EIKI

① What is AI?

> *The science and engineering of **making intelligent machines**, especially intelligent computer programs. It is related to the similar task of using computers to **understand human intelligence**, but AI does not have to confine itself to methods that are biologically observable.*
>
> John McCarthy, Turing Award 1971

> *The science of making machines do things that would require **intelligence** if done by men.*
>
> Marvin Minsky, Turing Award 1969

→ No easy answer to what is or isn't AI

Reasons: → no official definition
→ Science fiction influence
→ Easy tasks can become very hard
→ Hard tasks are mastered on a non-human level
→ Public perception of AI is nebulous ("neblig", "schleierhaft")

What is Intelligence

Turing Test: When does a system behave intelligently?

Assumption → if one entity cannot be distinguished from another by observing its behavior.

Test → 1. Human interrogate interacts with two players, A and B (one being a computer) blind.
2. If interrogator cannot determine whether A or B is a computer or human, the computer passes the test.

↳ Test still relevant today → different question → requires collaboration of major components of AI: knowledge, reasoning, language understanding, learning, ...

But hard/not reproducible and constructive or amenable to mathematical analysis.

Followup Question: Does being human-like mean you are intelligent?

Chinese Room Argument: Is intelligence the same as intelligent behavior?

Assumption → Even if a machine behave in an intelligent manner, it does not have to be intelligent *at all*

Test → 1. Person who doesn't know Chinese locked in a room. Outside person with paper, who can only communicate by writing Chinese and sliding notes under door.
2. Person inside has detailed instructions how to answer in chinese but can't translate it or understand it.

Follow-up Question: Is person in room intelligent? Is a self-driving car intelligent?

> The question of how an intelligent being is defined, is a long and difficult one, but...
>
> *"unlikely to have any more effect on the practice of AI research than philosophy of science generally has on the practice of science."*
>
> John McCarthy

Characteristics of AI → One AI to rule them all?

General vs Narrow AI: General / Strong AI → can handle any intellectual task
↳ Same as being vs. acting intelligent
Narrow / weak AI → specified to deal wit a concrete or a set of specified tasks.

→ Strong AI is goal in research, currently we are primarily using narrow AI.

Adaptability: improve performance by learning from experience

Autonomous: perform tasks in environments without constant guidance by user/expert

Rationality: Law of Thoughts → Beginning of reasoning and the question of what 'correct' argument and thought processes are.
Rational Behavior → question of "doing the right thing". In systems that act rationally → "right thing" is what is expected to maximize the achievement given available information
→ Rational behavior has two advantages over laws of thoughts → more general (in many situations, provably correct action non-existant)
more amenable (rationality can be defined and optimized)
However rarely a very good model of reality

Different definitions of AI due to different criteria: Two dimensions → 1. Thought process / reasonings vs. behavior / action
2. Success according to human standards vs. to an ideal concept of intelligence (rationality)

	Human	Rationality
Think	Systems that think like humans	Systems that think rationally
Act	Systems that act like humans	Systems that act rationally

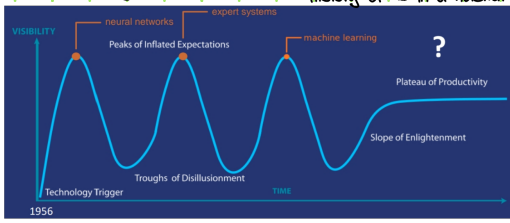
Cognitive Science: Theory of human thinking based on scientific theories of internal brain activities (cognitiv mode): → Predicting and testing human behavior
→ Identification from neurological data
(Systems that think like humans)
↳ Brings together computational models from AI and experimental techniques from psychology to construct precise and testable theories of the mind.
Often viewed distinct from AI → wrong!

Cognitive Neuroscience: How does brain work at neuronal level?

Foundations of AI: Different fields have contributed to AI and build foundation of modern AI →

- Philosophy (logic, reasoning, mind as physical system, foundations of learning, language and rationality)
- Mathematics (formal representation and proof algorithms, computation, (un-) decidability, (in-) tractability, probability)
- Psychology (adaptation, phenomena of perception and motor control)
- Economics (formal theory of rational decisions, game theory)
- Linguistics (knowledge representation, grammar)
- Neuroscience (physical substrate for mental activities)
- Control theory (homeostatic systems, stability, optimal agent design)

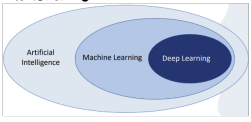
History of AI in a nutshell



Today: Nobel Prize in Physics (Hopfield & Hinton → "for foundational discoveries and inventions that enable machine learning with artificial neural networks")
Nobel Prize in Chemistry (Baker → "for computational protein design", Hassabis & Jumper → "for protein structure prediction")

Related fields and Taxonomy of AI:

Taxonomie



Subdisciplines → Machine Learning, Deep Learning, Search and Optimization, Robotics, Natural Language Processing, Computer Vision, Cognitive Science, AI Systems, Data Science, ...

