ARTIFICIAL INTELLIGENCE & MACHINE LEARNING

What is Artificial Intelligence?

- Human-like
- Rational-like
- Thinking rationally
- Acting rationally
- Acting humanly
- Thinking humanly
- Thought processes
- Behaviour

The Turing test

natural language, knowledge representation, automated reasoning, machine learning, (vision, robotics) for full test

Thinking humanly

introspection, the general problem solver, cognitive sciences

Thinking rationally

Logic, problems: how to represent and reason in a domain

Acting rationally

agents: perceive & act

AI examples

1. common sense reasoning
   - tweety is a bird → it can fly
   - Yale shooting problem

smoke: fire ⇒ smoke

☆ Tweety is a bird.
All birds can fly.
Since Tweety is also a bird → it can also fly.
↓ exception
☆ Tweety is an ostrich.
ostriches cannot fly.
Since Tweety is an ostrich, it cannot fly.

Yale shooting problem
sequence of events.

Initial — Gun is loaded (0)
\ Alive (4)

Loaded (0) → Loaded (1)
Alive (4) → Alive (3)

↓

like a search problem { used typically in games. move from one state to another state till the goal is reached.

	Loaded	Alive
(many problem states)	0	4
	1	4
	1	3
	2	2
	:	

Final goal: to kill all
loaded → anything
alive → 0

② Update vs revise knowledge :

A (or) B → C

⇓

observe C = 0, vs Do C = 0

③ chaining theories of actions

Looks like P → Is P

Make looks like P → Looks like P

Make looks like P → Is P ??

eg
- Birds → fly
- Tweety → Bird
- Tweety → fly

* Garage door example : garage door not included
* planning benchmarks .
* 8 puzzle, 8 queen, Block world, Grid-type world
* Abduction : cambridge parking example.

PROGRAM : 1 ⇒ TIC TAC TOE

data structure approach

1	2	3
4	5	6
7	8	9

⇒ 3^9 combinations

[empty, X, O]

3^9 →

0
1
2
.
.
.

unnecessary space, looking at all combinations → a waste of time

Another DS approach

2 → blank   3 → x   5 → o

9x9 vector   <1, 2, ..., 9>

<2, 3, 5, ... >

↓

returns 5 if centre is blank

using prefix and suffix

go() function → either help produce a win (or) draw

state space tree

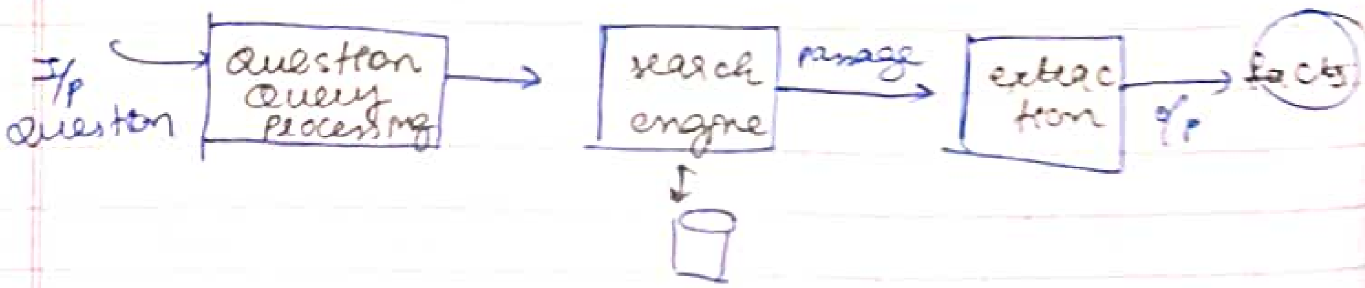
α – β

throw unnecessary nodes & keep only those which are useful

eg automatic draw / resign by the computer during chess ⇒ α–β algorithm

Q – A system

Question – Definitional ⤴ search, taking a passage & giving it user
– Factoral
↳ facts, intelligence needed



eg biography given

AGENTS (CHAPTER-2)

contents:

- Agents and environments
- Rationality
- ↳ PEAS (performance measure, environment, actuators, sensors)
- Environment types
- Agent types

Agents:

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

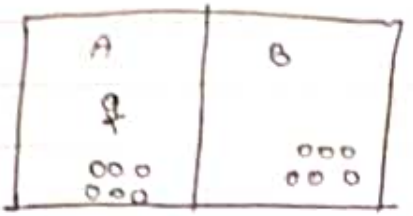
Human Agent
sensors: eyes, ears, other organs
Actuators: hands, legs, mouth, other body parts

Robotic agent
sensors: cameras & infrared range finders
actuators: various motors

- Agents and environments

Agent function, maps from percept histories to actions. $f : P^* \rightarrow A$
Agent program: runs on the physical architecture to produce f.
agent = architecture + program

VACUUM-CLEANER WORLD



Percepts : location and contents, eg [A, Dirty]
Action: left, right, suck, noop

Rational Agents

↓

An agent should strive to "do the right thing" based on what it can perceive and the actions it can perform. The right action is the one that will cause the agent to be most successful.

Performance measure. an objective criterion for success of an agent's behaviour.
eg: performance measure of a vacuum-cleaner agent could be:

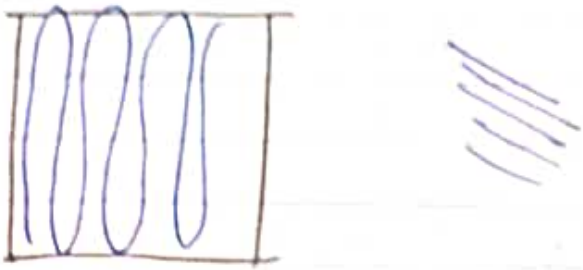
- amount of dirt cleaned up
- amount of time taken
- amount of electricity consumed
- amount of noise generated, etc

$P \rightarrow f(D, T, E, N, M)$
↳ (no. of moves)

work of 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.



What is involved in intelligence?

Intelligent agents

- Ability to interact with the real world
- knowledge representation, reasoning and planning
- learning and adaptation

usually challenged ATM
face identification + PIN

ML algos
- 80% → data for training
- 20% → data for testing (unseen data)

Implementing agents

- Table look-ups
- Autonomy
- structure of an agent
  agent = architecture + program

Autonomy
↳ even in a new environment the agent should be in a position to design an act

omniscience + learning + autonomy

omniscience
↳ actual outcome of an action that is carried out

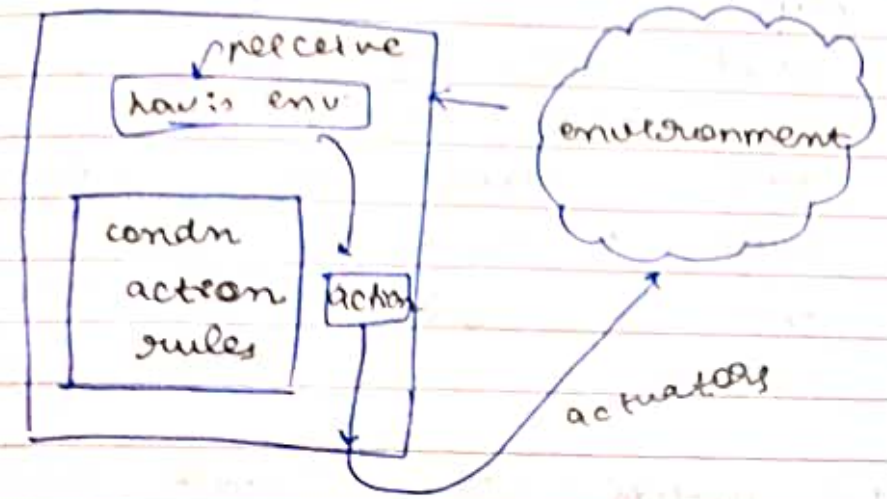
vacuum cleaner → 2 cameras
algorithm → camera captured info → convert into a rectangle of coordinates

Reflex vacuum Agent
function REFLEX-VACUUM-AGENT
([location, status]) returns an action
if status = Dirty then return Suck
else if location = A then return Right
else if location = B then return Left

table driven agent

function table-driven (percept) returns an action
percept sequence → empty
table → table of actions, indexed by percept sequence
append (percept) to the end of percepts
action ← LOOK-UP (percepts, table)

types of agents
- simple reflex agent
- model based reflex agent
- goal based agent
- utility based agent



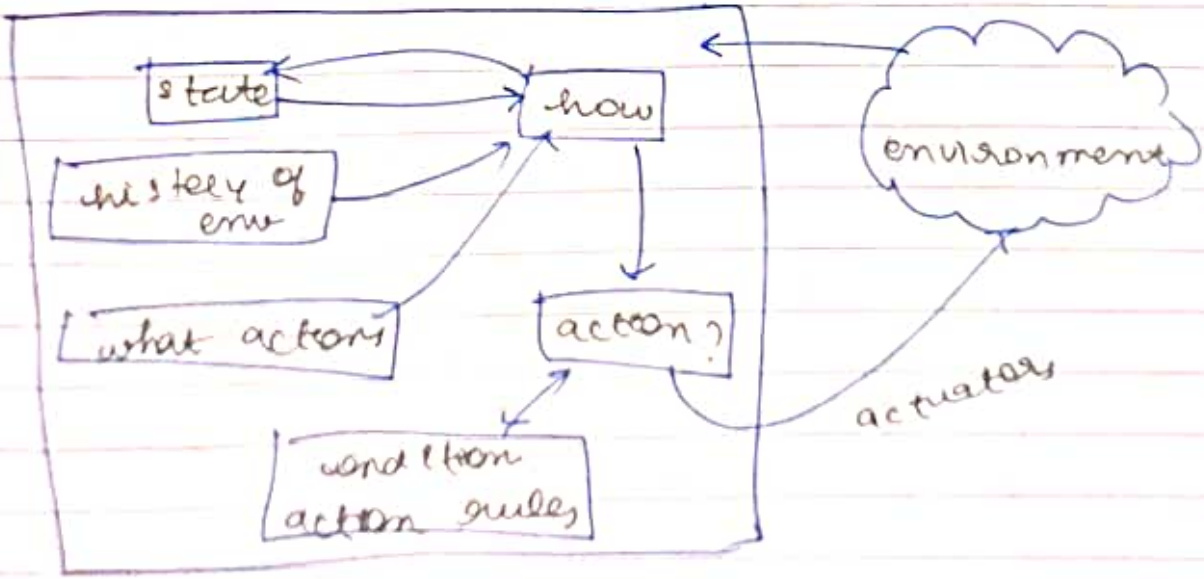
Automated driving car

red light → brake

(new vehicle, red) → break

(new vehicle, no red) → drive

Model-Based Reflex agent



↓

maintaining & updating the state as an when we try to capture

Google Maps

model - based reflex agent (percept),
returns action

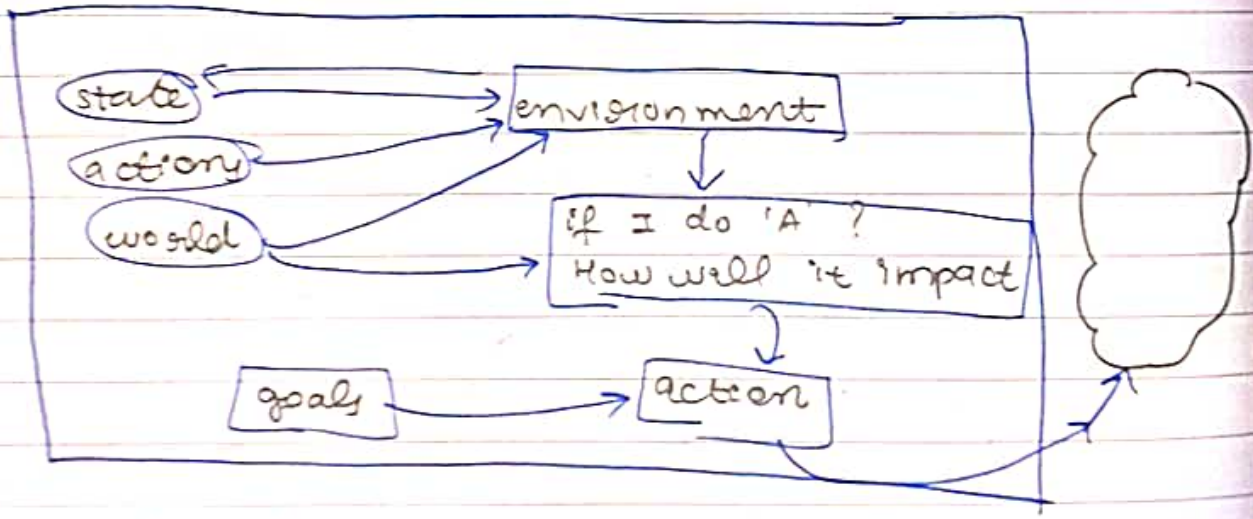
- state → current environment
- model → description of the state
- rules → conditions & actions
- actions → to be carried out

state ← update - state (state, action, percept, model)

Rule ← Rule - match (state, rules)

Action ← Rule action

Goal- based Agent

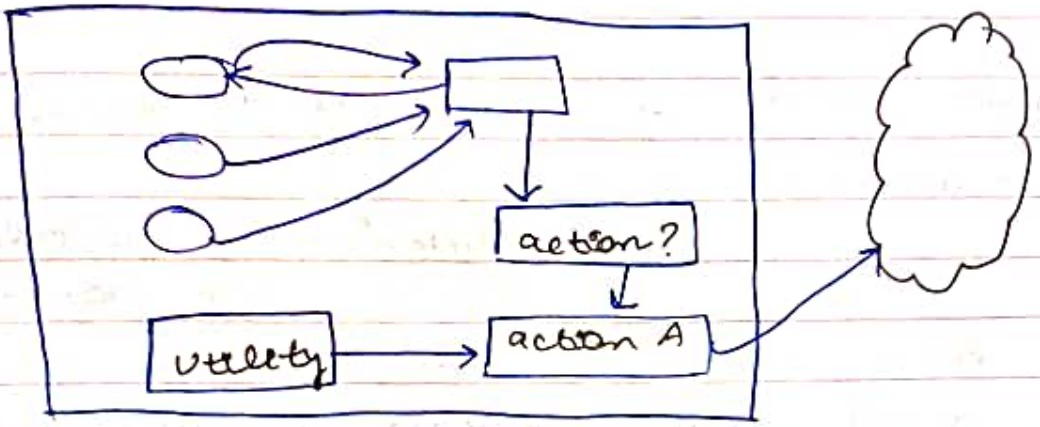


Goal- based agents are modifications to model - based agents which add goals to the parameters and deciding what actions to be carried out.

Automated driver : red, green, orange signals, no overtake, keep left (or right), etc.

Utility - Based Agent → user experience
* comfortable
* happy

goal / model / experience



goal - based agents & utility - based agents work by a lot of searching & planning.

