom to up.

⋮ ⋮

$k=3 \Rightarrow$ path consistency.

How to utilize the structure of problems to gain advantages ?

Dividing a given problem into smaller sub-problems.

eg: If originally we have n variables $\Rightarrow$ divided into sub-problems of only c variables

$T = O((n/c)\, d^c)$, linear in n.

Theorem: If consistent graph has no loops, then CSP can be solved in $O(nd^2)$

Compared to general case, $T = O(d^n)$

Tree-Structured CSP

Algo :

- Order : Choose a root, order the variables so that parents preceed children
  (Topological Sorting)
- Remove backward : for $i = n$ to 2 : RemoveInconsistent(Par(X_i), X_i)

- Assign Forward : For $i = 1$ to n : assign X_i consistently with Parent(X_i)

<u>Run-time</u> : $O(nd^2)$ $\longrightarrow$ Refer the book for the proof.

Adversarial Search.

All searches we've seen till now → Single Agent Trees

Leaves ⇒ values of the state

Value of a state is the best achievable outcome (utility) from that state.

$V(s) =$ known for terminal states

$V(s) = \max_{s' \in child(s)} V(s')$ for non-terminal states.

Minimax (Cuz it's like exhaustive DFS)

$T = O(b^m)$

$S = O(bm)$