Applied AI in modern fields: Autonomous Planning and Scheduling, Games (Planning, Reasoning, Understanding, ...), Robotics (Autonomous driving, ...), Natural Language Processing (Text Generation, Understanding, Translation, Speech Recognition, ...) Scientific Discovery (AI found new quasars, proved theorems, ...), Computer Vision (Autonomous vehicles, Classification, Segmentation, ...), ...

→ Current AI often isolated to single problems and not superhuman in every task
→ AI models are not without bias → fundamental differences in how AI percieves the world / environment
↳ can be tricked ↳ because of data it is fed / learning with

② AI Systems

AI System → can be defined as the study of rational agents and their environments

1. Environment „surroundings or conditions in which a person, animal or plant lives or operates"

↳ context of AI → surroundings of an (AI) agent and where agent operates
↳ does not have to be real. It could also be artificial.

Examples: Self-driving cars: Streets, traffic, weather, road signs, pedestrians, ... ; Chess: chess board, chess pieces, ... ;...

Characteristics: Discrete vs. Continuous → limited / countable number of distinct, clearly defined states → discrete, otherwise continuous (e.g. self-driving car)

Difference: ...
- Observable vs. Partially Observable or Unobservable → possible to determine complete state of environment at each time point → observable, otherwise partially observable or unobservable
- Static vs. Dynamic → environment does not change while agent is acting → static (e.g. Crossword puzzle), otherwise dynamic (e.g. Taxi driving)
- Single Agent vs. Multiple Agents → may contain other agents of same or different kind (agent design problems in multi-agent environment different from single agent)
- Accesible vs. Inaccesible → agent can obtain complete and accurate information about state's environment → accesible environment, otherwise inaccessible (empty room with state defined by temperature → accesible; event on earth → inaccessible)
- Deterministic vs. Non-deterministic / Stochastic → next stage completely determined by current state and actions of agent → deterministic, otherwise non-deterministic (stochastic event is random in nature → not determined by agent)
- Episodic vs. Non-episodic / Sequential → episodic → each episode consists of agen perciving and then acting. Quality of action depends just on episode.
  sequential → agent requires memory of past actions

Importance: Different environments need different agents
Not every algorithm works in every environment (e.g. some algorithms do not work with uncertainty i.e. need perfect information)

E.g:

Environment	Discrete?	Observable?	Static?	Single Agent?	Accessible?	Deterministic?	Episodic?
Chess	Discrete	Observable	Static	Multi-Agent	Accessible	Deterministic	Sequential

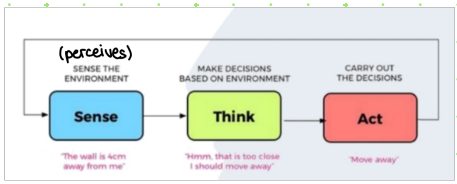
Easy and Difficult Environments:

Szenario 1: (easy) Static (no attention to possible environment changes), Observable (observe initial state at least), Discrete (Possible actions can be enumerated) Deterministic (expected outcome of action is always identical to true outcome)

Szenario 2: (Difficult) Dynamic (changing even without acting), Partially Observable (current state?), Continious (many possibilities), Stochastically (same actions in same state leads to different results)

Questions to specify/define:
- Infinite number of states?, - observe game perfectly?, - states/actions we cannot observe?, - enviroment changing?, - actions irreversible?,
- Contain other agents?, - Other agents have other forms than us?, - Opponent?, - Observe complete environment?, - observation accurate?,
- actions always result in same next state?, - randomness in environment?, - determine end state taking specific actions?, - memory of past needed to determine best action?

2. Agent



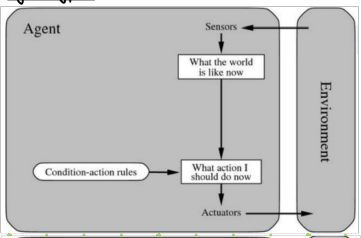
Rules: 1. Agent must be able to percieve the environment)
2. The environments observations must be used to make decisions
3. The decisions should result in action
4. (The action taken by the agent must be rational)

Example of Agents: Humans (senses, head, acts with body), Robots (sensors, decisions based on inputs, rules or a program, acts with motors for actuators), ...

→ A rational agent is an agent that does the "right thing"
↳ rational agents are not omniscient → omniscient agent would know the actual outcome of its actions
↳ rational agents are not perfect → Rationality maximises expected performance → may not be ophimal outcome (e.g. playing lottery has a negative expected outcome, better not to play, but...)

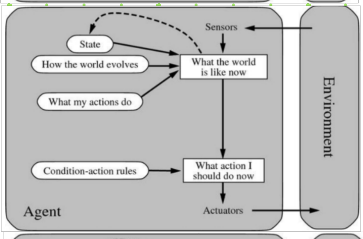
Measure performance: function that evaluates sequence of actions, task-dependent (e.g. vacuum different performance criteria (noise, time, energy, ...) than a self-driving car (safety, comfort, routing...)
- Design performance measure based on desired outcome, not desired agent behaviour (general rule)

Agent Types:



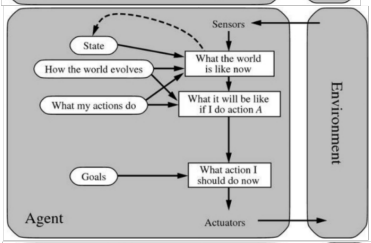
Reflex Agent: Select action on basis of only current percept but ignores percept history.
→ Implemented through condition-action rules i.e. "map state to action"
→ Makes very bad chess player → Does not look on the board, only at last move & no explicit goal of winning the game

Problem
→ very limited in its decision making
→ No knowledge about anything which the agent cannot actively perceive
→ Can become very hard to handle (store, update, ...) in complex environments



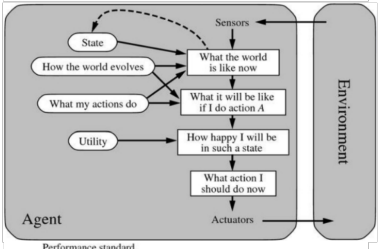
Model-based Agent: chose actions like reflex agents, but have a better comprehensive view of environment, i.e. keep track of world state
→ Input is not only interpreted, but mapped into internal state (and state dynamics) description (a world model)
→ Makes better chess agent → could keep track of current board situation when its percepts are only the moves

Problem
→ How do actions affect the world?
→ What world model do I desire?



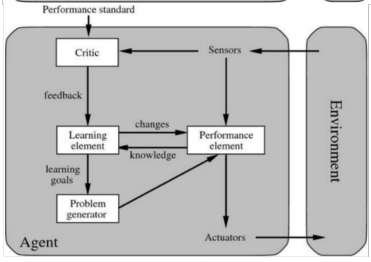
Goal-based-Agent: Build on the information that a model-based agent stores but in addition knows desirable states
→ Agent knows desireable states and will try to choose actions accordingly
→ Main difference to previous approaches: takes decision-making into account (e.g. "What will happen if I do ...?", What will make me happy?")

Problem
→ Things become difficult when long sequences of actions are required to find / achive a goal



Utility-based Agent: Instead of providing goals, Utility-based agents use utility function providing a way to rate each action / state based on desired result
→ Goals provide binary distinction, while utility function provides continuous scale
→ Can help selecting between conflicting goals (e.g. speed or safety more important)
→ Certain goals can be reached in different ways (some ways are quicker, safer, more reliable, ... (higher utility))

Utility function
→ Maps a state or sequence of state onto a real number



Learning Agents: employ an additional learning element to gradually improve and become more knowledgeable over time about an environment
→ Can learn from its past experience, i.e. is able to adapt automatically based on experience
→ More robust toward unknown environments
Has 4 conceptual components:
1. Learning Element: responsible for making improvements by learning from environment
2. Critic: Gives feedback, describing how well agent is doing with respect to fixed measurement
3. Performance Element: responsible for selecting actions
4. Problem Generator: suggesting actions which will lead to new experiences

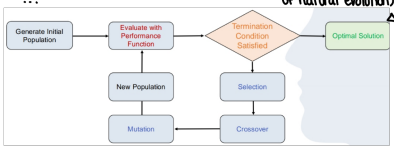
Intelligent agents make intelligent actions → How do we decide what an intelligent action is and how do we select them?

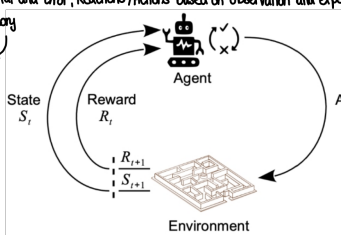
↳ Approaches: Search Algorithms: Understand/Define "finding a good action" as search problem and use search algorithms; Most common search algorithms are tree-based

Reinforcement Learning: Developed within field of psychology, "trial and Error", Reactions/Actions based on observation and experience

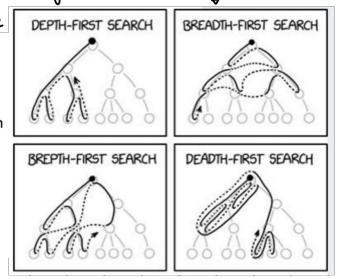
Genetic Algorithms: Survival of the fittest (Charles Darwin's theory of natural evolution)

...





Reinforcement Learning Loop



Improve Performance:

Human	Agents
Perform task	Perform task
Remember result	Save results of attempt
learn based on experience	Adjust based on saved data

③ Uninformed and Informed Search

→ Problem-solving agents are result driven → Always focus on satisfying its goals, i.e. solve problem

→ Problems given in human understandable way → reformulate problem for agent

→ Problem-solving agents employ algorithms to develop/find solutions

Steps to formulate solvable problem: formulate the goal → formulate problem given the goal

Components to formulate problem: (single state problem)

1. State Space and Initial State (Description of all possible states and initial environment)
2. Description of Actions (Function that maps state to set of possible actions)
3. Goal Test (Function to test whether current state fulfils goal definition)
4. Costs (cost function, mapping actions to costs; easy way is additive costs (sum of cost for all actions taken)

Key terminology to describe problems:

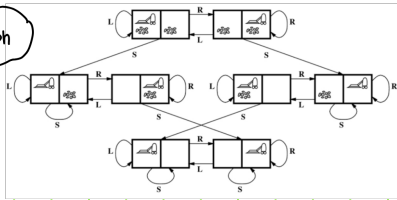
- The State Space/States → describes possible situation in environment. State space is set of all possible states.
- Transition/Action → Transitions describe possible actions between two states. Only count direct transitions (single actions)
- Costs → Transitions differ → express by adding "cost" to action. Goal in search algorithms to minimize cost to reach goal.