TYPES OF ENVIRONMENT

① Fully observable vs Partially observable
↓ ↓
complete environment is available to us
only a portion of the environment is available to us.

Unobservable environment
↓
we are not able to get the environment at all

Google - maps : fully observable,
some information → return action
senses the complete environment at any given point of time

Autonomous vehicle : also fully observable if cameras are kept all over

vacuum cleaner : partially observable environment

② single - agent vs multi - agent Relevance

↳ crossword puzzle

autonomous vehicle
driver → one agent
other drivers → also agents with which driver should interact to make a decision

single - agent → one agent
more than one agent → multi - agent

③ Deterministic vs stochastic.

↓ next state depends upon the current state & action

↓ next state does not depend upon the current state & action

Stochastic → thought of as a partially observable environment

similarly Deterministic ↔ fully observable

Taxi - driving : multi - agent environment

deterministic

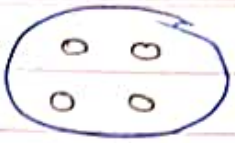
perfect driver
✗ (not always)

Stochastic

④ Episodic vs sequential

Episodic → atomic components → receives, action

sequential → individual components → eg watering plants → receives individually, action individually



current action depends upon the previous action

⑤ static vs Dynamic

static → stationary → eg garden, crossword puzzles

Dynamic → changes as agent is working on it → eg games, taxi driving

⑥ Discrete vs continuous

state of time → defined → state available only for fixed points of time → eg chess, gardening, games

environment factor → available for all points of time → eg taxi driving, google maps,

⑦ known vs Unknown

agent knows / doesn't know the environment

known → outcome of all actions are given (table-driven) → have full state space tree → eg tic tac toe

Unknown → outcome is handled dynamically → do not have full state space tree → encountering a particular combination of moves

PEAS

To design a rational agent, we must specify the task environment.

P → Performance measure
E → Environment
A → Actuators
S → sensors

eg taxi (automated):

P → safety, destination, profits, legality, comfort
E → US streets/freeways, traffic, pedestrians, weather
A → steering, accelerator, brake, horn, speaker/display
S → video, accelerometers, gauges, engine sensors, keyboard, GPS.

Environment types

	MD	solitaire	Tutor	Backgammon	Internet shopping	Taxi
observable	✗ partly	✓	✗ partially	✓	✗	✗ partly
deterministic	✗	✓	✗	✗	partly	✗
Episodic	✗	✗	✗	✗	✗	✗
static	✗	✓	✗	semi	semi	✗
discrete	✗	✓	✓	✓	✓	✗
single-agent	✓	✓	✗	✗	no except auctions	✗

the environment type largely determines the agent design.

The real world is: partially observable, stochastic, sequential, dynamic, continuous and multi-agent

MD → medical diagnosis

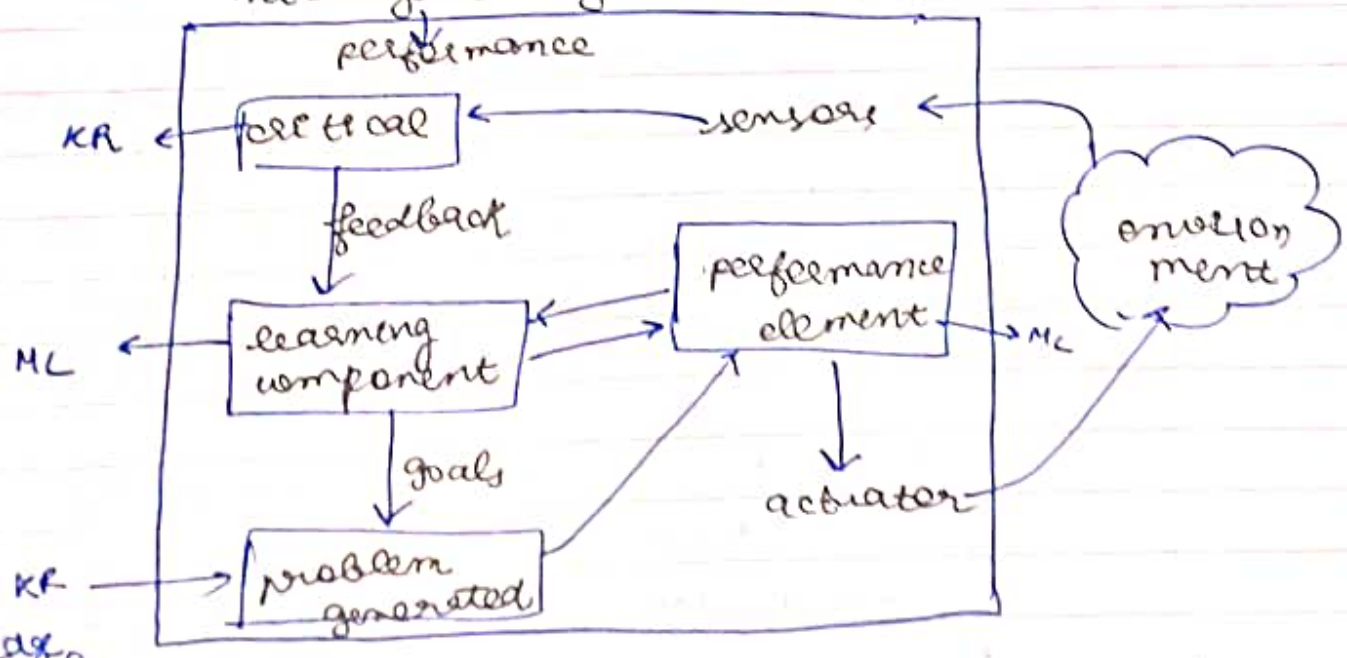
PEAS : English Tutor

P : student's performance ⟶ oral written
E : set of students, testing agency / test
A : exercises, suggestions, speech, corrections
S : keyboard only / oral english captile

PEAS : Medical Diagnosis

P : healthy patient, low cost
E : hospital, patient, staff, lab
A : diagnostic report, treatment, referral, Q/A
S : entry of symptoms, patient's answers

learning agent
↳ intelligent agent

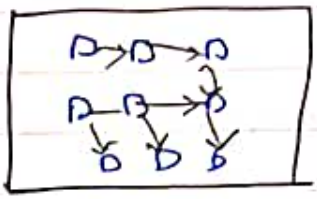


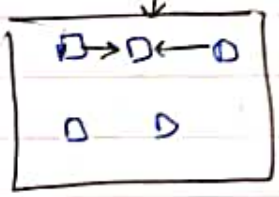
learning element → make modifications/ outcome better

performance element → actual action
ML → machine learning
KR → knowledge representation
KRR → knowledge reasoning

Environment components :

1. Atomic → B → C
2. Factored → split each state into a set of fixed variables / attributes
3. structured → solved separately, combine to get overall so





objects
attributes
relationships among objects

Examples :

atomic → game / Hidden Markov Models
HMM
Markov Decision Process
MDP

Factored → CSP constraint satisfaction problem
propositional logic /
bayesian networks, ML

structured → First ordered logic / NLP
natural Language understanding / KBU
knowledge based understanding

knowledge representation KR → ontology →
entity & their relationships

Planning Agents :-

- "what If"
- Decissions based on (hypothesized) consequence of actions
- Must have a model of how the world evolves in response to actions
- Must formulate a goal (test)
- consider how the world WOULD BE.

Optimal is complete planning
Planning is algorithm

search problems
problem solving agents

- Atomic representation
  states are considered as whole with no internal structure visible to the problem
- Goal formulation
  Based on the current situation and the agent's performance measure
- Problem formulation
  what actions and states to consider given a goal.
- Environment - observable, discrete, deterministic.
- search - process of looking for a sequence of actions
  Takes problem as input & returns solution in the form of an action
  Actions → carried out in the execution phase

- Initial state – starting point
- Actions – description of the possible actions available
- Applicable actions – From a state 's' what actions are possible
- Transition model – A description of what each action does
- Successor – state reachable from a given state
- State space – set of all states reachable from the initial state by any sequence of actions
- Graph – Nodes are states & links between the nodes are actions
- Path – sequence of states

transition model { 's' → action 'a' → 'p'
$\delta(s, a) = p$

- Goal test – test to verify whether a state is a goal
- Path cost – function to assign a numeric cost to each path
- solution – leads from start state to final state

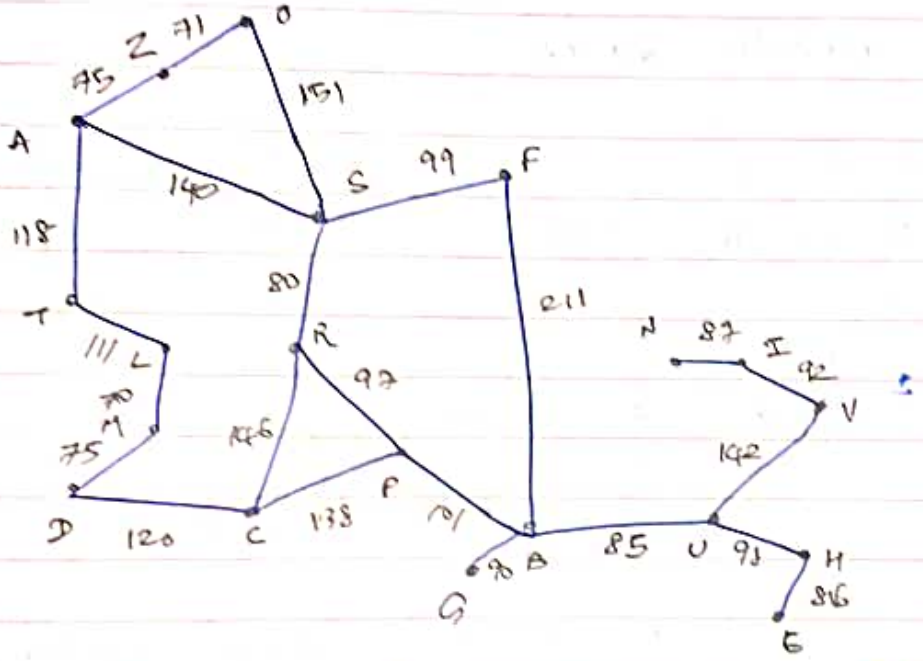
search problem

A search problem consists of
* state space
* successor function (with actions, costs)
* start state & goal test

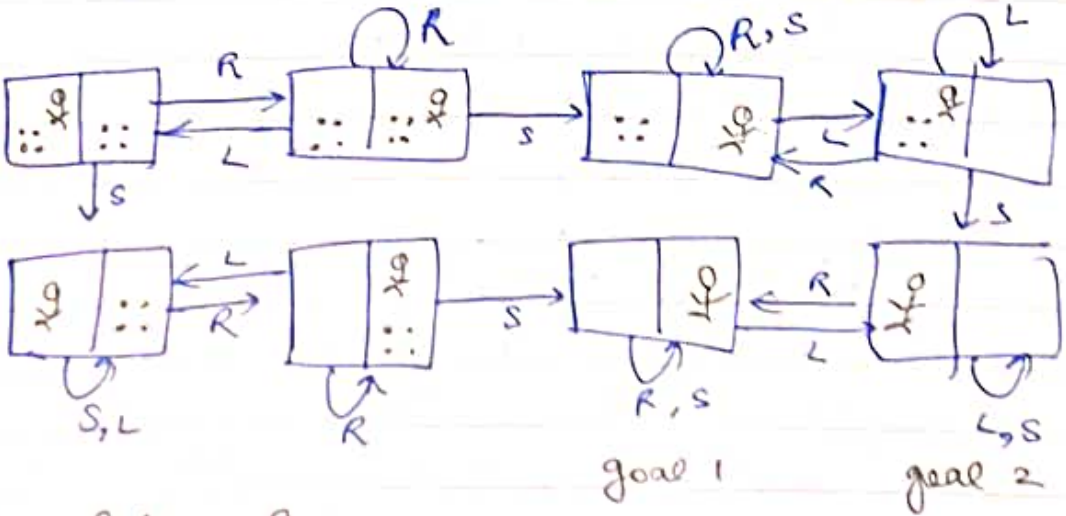
search problems are modeled

Example : Travelling in Romania

state space → cities
successor function → roads : go to an adjacent city with cost = distance
start state : A
Goal test : Is state == B ?
solution ?



state space for a vacuum world



{ L → left
R → right
S → suck }

n locations : $n \times 2^n$ states

Here $n=2$, 2×2^2 => agent & location states

Initial state :

Actions : L, R, S

Transition model :

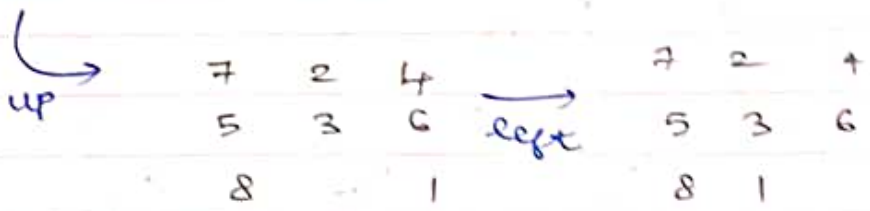
Goal Test : all squares are clean

path cost : each step cost 1 → no of steps

8 puzzle Game

7	2	4		1	2	3
5		6	⟹	4	5	6
8	3	1		7	8	

left, right, up, down movements



9 locations ⟹ (9×2^9 states)

8 Queens problem

place 8 queens on the chessboard such that no 2 queens attack each other

Route finding

states : locations

Initial state : user ?

Actions : mode of transport (run, walk, car, auto, etc)

Transition Model : city / Place → place

Goal test : reached final destination

Path cost : calories / petrol

State space :

world state
↳ includes every last detail of the environment

search state
↳ keeps only the details needed for planning (abstraction)

Problem, Pathing
states : (x, y) location
Actions : N S E W
successor : update location only
Goal test : is (x, y) = END

Problem : Eat-All-Dots
states : {(x, y), dot booleans}
Actions : N S E W
successor : update location & possibility a dot boolean

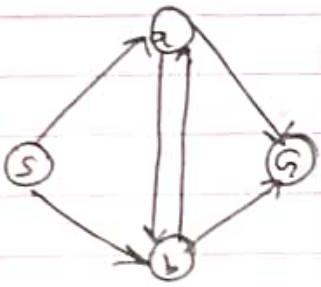
State space Graph
mathematical representation of a search problem
- nodes are abstracted world configurations
- arcs represent successors (action results)
- goal test is a set of goal nodes (may be only one)
- In a state space, each state occurs only once
- Full graph in memory → too big, but useful

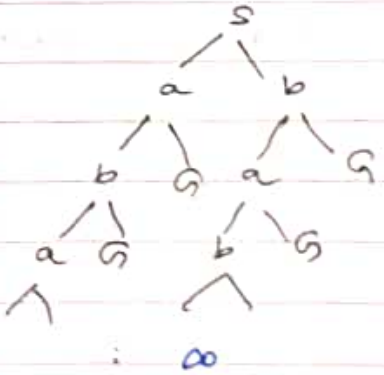
Search Tree

A "what if" tree of plans & outcomes
start state → root node
children → successors
nodes show states → correspond to PLANS to achieve those states
we never actually build the whole tree
we construct nodes & paths on demand & we construct as little as possible.

NODE (search tree) ⟺ PATH (state space graph)

State space Graphs as search Trees





(lots of repeated structures in the search tree)
↓
construct only these needed

searching with a search tree

search
- Expand out potential plans (tree nodes)
- Maintain a fringe of partial plans under consideration
- Try to expand as few tree nodes as possible

General tree search

function TREE-SEARCH (problem, strategy)
returns a solution, or failure

initialize the search tree using the initial state of problem

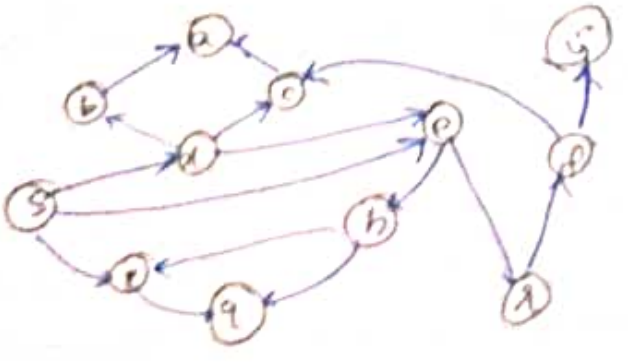
loop do
if there are no candidates for expansion
then return failure
choose a leaf node for expansion
according to strategy
if node contains a goal state then
return the corresponding solution
else expand the node and add the
resulting nodes to the search tree
end

Important ideas:
- fringe
- expansion
- exploration strategy

Main Q: which fringe nodes to explore?

Depth-First search

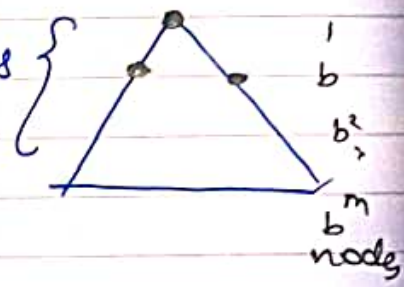
strategy: expand a deepest node first
Implementation: Fringe is a LIFO stack



search algorithm properties :

complete : Guaranteed to find a solution if one exists ?
optimal : Guaranteed to find the least cost path
Time complexity ?
space complexity ?

cartoon of search tree :



b → branching factor
m → maximum depth
solutions at various depth

no. of nodes in entire tree
$1 + b + b^2 + \dots + b^m = O(b^m)$

Depth-First search (DFS) properties

strategy : expand a deepest node first
what nodes DFS expand ?
* some left prefix of the tree.
* could process the whole tree
* If m is finite, takes time $O(b^m)$.

How much space does the fringe take ?
only has siblings on path to root so $O(b^m)$
Is it complete ?
m could be infinite, so only if we prevent cycles (more later)
Is it optimal ?
No, it finds the "leftmost" solution regardless of depth or cost.

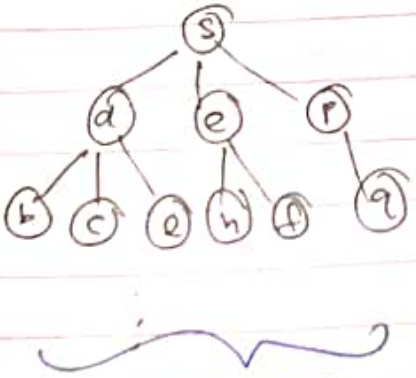
Uninformed search techniques
↓
ahead of time
goal state is not known, if the goal is hit while we keep expanding, it is done.

1. DFS (Depth First search)
2. BFS (Breadth First search)
3. Uniform cost search
4. Depth Limited search
5. Iterative Deep search
6. Bidirectional search

completeness ✓ optimality ✓
time complexity ✓ space complexity ✓

Depth Limited search
↳ no guaranteed solution [same as in DFS]
but it limits the branching factor to l depth, prefixed by the user.

Breadth First search
strategy: expand a shallowest node first
Implementation: fringe is a FIFO Queue.



search tees

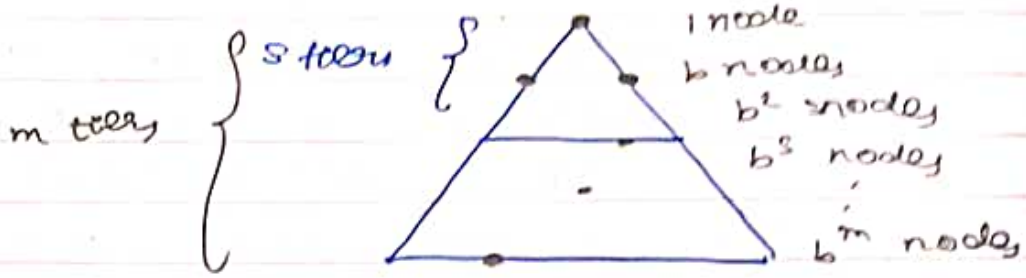
Queue
S d e p b c e h r q
:
[same graph as used in DFS]

S → e → r → f → G

Breadth First search Properties

what nodes does BFS expand?

- ☆ processes all nodes above shallowest solution
- ☆ let depth of shallowest solution be s
- ☆ search time takes $O(b^s)$



How much space does the fringe take?

Has roughly the last tier, so $O(b^s)$

Is it complete?

s must be finite if a solution exists

so yes!

Is it optimal?

only if costs are all 1 (more on costs later)

when will BFS outperform DFS?

Goal state with less connected graph ⟶ BFS outperforms DFS.

when will DFS outperform BFS?

complete graph & every node is connected to every other node in all possible scenarios.

↳ leftmost solution is obtained faster in DFS, outperforms BFS

Iterative Deepening

Idea: get DFS's space advantage with BFS's time / shallow - solution advantages

* Run a DFS with limit 1. If no soln:
  Increase limit: 0, 1, 2 limit → 2
* Run a DFS with depth limit 2. If no soln:
* Run a DFS with depth limit 3.

↓

Isn't that wastefully redundant?
Generally most work happens in the lowest level searched, so not so bad!

Cost-sensitive search

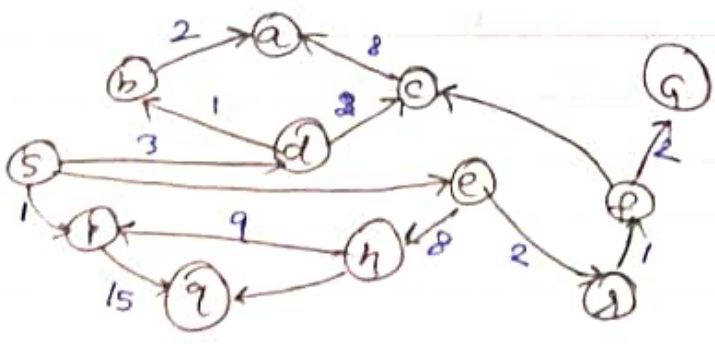
BFS finds the shortest path in terms of number of actions. It does not find the least cost path. Another similar algorithm helps find the same.

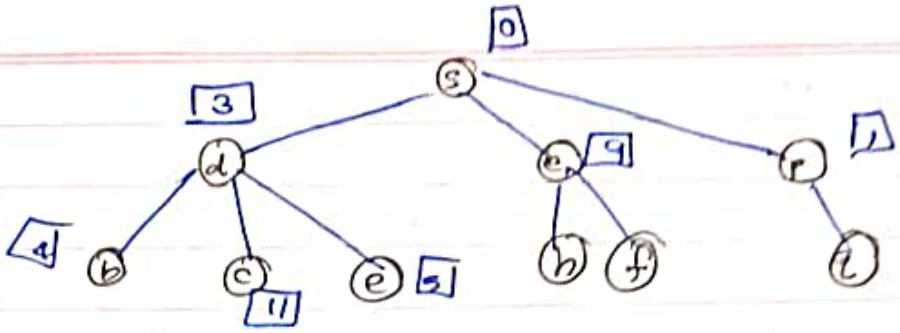
Uniform cost search

Is variation of BFS which gets the least possible cost.

Fringe ⇒ greedy in nature.
put in queue based on cost

Strategy: expand a cheapest node first
Fringe is a priority queue (priority: cumulative cost)





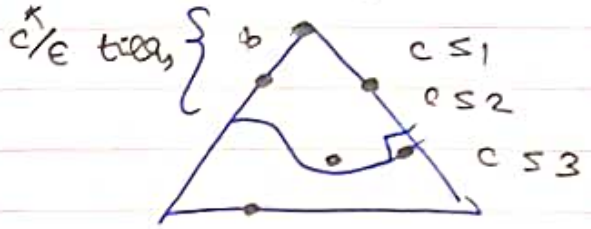
In priority queue : s | p | d | e | p | b | e | c
3 9 15

keep removing, adding and updating values from the queue.

uniform cost search (UCS) properties

what nodes does UCS expand?

* Processes all nodes with cost less than cheapest solution!
* If that solution costs C^* and arcs cost at least ϵ, then the "effective depth" is roughly C^*/ϵ.



How much space does the fringe take?

Has roughly the last tier, so $O(b^{C^*/\epsilon})$

Is it complete?

Assuming best solution has a finite cost and minimum arc cost is positive, yes!

Is it optimal?

Yes! (proof via A^*)

Uniform cost Issues

UCS explores increasing cost contours

The good: UCS is complete & optimal

The bad:

* explores options in every "direction"
* no information about goal location

The one Queue

All these search algorithms are the same except for fringe strategies.

* conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities).
* practically for DFS & BFS you can avoid the log(n) overhead from an actual priority queue by using stacks & queues.
* can even code one implementation that takes a variable queuing object

search and Models

search operates over models of the world

* the agent doesn't actually try all the plans out in the real world!
* planning is all "in simulation"
* your search is only good as your models

Example: Pancake problem

stack of pancakes, flip them such that smallest pancake is on top & all of them are in increasing order

How many flips are needed to finally achieve this

cost: no. of pancakes flipped

[sorting by prefix reversal]

Bidirectional search

any { BFS, DFS, UCS }

b^d

$b^{d/2} + b^{d/2} \neq b^d$

$O(b^{d/2})$ (reduced time complexity)

Informed search

↓

Goal is already known → more information is available about it than in the uninformed searches

↓

(Heuristic function)

Search Heuristic

general approach: best first search

which node to expand first?

Tree search / Graph search

$O \longrightarrow$ evaluation function

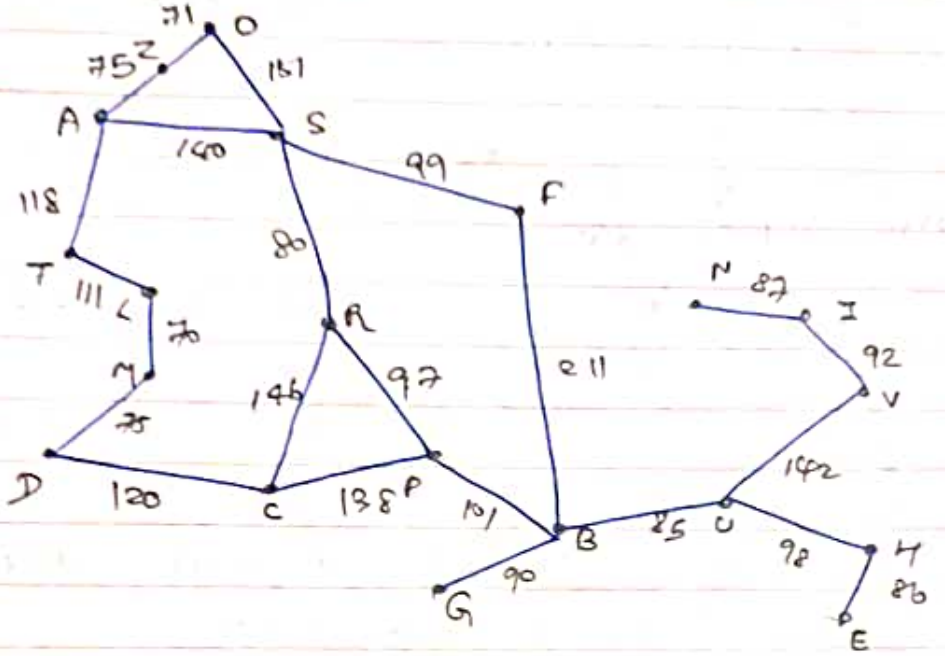
$f(n) \Rightarrow$ heuristic function

$h(n)$ = cost of lowest path from state node 's' to goal state 'g'

A heuristic is:

* A function that estimates how close a state is to a goal
* designed for a particular search problem Eg: manhattan distance, Euclidean distance for path

SLD : straight line distance
↳ eg for a heuristic function



straight line distances to B :

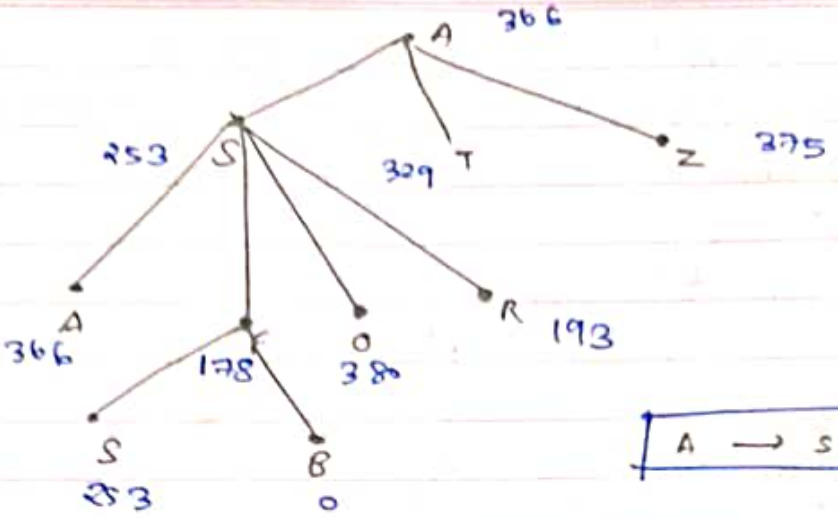
h(x)

A - 366	B - 0	C - 160	D - 242
E - 161	F - 178	G - 77	H - 151
I - 226	L - 244	M - 241	N - 234
O - 380	P - 98	R - 193	S - 253
T - 329	U - 80	V - 199	Z - 374