State Space: → set of all states reachable from initial state; defined by initial state and successor function → state-space graph

Path: sequence of states connected by sequence of actions

Solution: path from initial to goal state

Optimal solution: solution with min. path cost



Eg. ▷ finding route on map → cost = gas use in relation to distance and time

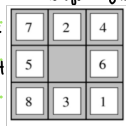
↳ Abstraction to gain problem understanding can be used to solve real world complexity

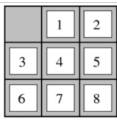
▷ 8-Puzzle:

States → Location of tiles (ignore during sliding); Goal test → Situation corresponds to goal state; Path cost → Number of steps in path; Actions → Move blank tile (left, right, up, down)

↳ easier than seperate moves for each tile

↳ ignore actions like unjamming if stuck etc.





Start State Goal State

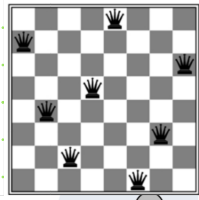
▷ 8-Queens Problem: (placing 8 queens on an chessboard without threatening each other)

States → Any configuration of 8 queens on board; Goal test → no pair threatening each other; Path cost → Not of interest; Actions → move queen to other square

↳ inefficient complete-state formulation → $64 \cdot 63 \cdot \ldots \cdot 57 \approx 3 \cdot 10^{14}$ states ▼

States → n non-attacking queens in left n columns; Goal test → // ; Path cost → // ; Actions → add queen in column n+1, without attacking others

↳ more efficient incremental formulation → 2057 states ▼

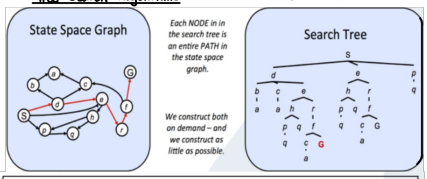


Planning Problem: transform initial state into goal considering future actions and their outcomes

Search: Process of finding (optimal) solution to such problem in form of sequence of actions.

Tree Search Algorithms → treat state-space graph as tree → use simulated iterative exploration of state space; needs strategy to determine next node

E.g. Route finding → Initial state node = start city → next nodes cities reachable from initial city → and so on with every city

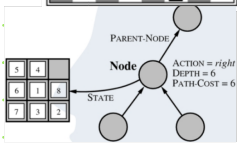


Fringe: Set of all nodes at the end of all visited paths (frontier/border)

Depth: Number of levels in tree

state: Representation of physical configuration.