Lat, lon → Haversine distances → used to estimate SLD between 2 points

For pancake problem
Heuristic : the no of the largest pancake that is still out of place

Greedy search — Greedy best first search
SLD ⇒ h(x)
↙
SLD is used as a basis for expansion



(greedy approach ↓ SLD)

A → S → F → B

$O(b^m)$ worst case time & space complexity

↓

Greedy search

strategy : expand a node you think is closest to a goal state

Heuristic : estimate of distance to nearest goal for each state

A common case :

Best-first takes you straight to the (wrong) goal

worst case : like a badly guided DFS

A^* search

variations : Memory Bounded $A^* \Rightarrow MA^*$

simplified Memory Bounded $A^* \Rightarrow SMA^*$

(Greedy BFS)

In GBFS SLD from new state to goal state, but path from the source to new state was not considered

↓

A^* search considers this aspect also

$g(n) \rightarrow$ cost to reach a node

$h(n) \rightarrow$ cost to get from the node to goal

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

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

$g(n)$ → path cost from start to node

$h(n)$ → SLD ($f(n)$ → heuristic)

A^* ⇒ UCS ⇒ $(g+h)$ instead of g

- Admissibility
- consistency

} (optimality)

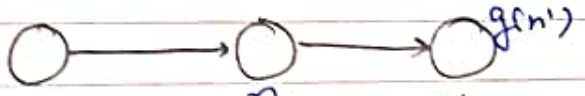
Admissibility

admissible heuristic

↓

not over-estimate the cost to reach the goal

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



consistency → stronger cond

↓

monotonicity

A^* → graph search

$h(n)$ is consistent if $\forall$ 'n', and n' is a successor of n, the estimated cost 'n' is not greater than the step cost of getting to n' + the goal from n'

Triangular inequality

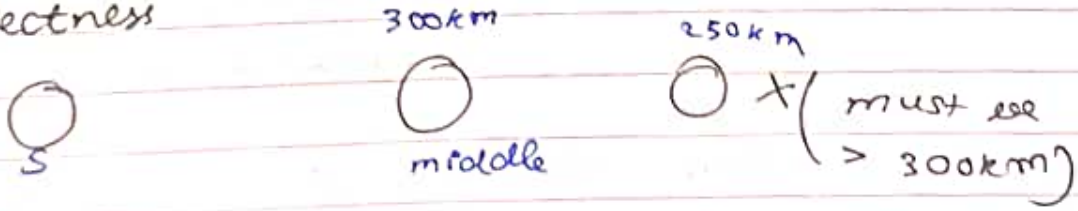
↓

$h(n) \leq c(n, a, n') + h(n')$

$h(n)$ → cost to reach the goal from n

$h(n')$ → cost to reach the goal from n'

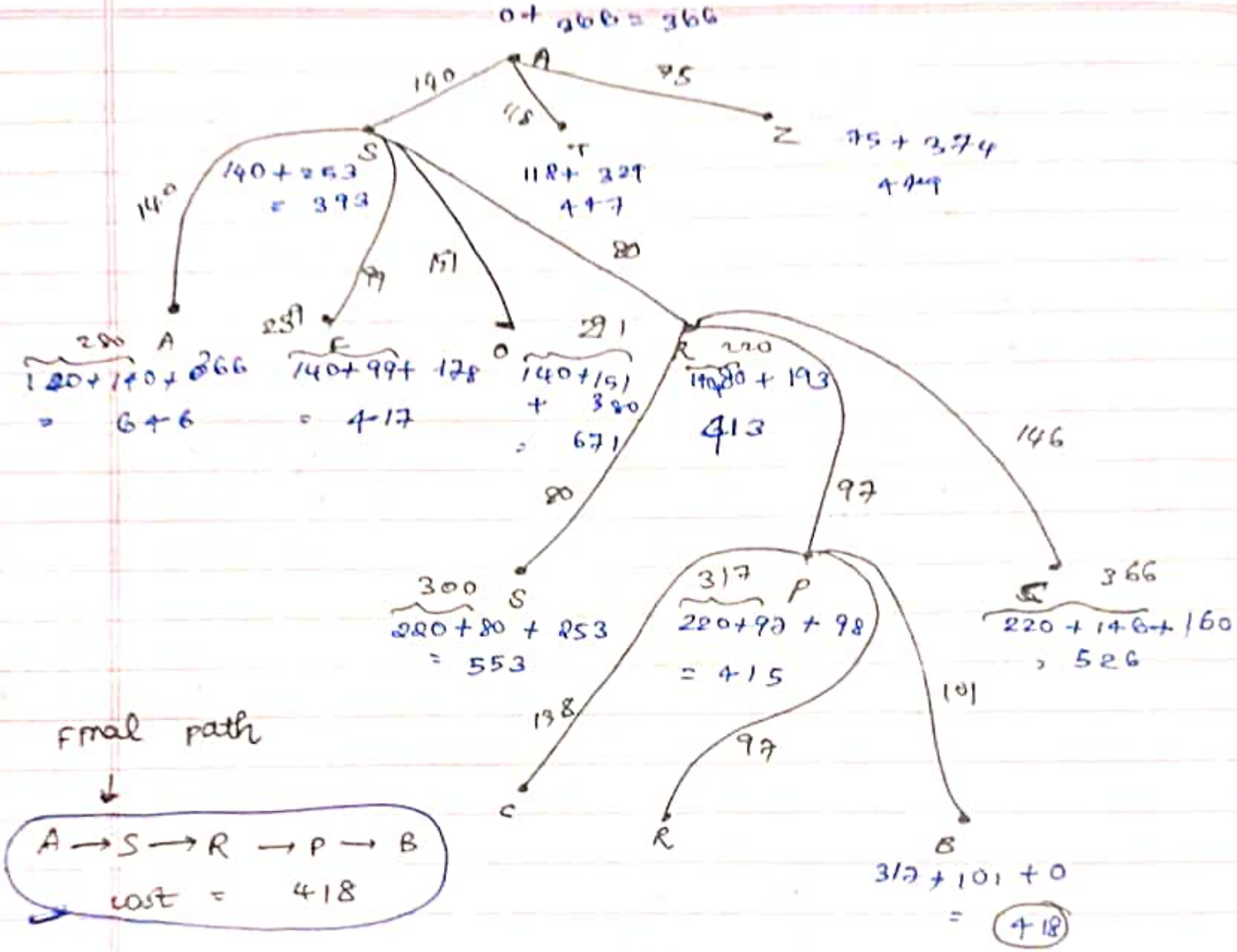
correctness



Every consistent heuristic is admissable.

↓

follows this triangular inequality

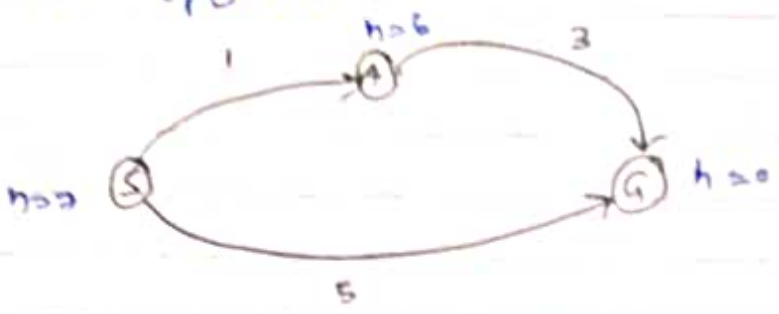


* Uniform cost orders by path cost or backward cost $g(n)$
* Greedy orders by goal proximity or forward cost $h(n)$
* A^* search orders by the sum $f(n) = g(n) + h(n)$

When should A^* terminate?

Should we stop when we enqueue a goal? No, only stop when we dequeue a goal.

Is A^* optimal ?



what went wrong ?

Actual bad good cost < estimated good goal cost

Idea : Admissibility

Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe.

Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs

Admissible Heuristics

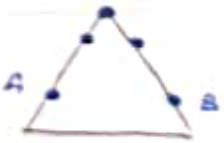
A heuristic h is admissible (optimistic) if

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

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

$$h(n) \leq c(n, a, n^*) + h(n^*)$$

optimality of A^* Tree search



Assume :

- A is an optimal goal node
- B is a suboptimal goal node
- h is admissible

claim : A will exit the fringe before B

optimality of A^* Tree search : Blocking

Proof

Imagine B is on the fringe

Some ancestor n of A is on the fringe too (maybe A)

Claim : n will be expanded before B

① $f(n)$ is less than equal to $f(A)$

$f(n) = g(n) + h(n)$ Defn of f-unit

$f(n) \leq g(A)$ admissibility of h

② $f(A)$ is less than $f(B)$

$g(A) < g(B)$ B is suboptimal

$f(A) < f(B)$ $h = 0$ at goal

③ n expands before B

All ancestors of A expand before B

A expands before B

Properties of A^*

UCS vs A^* contours

UCS expands equally in all directions

A^* expands mainly toward the goal, but does hedge its bets to ensure optimality

A* Applications

* video games
* pathing / routing problems
* resource planning problems
* robot motion planning
* language analysis
* machine translation
* speech recognition

Creating Admissible Heuristics

↳ (most work)

admissible heuristics are solutions to relaxed problems, where new actions are available

[sometimes inadmissible heuristics are useful too]

Example : 8 puzzle

what are the states ?
How many states ?
what are the actions ?
How many successors from the start state ?
what should the costs be ?

Heuristic : no. of tiles misplaced.

$h(start) = 8$ ⇒ relaxed problem heuristic

[what if we had an easier 8 puzzle where any tile could slide any direction at any time ignoring other tiles]

Total manhattan distance / city block distance
sum of all edge costs

using Actual cost as a heuristic ?
Admissible ?
save on nodes expanded ?
what wrong with it ?

A^* ⇒ trade off Between quality of estimate and work per node

Memory Bounded Heuristic search
Iterative deepening A^* (IDA*)

IDA* → ID

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

↓

reduces amount of memory needed to store the intermediate results

Recursive Best First search

RBFS → Linear search → A^* modification
similar to recursive depth first search
limit + alternate paths are also remembered

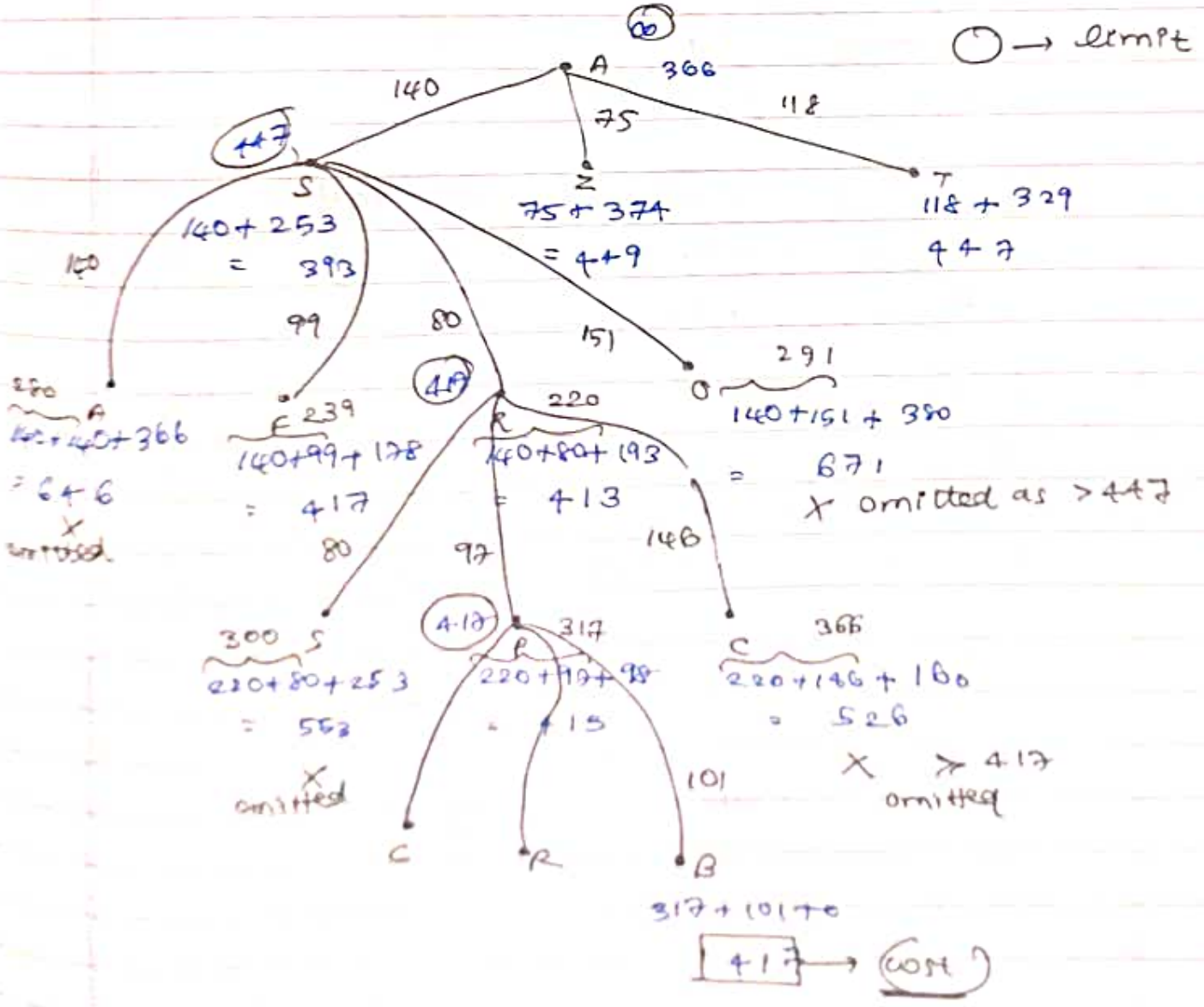
RBFS algorithm

(problem) returns a soln / failure
return RBFS (problem, make_node (problem, initial state), ∞)

RBFS (problem, node, f_limit)
returns soln / failure

if (problem goal_test (node.state)) return solution
successor [ ],
for each action in problem action (node.state)
add_child node (problem, node, action) in successor

if successor is empty return failure
∀ 's' in successor
s.f = max (s.g + s.h, node.f)
do
best = lowest f. value node in successor
if best.f > f_limit
return failure, best.f
alternate = second. lowest f value of successor
result, best.f = RBFS (problem, best, min (f_limit, alternate))
if result ≠ failure
return result as soln



Memory Bounded A^* $\Rightarrow$ MA^*

Simplified memory bounded A^* $\Rightarrow$ SMA^*

↓

cannot add a new node without dropping an earlier node

$f(x) + g(x)$

↙ RBFS ↓ SMA^* → A^* search

Semi-Lattice of heuristics

Trivial Heuristics, Dominance

↓

Dominance: $h_a \geq h_c$ if

$\forall n: h_a(n) \geq h_c(n)$

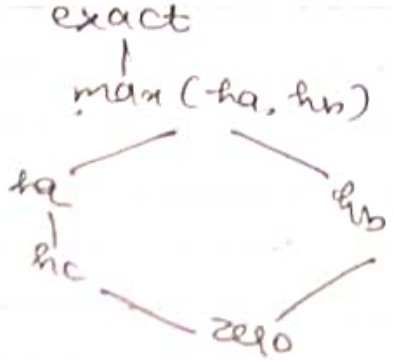
Heuristics form a semi-lattice:

max of admissible heuristics is admissible

$h(n) = \max[h_a(n), h_b(n)]$

Trivial heuristics

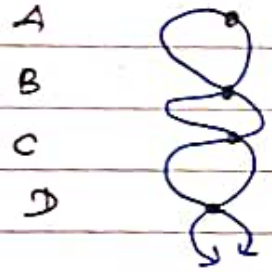
* Bottom of lattice is the zero heuristic (what does this give us)?
* Top of lattice is the exact heuristic



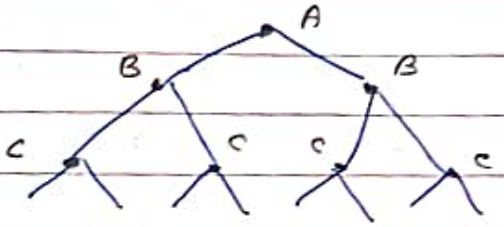
Graph search → loops

Tree search → failure to detect repeated states can cause exponentially more work.

State Graph



search Tree



In BFS, don't expand the repeating nodes

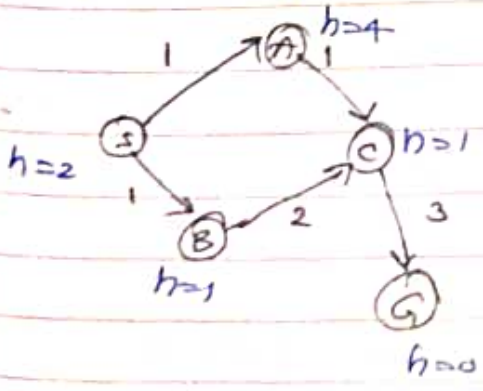
Idea: never expand a state twice.

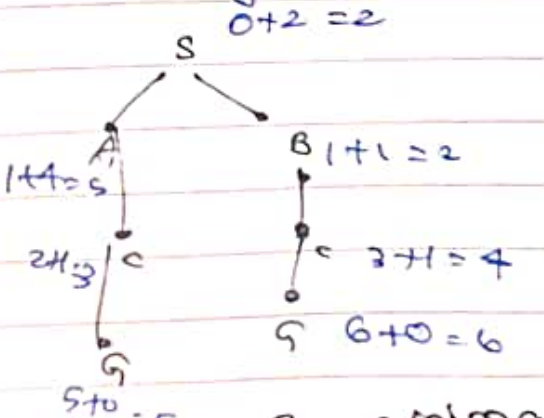
How to implement:

* Tree search + set of expanded states (closed set)
* Expand the search tree node by node but
* Before expanding a node, check to make sure its state has never been expanded before
* If not new, skip it, if new add to closed set.

Important: store the closed set as a set, not as a list

A* Graph search gone wrong?





A* → checks local maxima & minima

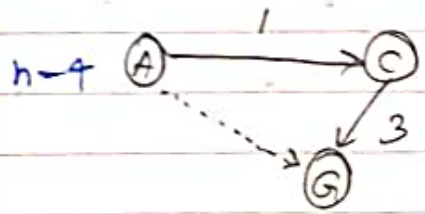
But we need global maxima & minima

consistency of Heuristics

Main idea : estimated heuristic costs ≤ actual costs

Admissability : heuristic cost ≤ actual cost to goal

h(A) ≤ actual cost from A to G



consistency heuristic arc cost ≤ actual cost for each arc

h(A) - h(C) ≤ cost(A to C)

consequences of consistency

The f value along a path never decreases

h(A) ≤ cost(A to C) + h(C)

A* graph search is optimal

optimality of A* Graph search

sketch : consider what A* does with a consistent heuristic

Fact 1 : In tree search, A* expands nodes in increasing total f value (f-contours)

Fact-2 : For every state s, nodes that reach s optimally are expanded before nodes that reach s sub-optimally

Result : A* graph search is optimal

optimality

Tree search:

* A^* is optimal if heuristic is admissible
* UCS is a special case ($h = 0$)

Graph search:

* A^* is optimal if heuristic is consistent
* UCS is optimal [$h = 0$ is consistent]

consistency implies admissibility.

In general, most natural admissible heuristics tend to be consistent especially if from relaxed problems.

A^* summary

* A^* uses both backward costs and estimates of forward costs.
* A^* is optimal with admissible / consistent heuristics.
* Heuristic design is key: often use relaxed problems

optimality of A^* Graph search

↓

complete mathematical proof during next course: AI and applications

All above algorithms → Local search algorithms

Local search Algorithms

1. Hill climbing
2. simulated Annealing
3. Local beam search
4. Genetic algorithm search

↓
(greedy approach)
↳ current node [only local optimum solns, not global]

Hill climbing search
↓
always keep climbing up-hill

function HC (problem)
current ← make_node (problem, initial state)
do
neighbour ← a high valued successor of current
if neighbour.value ≤ current value
return current.state
current ← neighbour

Heuristic :

local maxima → peak

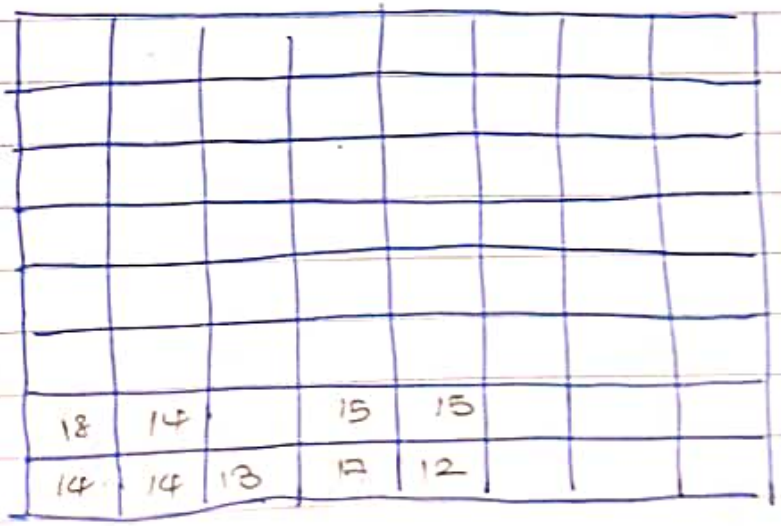
Ridge : result of so many local maxima available close to each other.

Plateau : flat area of state space
↳ flat local maxima / shoulder.
[no hill climbing → try to avoid plateau as much as possible]

8 Queen's problem

8 Queen move : 8 + 8 = 16

Taking heuristic h = 17 → not form hill climbing



placed queen → only go in diagonal direction
else → keep going sideways

move left, move right, move diagonal, place queen } actions possible

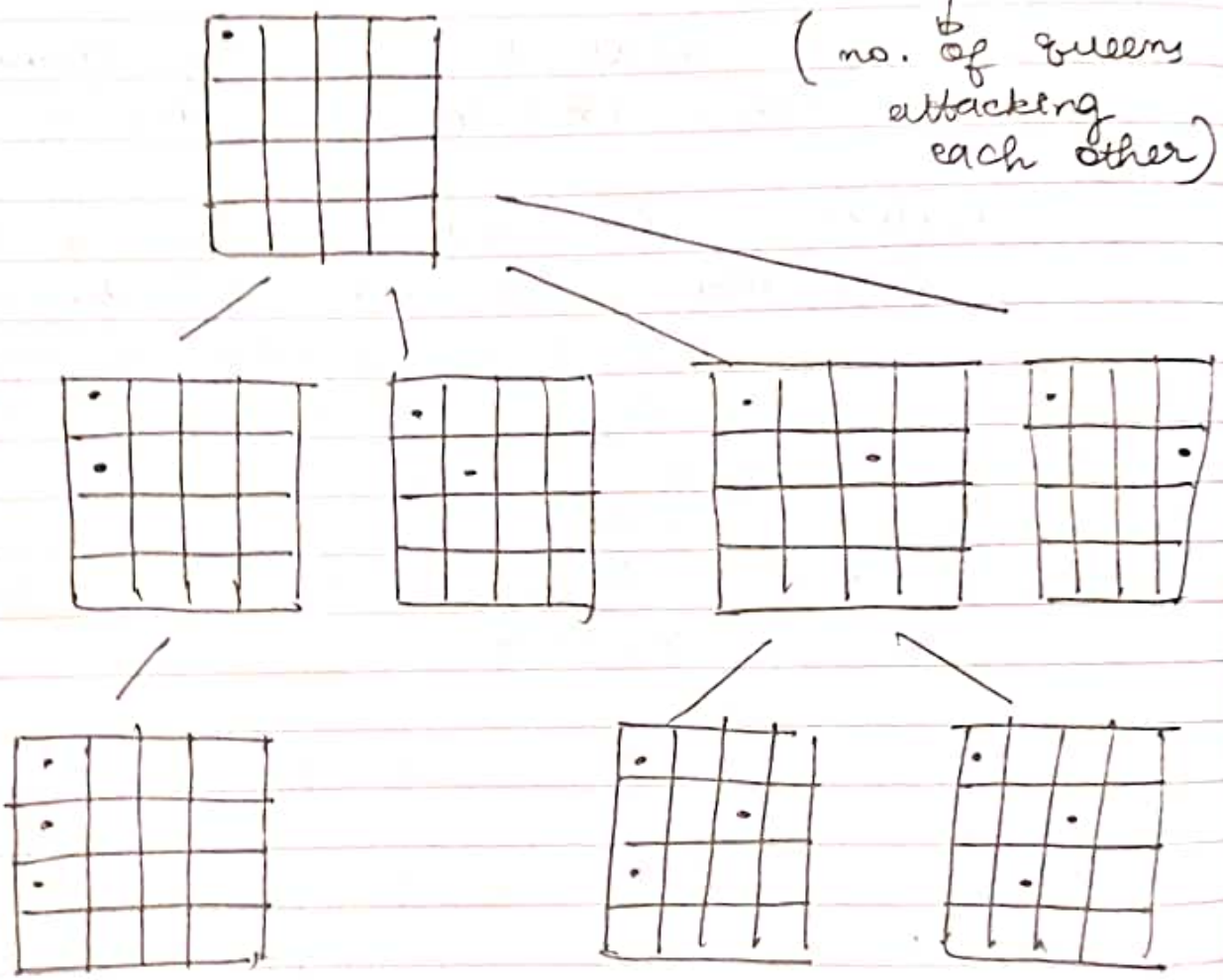
↓

(finally place 8 queens)

state space formulation

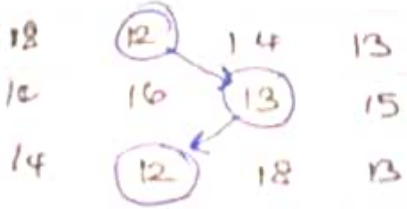
4 Queen Problem

heuristic function
(no. of queens attacking each other)



travel in the direction of least value of heuristic function

Drawback : too much memory, some states can be obviously left out & not expanded



sideways ridges } problems

More search Methods

* Local search → (4) types
* Local search in continuous spaces
* searching with non-deterministic actions (decision trees)
* online search (agent is executing actions)

Local search Algorithms and optimization Problems

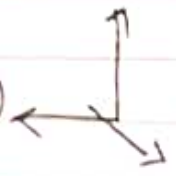
↓

- complete state formulation
  for example, for 8 queens problem, all 8 queens are on the board and need to be moved around to get to a goal state
- Equivalent to optimization problems often found in science and engineering.
- start somewhere and try to get to the solution from there
- local search around the current state to decide where to go next

Pose Estimation Example

Given a geometric model of a 3D object and a 2D image of the object determine the position and orientation of the object wrt the camera that snapped the image.

state (x, y, z, x-angle, y-angle, z-angle)



Hill climbing [from PPT]

Goal: optimizing an objective function
does not require differentiable functions.
can be applied to "goal" predicate type of problems.
BSAT with objective function number of clauses satisfied.
Intuition: always move to a better state

some Hill climbing Algo's

* start = random state or special state
* until (no improvement)
  - ⇒ steepest ascent: find best successor
  - ⇒ (OR) greedy: select first improving successor
  - ⇒ go to that successor
* Repeat the above process some number of times (restarts).
* can be done with partial solutions or full solutions

Implicit in this scheme is the notion of a neighbourhood that in some way preserves the cost behaviour of the solution space.

↓

Thinking about the TSP problem again...
If a current tour is present what would a neighbouring tour look like?

↓

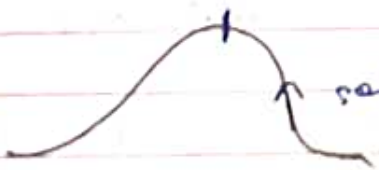
This is the way of asking for a successor function.

* The successor function is where the intelligence lies in hill climbing search
* It has to be conservative enough to preserve significant "good" portions of the current solution.
* and liberal enough to allow the state space to be traversed without degenerating into a random walk.

Hill climbing Algorithm

* In best first, replace storage by a single node
* works if single hill
* use restarts if multiple hills
* Problems :
  * finds local maximum not global
  * plateaux: large flat regions (happens in 8 8 7)
  * ridges : fast up ridge, slow on ridge
* not complete, not optimal
* no memory problems

variation of hill climbing search



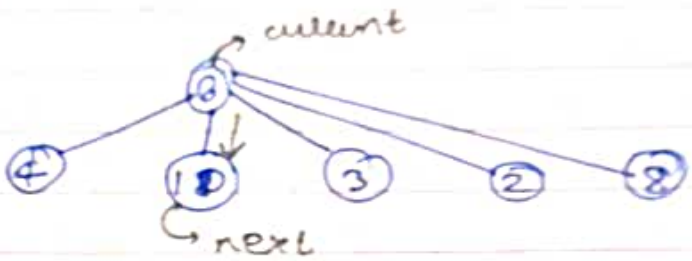
"gradient ascent"
[here solutions are max not min]

often used for numerical optimization problem

AI Hill climbing

steepest - ascent Hill climbing
current ← start node
loop do
* neighbour ← a highest valued successor of current

* if neighbour. value ≤ current. value
then return current. state
* current ← neighbour
end loop



if current = 12, again expand from
to (next, lowest/highest) / not possible /
change heuristic function.

Local maxima
Plateau
Diagonal ridge

↓
what is it sensitive to? does it
have any advantages?