Node: Data structure to present part of search tree. Includes state, parent node, taken action, path costs, current depth




```
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 the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

→ Expanding a node creates new nodes (leaf/successor nodes of chosen node) and updates tree

Search Strategy: defined by picking order of node expansion

Evaluated along: Completeness (Does it find solution if one exists), Time Complexity (Number of Node expansions), Space Complexity (Max. number of nodes in memory), Optimality (Does it always find the optimal solution)

Time and Space complexity measured in terms of:

- b: max. branching factor of search tree
- d: depth of optimal solution
- m: max. depth of state space tree

Two main groups of search strategies:

Uninformed Search
(have no information except problem definition)
- → Breadth-first search (BFS)
- → Uniform-cost search
- → Depth-first search (DFS)
- → Depth-limited search
- → Iterative deepening

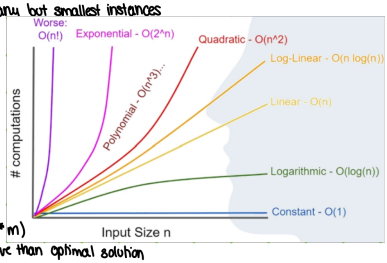
Informed Search
(have additional knowledge about problem and an idea where to "look" for solutions)
- → Greedy Best-first Search
- → A* Search
- → Memory-Bounded Heuristic Search

Uninformed Tree Search Strategies

Uniform-Cost Search (UCS)
→ Node associated with fixed cost and they accumulate over path within search. UCS uses lowest cumulative cost to find path.

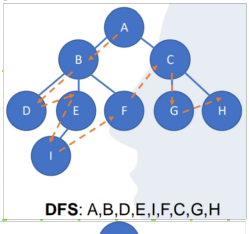
Breadth-First Search (BFS)
→ conceptually simple, but memory consumption can be too costly (Depth 4 → 1.07 kB, Depth 16 → 10 GigaBytes)
→ Special UCS case → all costs equal. Start at root and explore level by level. BFS stops as soon as goal generated → UCS examines all nodes at goal's depth → check whether one has lower cost.

Completeness: Yes, if each step has positive cost → otherwise, infinite loops possible (BFS complete aswell)
Complexity: BFS → $O(b^d)$, UCS → $O(b^{1+\text{floor}(\text{OptCost}/\text{eps})})$ every action costs at least eps
Optimality: Yes, since nodes expand in increasing order of path costs → BFS optimal.
BFS & UCS often implemented using priority queue ordered by costs
↳ exponential-complexity search cannot be solved by uninformed methods for any but smallest instances

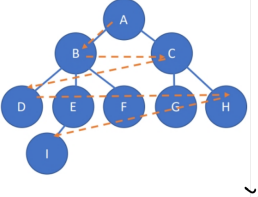


Depth-First Search (DFS)
→ Starts at tree root and explores tree as far as possible along one branch. Then going step-wise back and explore alternative branches.

Completeness: No, fails in infinite-depth search spaces and loops
↳ Can be modified to complete by avoiding repeated states and limit depth
Time Complexity: $O(b^m)$ → each branch until max. depth
↳ terrible if $m > d$, good in dense settings
Space Complexity: Only branch and unexpanded siblings stored → linear $O(b*m)$
Optimality: No, longer solutions may be found before shorter → more expensive than optimal solution



DFS: A,B,D,E,I,F,C,G,H



BFS: A,B,C,D,E,F,G,H,I

Criteria	BFS	DFS
Concept	Traversing tree level by level	Traversing tree sub-tree by sub-tree
Data Structure (Queue)	First In First Out (FIFO)	Last In First Out (LIFO)
Time Complexity	O(Vertices + Edges)	O(Vertices + Edges)
Backtracking	No	Yes
Memory	Requires more memory	Less nodes are stored normally (less memory)
Optimality	Yes	Not without modification
Speed	In most cases slower compared to DFS	In most cases faster compared to BFS
When to use	If the target is relatively close to the root node	If the goal state is relatively deep in the tree

Best of BFS & DFS: Depth-limited Search → choose l
↳ depth within search limited to l. Nodes with depth $d > l$ not considered.
Completeness: No
Time Complexity: $O(b^l)$
Space Complexity: $O(b \times l)$
Optimality: No (DFS)

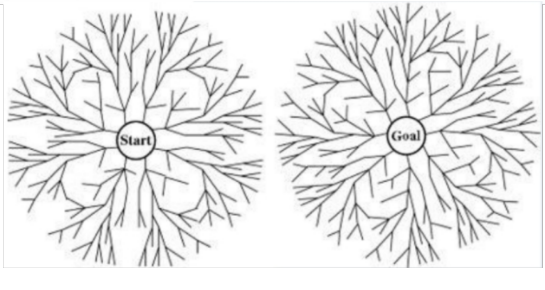
Iterative Deepening Search:
→ Increase l after each failed search
↳ l = 1, 2, 3 ...
Completeness: Yes
Time Complexity: first level searched d times → $d \cdot b + (d-1)b^2 + \ldots + 1 \cdot b^d = \sum_{i=1}^{d} (d-i+1) \cdot b^i$
Space Complexity: linear → $O(b \cdot d)$
Optimality: Yes, shortest path is found

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST(problem, STATE[node]) then return node
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
  inputs: problem, a problem
  for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
  end
```

Bidirectional Search: → two search simultaneously, starting with root and goal. Stop if node occurs in both searches
→ Reduction in complexity ($b^{d/2} + b^{d/2} \ll b^d$)
→ Only possible if actions reversable
→ Search paths may not meet for depth-first bidirectional search

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

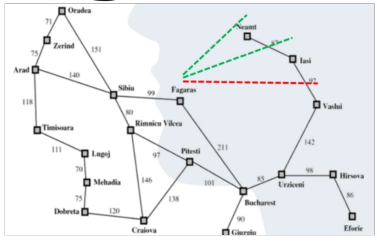


Informed Tree Search Strategies: → improve performance; giving hints about desirability of different states

Uninformed search algorithms are inefficient. → Bring knowledge into process of what node to expand to → e.g. straight-line distances may be good approximation for remaining traveldistance

Heuristics h: → can go wrong
→ Informally denotes a "rule of thumb", i.e. rule that may be helpful in solving the problem. In tree-search, heuristic denotes function h → estimates remaining costs to reach goal

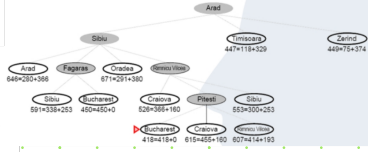
Greedy Best-first Search:



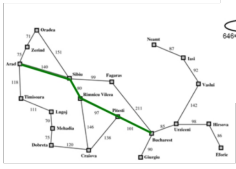
→ Using evaluation finction $f(n) = h(n)$ to estimate cost from node n to goal
↳ e.g. $h(n) = h_{SLD}(n)$ = straight-line distance from n to Bucharest (on map)
→ Expand node with smallest cost according to f(n), i.e. here the heuristic
Completeness: NO (loops) → complete in infinite state space (no loops)
Time Complexity: Worst case $O(b^m)$ → same as DFS → improved using good heuristics
Space Complexity: Worst case $O(b^m)$ → all nodes in memory
Optimality: No, solution depends on heuristic

Example:
→ heuristic not helping
From Iasi to Fagaras
Problem → Neamt "closer" to Fagaras than Vaslui

A* Search:



→ Build on best-first search → tries to minimize estimated h(n) and true g(n) cost.
Idea: Avoid expanding paths that are already expensive → evaluate complete not only remaining cost
→ i.e. $f(n) = g(n) + h(n)$
Evaluation function: g(n) → cost so far to reach n
h(n) → estimated cost to get from n to goal
f(n) → estimated cost of path to goal via n



Completeness: Yes, except if there are infinite nodes with $f(n) \leq f(G)$
Time Complexity: number of nodes grow exponentially unless error of h(n) is bounded by logarithm of value of actual path cost $h^*(n)$, i.e. $|h(n) - h^*(n)| \leq O(\log h^*(n))$
Space Complexity: keep all nodes in memory
Optimality: Depends on heuristic

Optimality of Heuristics: Example of Dominant Strategies of 8-Puzzles
Comparison of (nodes) searched by A* and Iterative Deepening Search (IDS)
→ average of 100 different 8-puzzles with different solutions
→ efficency of h2(n) [Manhatten] over h1(n) ["missing tiles"]

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Kingdome of Heuristics

Admissible Heuristics: → admissible if never overestimates cost to reach goal; formally: $h(n) \leq h^*(n)$ $h^*(n)$ true cost from n to goal; e.g. Straight-line distances ($h_{SLD}(n)$) admissable for actual distance

Consistent Heuristics: → consistent if for every n and every successor n' generated by action a it holds $h(n) \leq c(n,a,n') + h(n')$.
When going from neighboring nodes a to b, heuristic difference/step cost never overestimates actual step cost.
↳ if route from n to goal cheaper than h(n) → violates property that h(n) lower bound on cost to reach the goal
→ Every consistent heuristic is admissible.
→ If h(n) consistent, values of f(n) along any path are non-decreasing.

Designing Heuristics: e.g. for 8-puzzle
Two ideas $h_1(n)$ = (misplaced tiles) → $h(start) = 8$ → $h_{MIS}(n)$
$h_2(n)$ = "total Manhattan distance (sqaures from desired tile location)" → $h_2(start) = 3+1+2+2+2+3+3+2 = 18$ ← $h_{MAN}(n)$ → Manhattan distance
↳ admissable
$h^*(start) = 26$

Heuristics for Relaxed Problems:

Relaxed Problems: Problem with fewer restrictions on actions
"The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem."
e.g. ▷ if rules of 8-puzzle relaxed → tile can move anywhere → h_{MIS} gives shortest solution
▷ if rules are relaxed → tile can move to any adjacent square → h_{MAN} gives shortest solution
→ Looking for relaxed problems is a good strategy for inventing admissable heuristics.

Dominance:

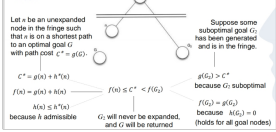
→ if h_1 and h_2 both admissable and $h_2(n) \geq h_1(n)$ for all n, then h_2 dominates h_1
e.g. h_2 dominates h_1 because if tile is misplaced, Manhattan distance ≥ 1
Search algorithms expand every node with $f(n) < C^*$ vs. $h(n) < C^* - g(n)$ → dominant heuristics result in less expansion

Combining Heuristics:

$h_1(n), h_2(n), \ldots, h_m(n)$ admissable heuristics → none dominate → "Theorem (Combining admissible heuristics):
→ $h(n) = \max\{h_1(n), h_2(n), \ldots, h_m(n)\}$ → admissible & dominates

Optimality of Heuristics:

If h(n) is admissible, A* using "tree-Search is optimal.



Assumption: If A* selects path p, p is shortest path.
Proof: Assume contradiction → p' is shortest.
- Moment before p is chosen from fringe.
- Some part of p' will also be on fringe → partial path p''.
- p was expanded before p'', $f(p) \leq f(p'')$
- p is goal, $h(p) = 0$, $cost(p) \leq cost(p'') + h(p'')$
- h admissible → $cost(p'') + h(p'') \leq cost(p')$ for any p' to goal extending p''
- Thus $cost(p) \leq cost(p')$ for any other path p' to goal → contradicts assumption that p' is shortest

3. Alternatives to A*: Memory-Bounded Heuristic Search:

Problem: A* (& Best-first) space problem

1. Iterative-deepening A* (IDA*)
- like iterative deepening → cutoff information is f-cost (g+h) instead of depth

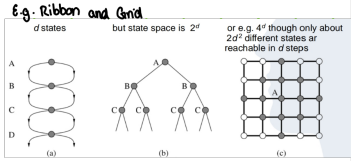
2. Recursive best-first search (RBFS)
- recursive algorithm → attempts mimicking standard best-first search with linear space
- keeps track of f-value of best alternative
- path available from any ancestor of current node and heuristic evaluations are updated with result of successors

3. (Simple) Memory-bounded A* ((S)MA*)
- drop worst leaf node when memory full
- its value updated to its parent
- may need to be re-searched later

Graph Search:

→ failure detecting repeated states can turn linear to exponential problem

E.g. Ribbon and Grid



→ Remember visited states in list → e.g. Dijkstra's algorithm is graph-search variant of uniform cost search

function TREE-SEARCH(*problem*) **returns** a solution, or failure
initialize the frontier using the initial state of *problem*
loop do
if the frontier is empty **then return** failure
choose a leaf node and remove it from the frontier
if the node contains a goal state **then return** the corresponding solution
expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
initialize the frontier using the initial state of *problem*
initialize the explored set to be empty
loop do
if the frontier is empty **then return** failure
choose a leaf node and remove it from the frontier
if the node contains a goal state **then return** the corresponding solution
add the node to the explored set
expand the chosen node, adding the resulting nodes to the frontier
only if not in the frontier or explored set

Tree Search vs. Graph Search:

Tree Search: node represents possible path to associated domain state

Problem: later detected path may be better → previous found paths reinvestigated with new, cheaper path

Solutions: 1. Add ability to detect repeated states → graph search → works also for BFS, DFS, ...
2. Ensure cheaper path is always taken first → consistent heuristics → specific solution for A*

4 Local Search and Adversarial Search → larger spaces, path isn't goal

Optimization Problem: → all states/nodes can be solution (to different degrees) → target is finding state which optimizes (min, max, ...) solution according to objective function → finding "best" states
→ no (explicit) goal state and no path (costs).

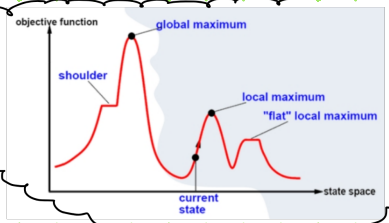
Objective/Evaluation function: → tells how good a state is (also compared to other states). value either minimized or maximized depending on optimization problem.

e.g. Darwinian evolution and survival of the fittest → optimization process

Convergence: Describes property of sequence of (function) values to approach a limit/value more and more

Global Optimum: Extremum (min. o. max.) of objective function for entire input search space.

Local Optimum: " " " " for given region of input space.



Local Search Algorithms (Iterative Improvement Methods):

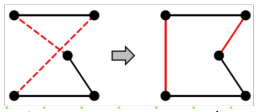
Local Search: traversing only a single state rather than saving multiple paths in memory. Modifies its state iterative, trying to improve a criteria.

try improving it by maximizing a heuristic evaluation using only "local" improvements → only modifies current stage

Goal is to find state fulfilling all goal constraints not path.

Advantages: → little/constant amount of memory usage
→ find reasonable solution in very large state spaces

Disadvantages: → No guarantees for completeness or optimality



Example:

Travelling Salesman Problem

→ Start with complete (prob. suboptimal) tour → perform pairwise exchanges → variants of approach get within 1% of optimal solution quickly with thousands of cities

n-Queens Problem

→ Place n-queens on chessboard without capturing each other → Move queen, so reduces number of conflicts

→ Only consider local changes to state at each step → series of local changes can reach all possible states

Hill Climbing / Greed Local Search:

→ expand current state → move to neighbour with highest evaluation → do so until maximum reached (evaluation goes down again)

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
inputs: *problem*, a problem
local variables: *current*, a node
neighbor, a node
current ← MAKE-NODE(INITIAL-STATE[*problem*])
loop do
neighbor ← a highest-valued successor of *current*
if VALUE[neighbor] ≤ VALUE[current] **then return** STATE[*current*]
current ← *neighbor*
end

Problem: Local Optima → algorithm stops at hilltop → hill does not have to be optimal

Idea: Randomized Hill Climbing Variants

1. Randomized Restart Hill Climbing → different initial positions result in different local optima
2. Stochastic Hill-Climbing → select successor node randomly → better nodes have higher probability of being selected

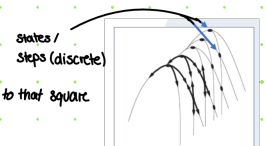
Ridge Problem / Issue:

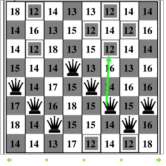
→ Every neighbour state appears to be downhill → search space has an uphill → neighbours not

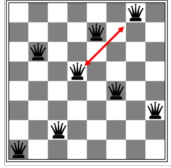
Example: 8-Queens Problem

→ Heuristic h: → number of pairs of queens that attack each other, if we move a queen in its column to that square

h=17 → Best neighbour(s): h=12







→ Local optimum with h = 1

→ no queen can move without increasing number of attacked pairs

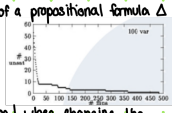
Greedy SAT:

▷ Task: find a satisfying configuration I of a propositional formula Δ

▷ Hill-climbing for SAT in a discrete place.