Local Maxima (Minima)

→ * Hill climbing is subject to getting stuck
in a variety of local conditions
* Two solutions.
↓
⇒ stochastic Hill climbing → chooses random
from among the uphill moves
⇒ first choice hill climbing → similar to
stochastic but creates successors at
random
⇒ Random restart hill climbing →

randomly generates initial states and keep climbing high → (8 Queens Problem)

* simulated annealing

Random Restart Hill climbing

* generate a random start state
* run a hill climbing & store answer
* iterate, keeping the current best answer as you go
* stop when goal is reached.

simulated Annealing

* variant of hill climbing (so up is good)
* tries to explore enough of the search space early on, so that the final solution is less sensitive to the start state.
* May make some downhill moves before finding a good way to move uphill.

local maxima → global maxima
local minima → global minima

* comes from the physical process of annealing in which substances are raised to high energy levels (melted) and then cooled to solid state.

* the probability of moving to a higher energy state instead of lower is
$p = e^{-\Delta E / kT}$
where ΔE is the positive change in energy level, T is the temperature, and k is Boltzmann's constant.

* like hill climbing, but probabilistically allows down moves, controlled by current temperature & how bad move is

* let $t[1], t[2] \ldots$ be a temperature schedule usually $t[1]$ is high, $t[k] = 0.9\, t[k-1]$

* let E be quality measure of state.
Goal: maximize E.

simulated Annealing algorithm

* current = random state, $k = 1$
* If $T(k) = 0$, stop
* next = random next state.
* If next is better than start, move there
* If next is worse:
  ⇒ let delta = $E(\text{next}) - E(\text{current})$
  ⇒ Move to next with probability $e^{\Delta / T(k)}$
* $k = k + 1$

simulated Annealing Discussion

* no guarantees
* when T is large, $e^{\Delta/T} \to e^0$ (or) 1.
so for large T, go anywhere.

* when T is small, $e^{\Delta/T} \rightarrow e^{-\infty}$ (or) 0. Avoid most bad moves
* when T is zero, do simple hill climbing
* Execution time depends on schedule, memory use is trivial.

* At the beginning, the temperature is high.
* As the temperature becomes lower
  $kT \downarrow \quad \Delta E/kT \uparrow \quad -\Delta E/kT \downarrow \quad e^{-\Delta E/kT} \downarrow$
* As the process continues, the probability of a downhill move gets smaller and smaller.

* ΔE → represents the change in the value of the objective function
* since the physical relationships no longer apply, drop k. so $p = e^{-\Delta E/T}$.
* we need an annealing schedule, which is a sequence of values of T:
  $T_0, T_1, T_2, \ldots$

Simulated Annealing Algorithm

* current ← start node
* for each T on the schedule (need a schedule)
  ⇒ next ← randomly selected successor of current
  ⇒ evaluate next, if it is a goal return it
  ⇒ ΔE ← next.value − current.value
  ⇒ if $\Delta E > 0$ (already neglected)
  then current ← next (better than current)
  else current ← next with probability $e^{-\Delta E/kT}$

Function

function SIMULATED_ANNEALING (problem, schedule) return a solution state
Input : problem, a problem
schedule, a mapping from time to temperature
Local variables : current, a node.
next, a node
T, a "temperature" controlling the probability of downward steps

current ← MAKE-NODE (INITIAL-STATE [problem])
for t ← 1 to ∞ do
T ← schedule (t)
if T=0 then return current
next ← a randomly selected successor of current
ΔE ← VALUE [next] - VALUE [current]
if ΔE > 0 then current ← next
else current ← next only with probability $e^{\Delta E/T}$

Properties of simulated annealing search
↓
one can prove. If T decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1.

widely used in VLSI layout, airline scheduling, etc

n → no of flights on runway
- no of flights ready to take off

Probabilistic selection:
select next with probability p
0 ★ p 1
random number
Generate a random number
If r ≤ p select next

Simulated Annealing properties
↓
At a fixed "temperature" T, state occupation probability reaches the Boltzmann distribution,
$P(x) = \alpha e^{E(x)/kT}$
If T is decreased slowly enough (very slowly), the procedure will reach the best state -
slowly enough has proven too slow for some researchers who have developed alternate schedules.

Simulated Annealing schedule
↓
Acceptance criterion & cooling schedule

if (delta ≥ 0) accept
else if (random < $e^{\delta/T}$) accept
else reject
[0 ≤ random ≤ 1]

(Aim: global maxima)

simulated Annealing Applications

Basic Problems

- Travelling salesman
- Graph partitioning
- Matching problems
- Graph colouring
- scheduling

Engineering

- VLSI design
  - ⇒ placement
  - ⇒ routing
  - ⇒ array logic minimization
  - ⇒ layout
- Facilities layout
- Image processing
- code design in information theory

Beam

- mix of hill climbing & best first
- storage is a cache of best k states
- solves storage problem but (not optimal and not complete

k directions

k - generated @ each step → successors

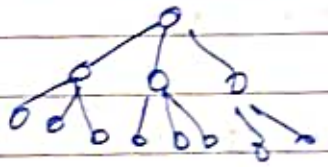
check best → moves in that direction

(or) any equivalent amount

Local Beam search

keeps more previous states in memory
simulated annealing just kept one previous state in memory.
This state keeps k states in memory.

→ Randomly generates k initial states.
→ If any state is a goal terminate
→ else generate all successors and select the best k
→ repeat



k → states gets concentrated

↓

Stochastic local beam search

Local (Iterative) Improving

Initial state = full candidate solution
Greedy hill climbing
if up, do it
if flat, probabilistically decide to accept move
if down, don't do it
we are gradually expanding the possible moves

↓

Local improving performance.

- solves 10^6 queen problem quickly
- useful for scheduling
- useful for BSAT
- solves (sometimes) large problems
- more time, better answer
- no memory problems
- no guarantees of anything

Genetic Algorithm

* weakly analogous to evolution
* no theoretical guarantees
* applies to nearly any problem

* population = set of individuals
* fitness function on individuals
* Mutation operator : new individual from old one
* cross over : new individuals from parents

GA Algorithm (a version)

* population = random set of n individuals
* probabilistically choose n pairs of individuals to mate
* probabilistically choose n descendants for next generation (may include parents or not)
* probability depends on fitness function
* in simulated annealing
* How well it works ⇒ ?

scores to probabilities

suppose the scores of n individuals are
$a[1], a[2], \ldots, a[n]$
The probability of choosing the j^{th} individual :

$$\text{prob} : \frac{a_j}{a_1 + a_2 + \ldots + a_n}$$

GA Example : Boolean satisfiability

Individual → bindings for variable
Mutation → flip a variable
Cross-over → for 2 parents, randomly positions from 1 parent. For one son take those binding and use other parent for other
Fitness → no. of clauses solved.

GA Example : N-Queen's problem

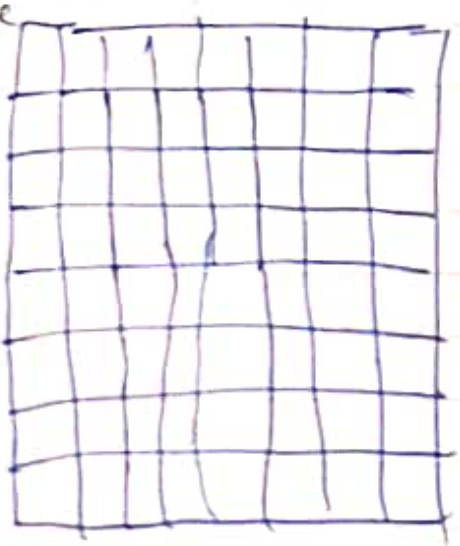
eg
3 2 5 0 1 4 7 6 Individual ← array indicating column where
Mating ← cross over ith queen is assigned
Fitness (minimize) ← no. of constraint violations

GA function optimization (Ctd).

* Let $f(x,y)$ be the function to optimize.
* Domain for x and y is a real number between 0 and 10.
* Say the hidden function is

$f(x,y) = 2$ if $x > 9$ & $y > 9$
$f(x,y) = 1$ if $x > 9$ or $y > 9$
$f(x,y) = 0$ otherwise

(cross over)
2 4 7 4 8 5 5 2
3 2 7 5 2 4 1 1
2 4 4 1 5 1 2 4
3 2 5 4 3 2 1 3



GA works well here:

* Individual = point = (x, y)
* mating: something from each so:
  mate $(\{x, y\}, \{x', y'\})$ is $\{x, y\}$ & $\{x', y'\}$
  * no mutation
  * hill-climbing does poorly, GA does well
  * this example generalizes function with large arity.

3 2 7 5 2 4 1 1 ✗ 3 2 7 4 & 5 5 2
2 4 7 4 & 5 5 2

GA discussion

* reported to work well on some problems
* typically not compared with other approaches eg hill climbing with restarts.
* opinion: works if the "mating" operation captures good substructures.

Genetic Algorithms

* start with random population of states
  * representation serialized (string of characters or bits)
  * states are ranked with fitness function
* produce new generation
  select random pair(s) using probability proportional to fitness
  * Randomly choose "crossover point"
    offspring mix halves
  * Randomly mutate one

Genetic Algorithm

* Given: population P and fitness function f
* repeat
  * new P ← empty set
  * for i=1 to size(P)
    * x ← random selection (P, f)
    * y ← random selection (P, f)
    * child ← reproduce (x, y)
    * if (small random probability) then
      * child ← mutate (child)
    * add child to new P
  * P ← new P
* until some individual is fit enough or enough time has elapsed.
* return the best individual in P according to f.

Using Genetic Algorithms

2 important aspects to use them

1. How to encode your real life problem
2. Choice of fitness function

Research example

We want to generate a new operator for finding interesting points on images.

The operator will be some function of a set of values $v_1, v_2, \ldots v_k$.

Idea: weighted sums, weighted min, weighted max;

$a_1, \ldots a_k$ $b_1, \ldots b_k$ $c_1 \ldots c_k$

Local search in continuous space

Given a continuous state space

$$S = \{ (x_1, x_2, \ldots, x_N) \mid x_i \in R \}$$

Given a continuous objective function

$$f(x_1, x_2, \ldots, x_N)$$

The gradient of the objective function is a vector $\nabla f = \partial f/\partial x_1, \partial f/\partial x_2, \ldots, \partial f/\partial x_n$.

The gradient gives the magnitude and direction of the steepest slope at a point.

To find a maximum the basic idea is to set $\nabla f = 0$

Then updating the current state becomes $x \leftarrow x + \alpha \nabla f(x)$ where α is a small constant.

Refer ⇒ computer vision pose Estimation Example

Searching with non-deterministic actions

↓

vacuum world (actions = left, right, suck)

↓

(8 states)

↳ 1-2 goal states

↓

In the non-deterministic case, the result of an action can vary.

Erratic vacuum world :

when sucking a dirty square, it cleans it and sometimes cleans up dirt in an adjacent square.
when sucking a clean square, it sometimes deposits dirt on the carpet.

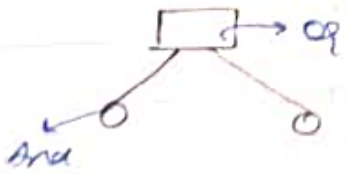
Generalisation of state space model.

① Generalize the transition function to return a set of possible outcomes.
old f : $S \times A \rightarrow s$ new f : $S \times A \rightarrow 2^S$

② Generalize the solution to a contingency plan.
if state = s then action-set-1
else action-set-2

③ Generalize the search tree to an AND-OR tree.

AND-OR SEARCH TREE : [decision tree]



searching with partial observations
The agent does not always know its state instead it maintains a belief state a set of possible states it might be in.
eg: a robot can be used to build a map of a hostile world environment. It will have sensors allowing it to see the world

Belief state space for sensorless Agent

knows it is over left

knows it is on right

online search Problems

Active agent : executes actions, acquires percepts from sensors, deterministic & fully observable, has to perform an action to know the outcome.
Eg. web search, autonomous vehicle

Agents → core of AI

search algorithm

automatic google news

Google News :

→ sports
→ ML, AI
→ central government
→ US universities
→ IEE main } frequent keep up.

→ ignored so down

"Depth limited search" + "Local beam search" combination

Iterative Deepening,
↪ Best First

News feed
↪ FB
↪ DLS, LBS

Advt in
↪ FB
↪ pages, related to adv
more adv +

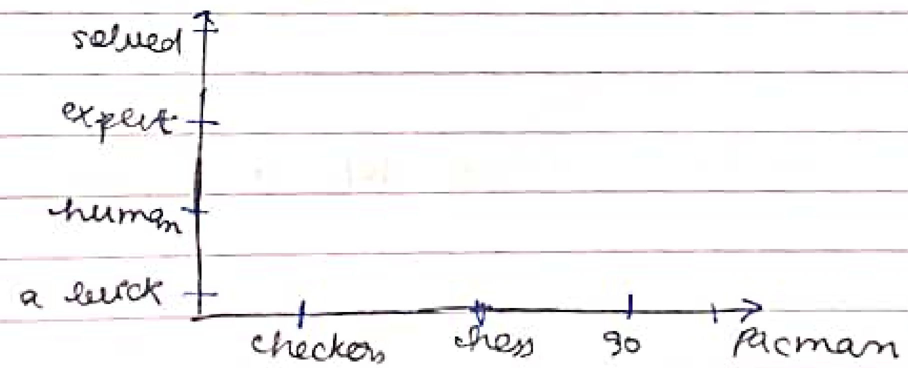
Youtube recommendations
↓
It. Deep

suggestions in FB/ Insta
↓
(same as advt)

Best First,
↑ ← Linkedin : employers looking for good resumes
Product Recommendations : E-commerce.
↪ simulated annealing / hill climbing

→ experience / skill set → heuristic function
to get the good quality human resource.

Game playing : state of the Art.



Adversarial Games :

Types of games
many different kinds of games!
Axes :
Deterministic or stochastic ?
one, two or more players
zero sum
Perfect information (is the state visible)
what algorithms for calculating a strategy (policy) which recommends a move from each state
↪ (search)

Deterministic Games

Many possible formalizations, one is:

- states S (start at s_0)
- players: $P = \{1, \dots N\}$ usually take turns (which player can move)
- actions: A (may depend on player / state)
- transition function: $S \times A \rightarrow S$ (return a set of legal moves)
- terminal test: $S \rightarrow \{t, f\}$
- terminal utilities: $S \times P \rightarrow R$

(result)

solution for a player is a policy $S \rightarrow A$

true → game is over
false → game is not over

objective function
↳ numeric value for a game.

utility function in chess
+1 (win) -1 (loss) ½ - ½ (draw)

zero-sum Games

- Agents have opposite utilities (values on outcomes)
- Let us think of a single value that one maximizes & the other minimizes
- Adversarial, pure competition (chess +1, -1, 0)

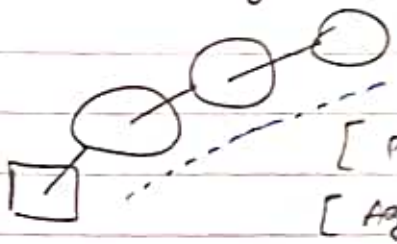
General Games [constant sum Games]

- Agents have independent utilities (values on outcomes)
- cooperation, indifference, competition, and more are all possible

Adversarial search

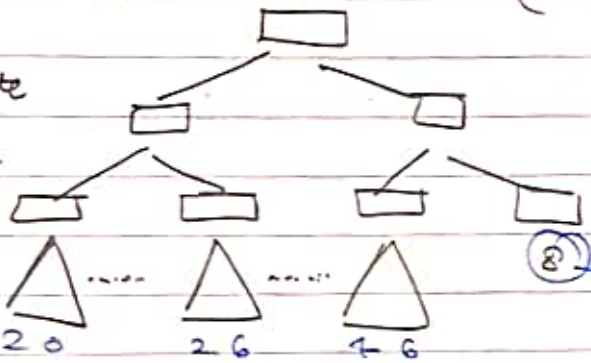
↓

eg chess : move, thinking what other person might do



other eg:
[pallanguzhi]
[Age of Empires]

single Agent Trees → (with one person)



value of a state
↳ best achievable outcome (utility) from that state

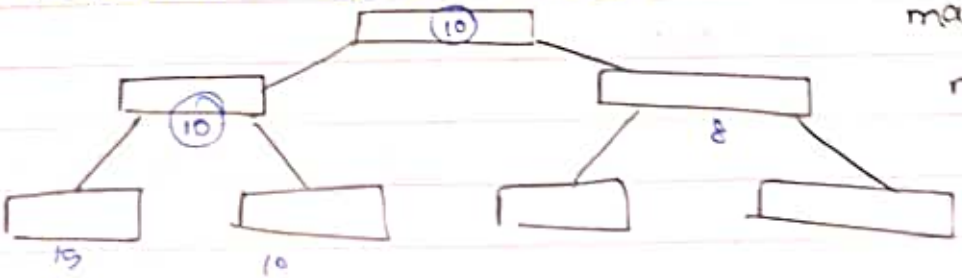
terminal states
v(s) → known

non-terminal states :

$$V(s) = \max_{s' \in \text{children}(s)} v(s')$$

(mini-max algorithm)

Adversarial Game trees



maximize
minimize.

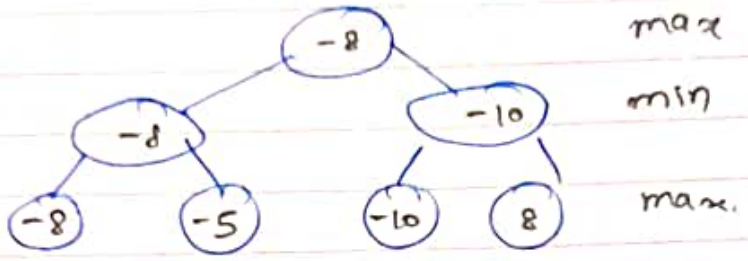
(terminal states have a utility value)

[V(s) = known]

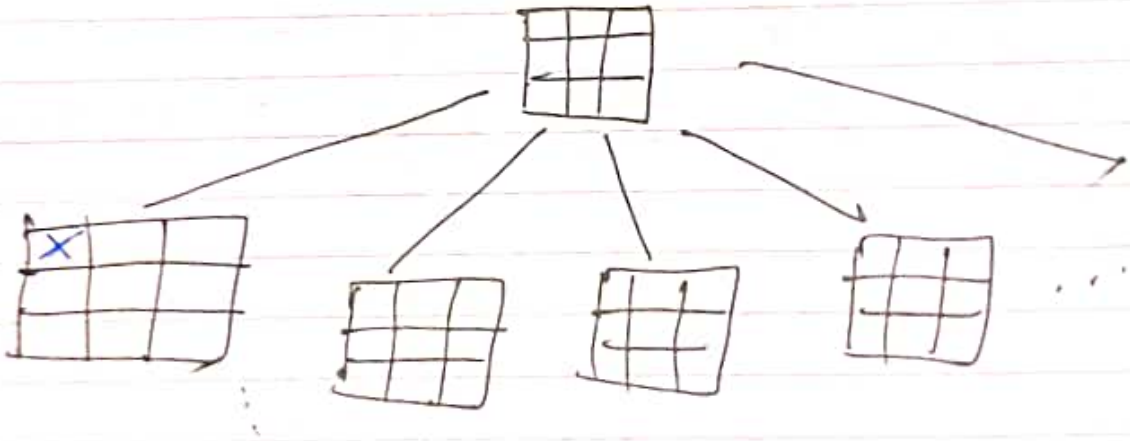
minimax values

States under opponent's control:

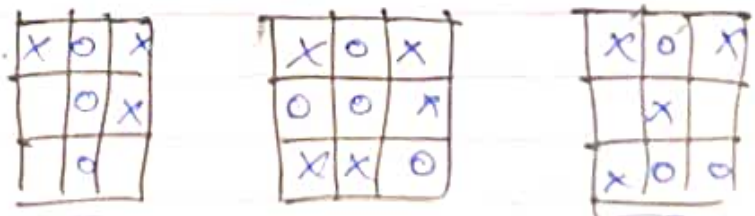
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Tic - Tac - Toe Game



Terminal

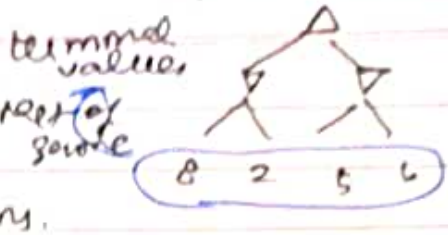


Utility → -1, 0, +1

9 × 8 × 7 × ... ⇒ 9! possible states

Adversarial search (minimax)

- deterministic, zero sum games:
  - Tic Tac Toe, chess, checkers
  - one player maximizes result
  - the other minimizes result



Minimax search:

- A state space tree.
- players alternate turns.
- compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

Minimax Implementation

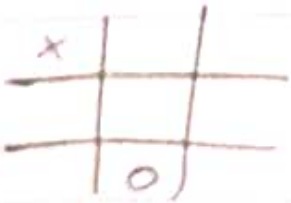
```
def max-value (state):
    initialize v = -∞
    for each successor of state:
        v = max (v, min-value (successor))
    return v

def min-value (state):
    initialize v = +∞
    for each successor of state:
        v = min (v, max-value (successor))
    return v
```

$v(s) = \max_{s' \in successors(s)} v(s')$

$v(s) = \min_{s' \in successors(s)} v(s')$

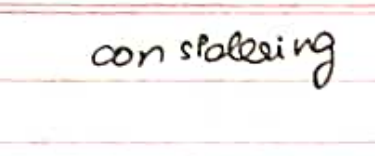
Dispatch

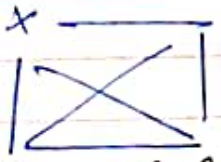
```
def value (state):
    if the state is a terminal state:
        return the state's utility
    if the next agent is MAX:
        return max-value (state)
    if the next agent is MIN:
        return min-value (state)
```



(eg)

considering



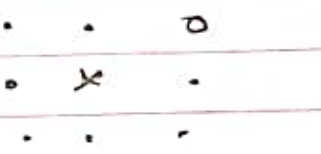


6 possibilities of success of X

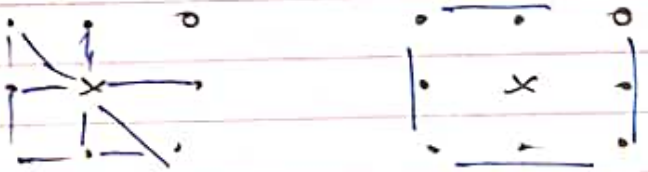
5 possibilities of success of O

value of state : 6 − 5 = (1)

considering

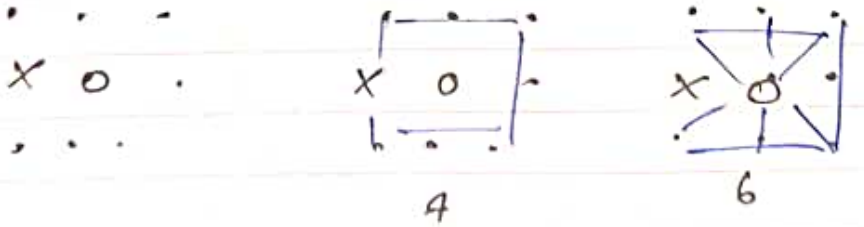


value of each state in tic tac toe can hence be calculated



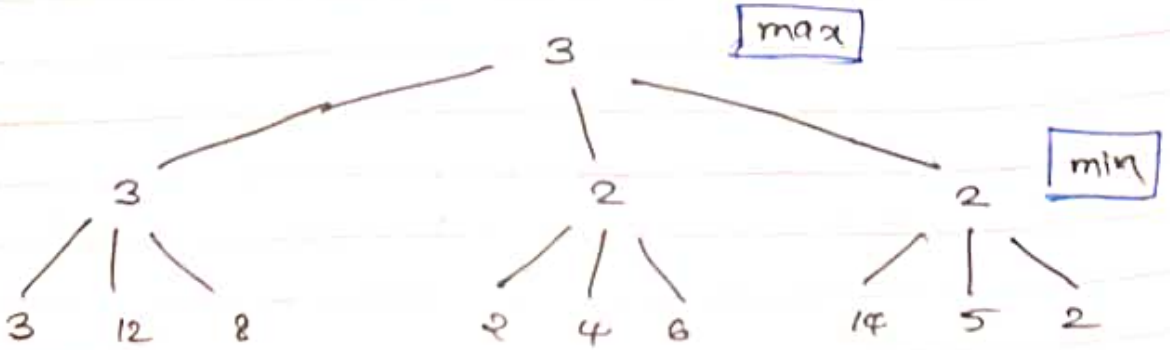
(5) − (4) = (1) heuristic by value

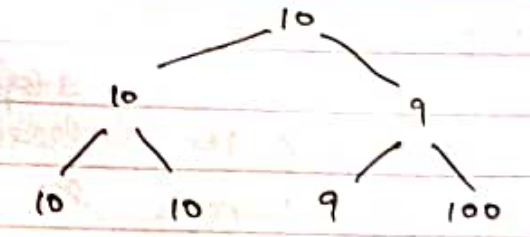
considering



4 − 6 = (−2)

minimax example





optimal against a perfect player. otherwise ?

Minmax efficiency

* just like exhaustive DFS
* Time : $O(b^m)$
* space : $O(bm)$

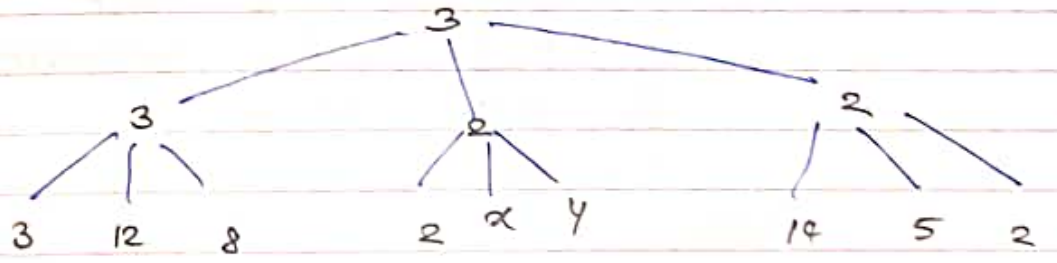
Eg : chess, $b \approx 35$, $m \approx 100$

exact solution is completely infeasible

But do we need to explore the whole tree ?

GAME TREE PRUNING

Minmax pruning



$x, y \leq 2$

Alpha-Beta pruning

General configuration (MIN version) :

* we are computing the MIN-VALUE at some node n.
* we'll looping over n's children.
* n's estimate of the children's min is dropping
* who cares about n's value ? MAX
* Let α be the best value that MAX can get any choice point along the current path from the root.

* If n becomes worse than a, MAX will avoid it, so we can stop considering n's other children (it's already bad enough that it won't be played)

[MAX version is symmetric]

alpha → minimizing

beta → maximizing

Alpha - Beta Implementation

α: MAX's best option on path to root

β: MIN's best option on path to root

```
def max-value (state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v

def min-value (state, α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Pruning Properties

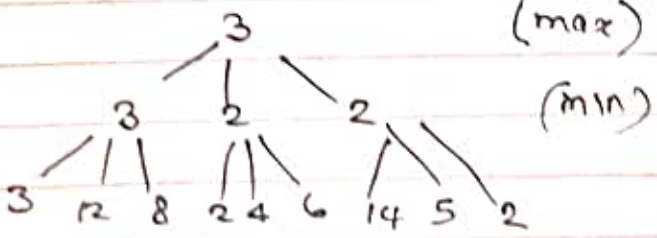
- This pruning has no effect on minimax value computed for the root.

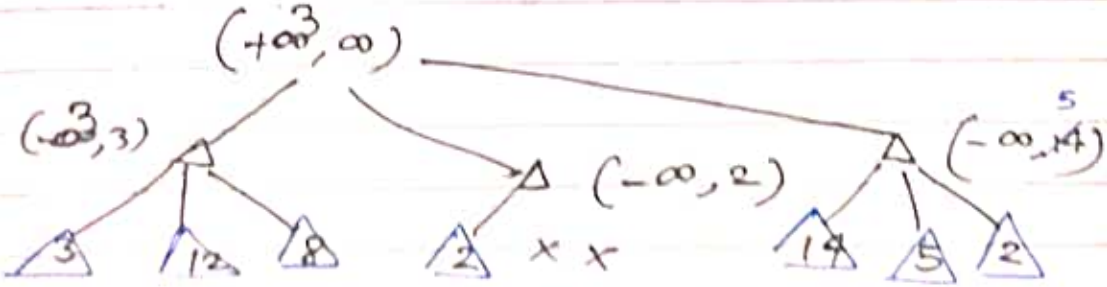
* values of intermediate nodes might be wrong
  ↓
  Important: children of the root may have the wrong value.
  So the most naive version won't let you do action selection

* Good child ordering improves effectiveness of pruning

* With "perfect ordering"
  time complexity drops to $O(b^{m/2})$
  doubles solvable depth
  full search of eg chess is still hopeless

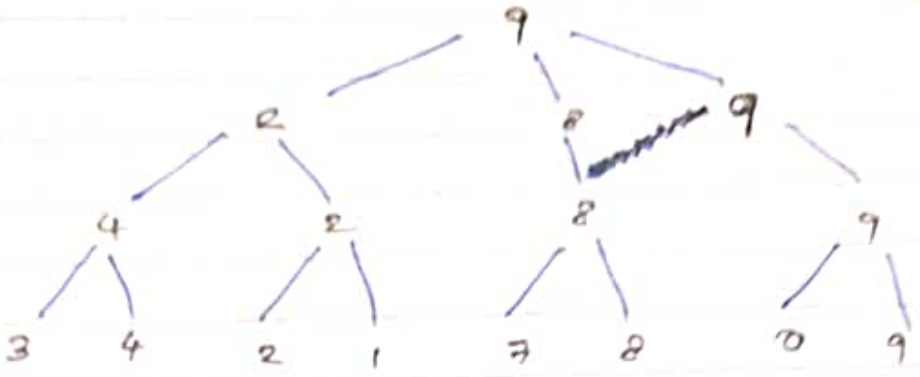
α-β Pruning example
↓

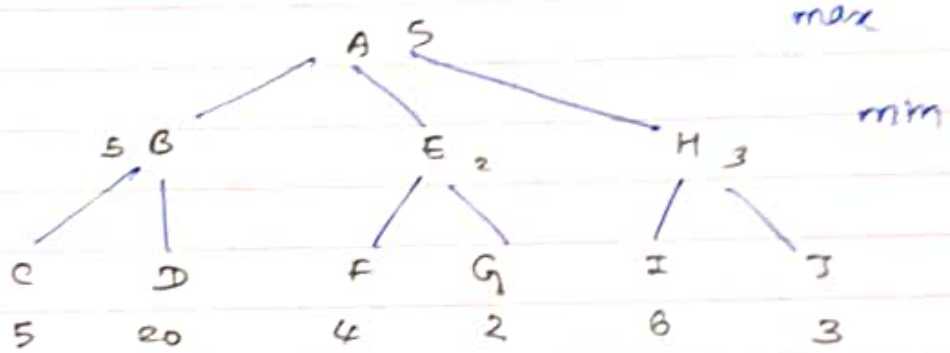




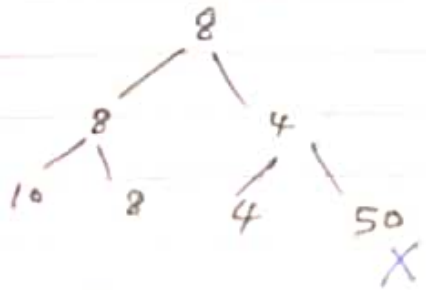
[2 nodes are pruned by observing α, β neighbours]

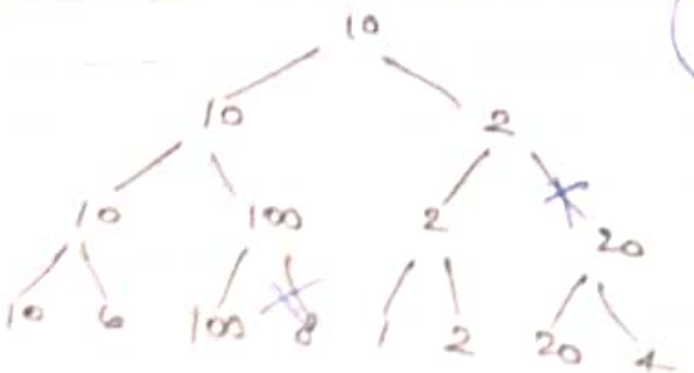
function α-β (state) returns an action
  v ⟶ max-value (state, −∞, ∞)
  return the action in actions(state) with value v





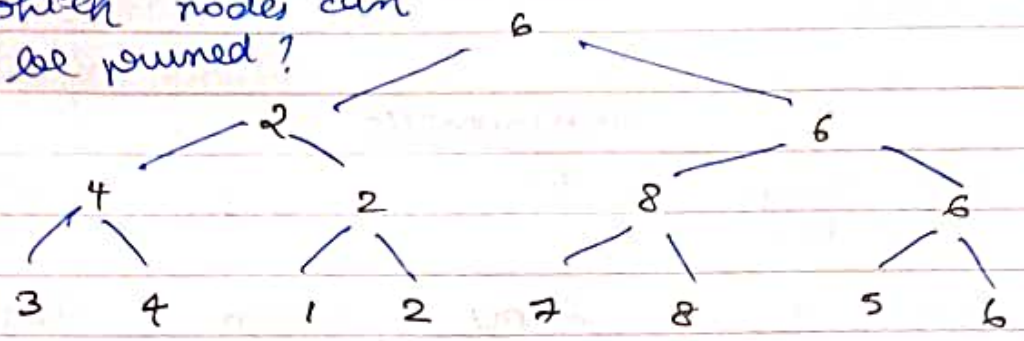
[Perfect decision games
chess, checkers, monopoly
Imperfect decision games
card games]





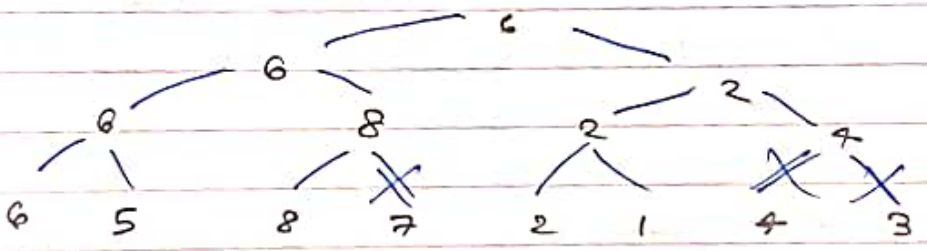
(✗ pruning)

which nodes can be pruned?



⇓

no nodes can be (expanded) pruned as the most favourable nodes for both are explored last (i.e. in the diagram, are on the right-hand side)



↓

Lots of nodes can be pruned as the most favourable nodes for both are explored first (i.e. in the diagram are on the left-hand side).

killer moves → Best moves are called as killer moves.

transpositions → repeated states occurring frequently

↓

transposition table } moves → moving! eg checkers, paragole pob solitaire

(TTC-TAE -TOE also)

Games

	Deterministic	Stochastic
Perfect Information	chess, checkers, go	backgammon, monopoly
Imperfect Information	blind tictactoe, battleships	bridge, poker, scrabble, nuclear war

Resource Limits

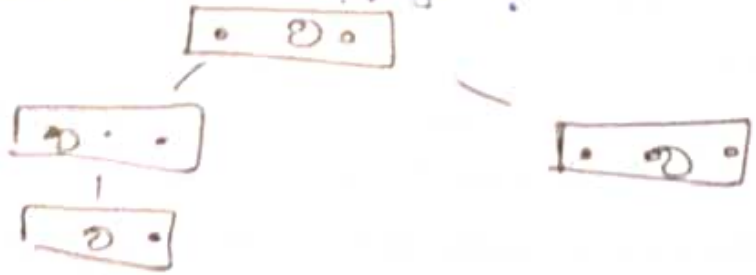
↓

* In realistic games, cannot search to leaves
* soln: Depth limited search
  Instead, only search to a limited depth in the tree
  Replace terminal utilities with an evaluation function in non-terminal positions.
* Eg
  suppose we have 100 seconds, we can explore 10 k nodes / sec.
  so can check 1M nodes per move
  α-β reaches about depth 8 – decent chess program

↓

* Guarantee of optimal play is gone
* Use iterative deepening for an anytime algorithm.

Why pacman starves ?



A danger of replanning agents!

* He knows his score will go up by eating the dot now (W, E)
* He knows his score will go up just as much by eating the dot later (W E)
* There are no point scoring opportunities after eating the dot (within the horizon, two here)
* Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning

Evaluation functions,

↳ imperfect real time decisions

* minimax algorithm generates the entire game space
* α-β pruning still has to search all the way to the terminal states
* algorithms should cut-off the search earlier and apply heuristic evaluation function. Turns non-terminal nodes to terminal leaves.
* Replace utility function by a heuristic function which estimates the position utility and replace the terminal test by a cut off test
- Returns an estimate of the expected utility of the game from any given position
* order the terminal states in the same way as the utility function
- computation must not take too long

cut-off test / terminal test

α-β search

v → max-value (state, −∞, ∞)

```
max-value (state, α, β)
    if Terminal-test (state)
        return utility (state)
    v = −∞
    max (v, min-value (successors))
```

```
H-Minimax (s, d) :
    EVAL (s)                    if cut-off test (s, d)
max H-minimax (result (s, a), d+1)   if player = max
min                                  if player = min
```

evaluation function → features

TTT TTC TTO

no. of X in your features

no. of x in a column row

```
X X ?
X
  X
```

weightage

Evaluation Functions:

Evaluation functions score non-terminals in depth limited search

Ideal function: returns the actual minimax value of the position

In practice: typically weighted linear sum of features

E. Eval (s): $w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$

no threshold

Depth matters

* Evaluation functions are always imperfect.
* The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
* An important example of the trade-off between complexity of features and complexity of computation.

Quiescent Search :
↳ unlikely to exhibit wild swings in combination of moves

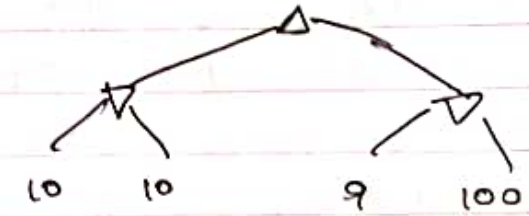
↳ eval fn : cut off test (s)

Horizon effect : → (by deferring)
↳ very difficult
program is facing opponent's move that causes a lot of damage.
(EVAL function)
↓
Avoid horizon effect : singular extension : move clearly better than all other moves

Forward pruning
↳ prune a move at a given node immediately without even considering it.
(Beam search)
↳ k-paths → best among the 'k' paths
probabilistic cut → forward pruning utilizing previous experience.
of α-β pruning :

Uncertain outcomes

Worst case vs Average case



Idea: Uncertain outcomes controlled by chance not an adversary?

Expectimax search

↓

Why wouldn't we know what the result of an action will be?

* Explicit randomness: rolling dice
* Unpredictable opponents: the ghosts respond randomly
* Actions can fail
  ↳ while moving a robot, wheels might slip

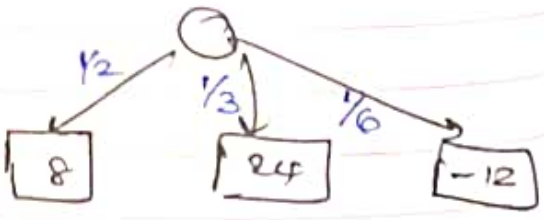
↓

Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes.

Expectimax Pseudocode

```
def exp-value (state):
    initialize v = 0
    for each successor of state:
        p = probability (successor)
        v += p * value (successor)
    return v
```

$v = (1/2)(8) + (1/3)(24) + 1/6(-12) = 10$

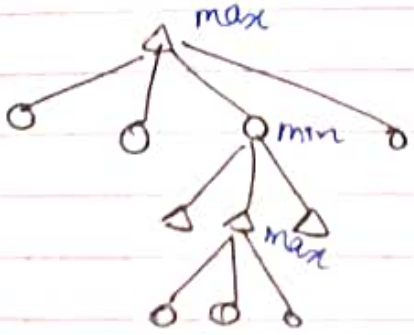


Expectimax search: compute the average score under optimal play

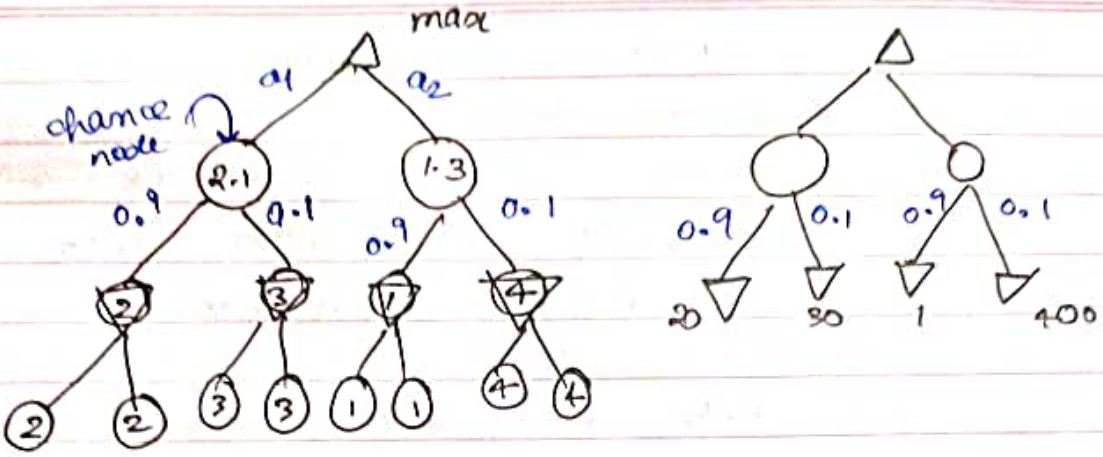
* max nodes as in minimax search
* chance nodes are like min nodes but the outcome is uncertain.
* calculate their expected utilities i.e. take weighted average (expectation) of children

Better Pseudocode

```
def max-value (state):
    initialize v = -∞
    for each successor of state:
        v = max (v, value (successor))
    return v

def value (state):
    if the state is a terminal state:
        return the state's utility
    if the next agent is MAX:
        return max-value (state)
    if the next agent is EXP:
        return exp-value (state)
```



EXPECTMINIMAX (s)
UTILITY (s) If TERMINAL Test(s)
max(EXPECTMINIMAX (Result(s,a)) If play(s) = MAX
min( ) ...
Exp (r) Ex.. Result(s,a) If play = chance



Depth-Limited Expectimax

↳ @ depth

estimate of true expectimax value (which would require a lot of work to compute)

Reminder : Probabilities.

↓

A random variable represents an event whose outcome is unknown.

A probability distribution is an assignment of weights to outcomes.

↓

Eg : Traffic on freeway

Random variable, T - whether there's traffic

outcomes : T in {none, light, heavy}

Distribution : P(T = none) = 0.25

P(T = light) = 0.50, P(T = heavy) = 0.25

Some laws of probability:

* probabilities are always non-negative
* probabilities over all possible outcomes sum to one

other Game types

Mixed layer types

eg backgammon

expectiminimax

- environment is an extra "random agent" player that moves after each min/max agent
- Each node computes the appropriate combination of its children

Example : Backgammon

Dice rolls increase b : 21 possible rolls with 2 dice

Backgammon ≈ 20 legal moves

Depth 2 = $20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$

☆

As depth increases, probability of reaching a given search node shrinks

- So usefulness of search is diminished
- So limiting depth is less damaging
- But pruning is trickier

Historic AI

↳ TD Gammon uses depth-2 search + very good evaluation function + reinforcement learning : world-level champion play

↓

1st AI world champion in any game!

Multi-agent utilities

↓

when game is not zero-sum or has multiple players ?

Generalization of minimax :

* terminals have utility tuples
* Node values are also utility tuples
* Each player maximizes its own component
* can give rise to cooperation and competition dynamically