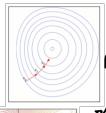
auxiliary functions:

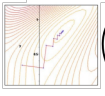
▷ violated(Δ, I): number of clauses in Δ not satisfied by I



▷ flip(I, v): assignment that results from I when changing the valuation of proposition v

```
function GSAT(Δ):
    repeat max-tries times:
        I := a random assignment
        repeat max-flips times:
            if I ⊨ Δ:
                return I
            V_greedy := the set of variables v occurring in Δ
                        for which violated(Δ, flip(I, v)) is minimal
            randomly select v ∈ V_greedy
            I := flip(I, v)
    return no solution found
```





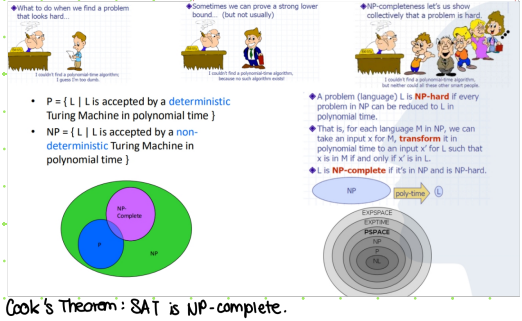
Satisfiability (SAT) Problem: given a propositional formula F in CNF, return a model (solution) to F or prove none exists

↳ Has myriads of applications

Example: F = (a + b) (a + ¬c) (¬a + c)
F has 3 models:
- M_1={a=0, b=1, c=0}
- M_2={a=1, b=0, c=1}
- M_3={a=1, b=1, c=1}

A SAT solver will return either of the 3 models

Example: F = (a) (¬a)
F is not satisfiable



Cook's Theorem: SAT is NP-complete.

Adversarial Search → playing games just search?

→ search, examine arising problem while trying to plan ahead of World and other players planning against or has conflicting goals while sharing same search space.

↳ used to model games as search problems → games presented as trees

↳ each player consider actions of other players and effect of their actions on their performance

↳ often used in combination with time limit → unlikely to find goal

Perfect Information vs. Imperfect Information → (No) information is hidden

Zero-Sum Games → if one party loses, other party wins, net change: wealth = 0

Game can be defined as a type of search problem:

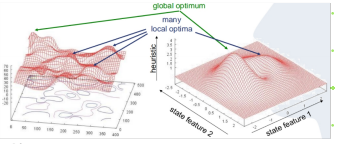
1. Initial state: How game is set up at start
2. Player(s): Specifies players turn
3. Action(s): Returns set of legal moves in state s
4. Result(s,a): Transition model, specifies resulting state s' doing move a in s
5. Terminal(s): Tests if state s fulfills goal / terminal constraints
6. Utility(s,p): Utility function returns numeric value for a terminal state s from the perspective of player p

Multi-Dimensional State-Landscape:

State Landscape → States are presented as locations with heuristic value as elevation

Problem → finding global optimum (local search) becomes worse with more dimensions

landscape view allows use of methods from mathematical analysis

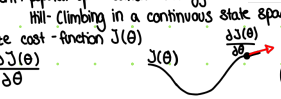


Gradient: derivative of function with >1 input var.
→ Slope of function → measures change in all weight regarding to change in error

Gradient Descent: popular optimization strategy → Hill-Climbing in a continuous state space (hiking downhill to bottom)

Goal: Minimize cost-function J(θ)

Derivative $\frac{\partial J(\theta)}{\partial \theta}$



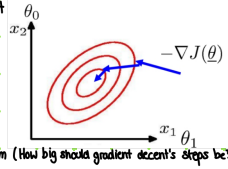
Gradient vector $\nabla J(\theta) = \left[\frac{\partial J(\theta)}{\partial \theta_0}, \frac{\partial J(\theta)}{\partial \theta_n}, \ldots\right]$

↳ contains direction of steepest step and length of it

Assume cost-function: $J(x_1, x_2, \ldots, x_n)$

→ want to minimize over continuous variables $x_1, x_2, \ldots, x_n$

1. Compute gradient: $\frac{\partial}{\partial x_i} J(x_1, \ldots, x_n)\ \forall i$
2. Take small step downhill in gradients direction: $x_i' = x_i - \lambda \frac{\partial}{\partial x_i} J(x_1, \ldots, x_n)$
3. Check if $J(x_1', \ldots, x_n') < J(x_1, \ldots, x_n)$
4. If true accept move, otherwise reject



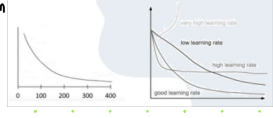
Learning rate: hyperparameter, controlling how quickly mode is adapted to problem (How big should gradient decent's steps be?)

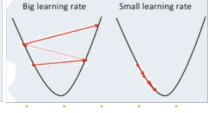
Smaller learning rates → smaller changes → more training epochs

Larger learning rates → rapid changes → fewer epochs → can converge to local optima or not at all

▷ Variants of gradient descent can improve performance in special cases (Simulated Annealing, Linear Programming)

▷ Works well in smooth spaces, poorly in rough

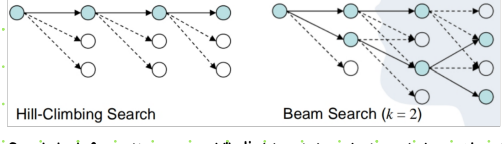




Beam Search: ▷ Keep track of k states rather than just one (k is beam size)

→ Start with k randomly generated states → at each iteration, all successors of all k states generated

→ Select the k best successors from complete list and repeat



Simulated Annealing: → use hill-climbing style techniques, but occasionally take step in direction other than where improvement (downhill away from solution)

→ As time passes probability that downhill-step is taken gradually reduced and size of any downhill step taken is decreased

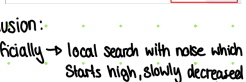
→ Allow "bad moves" to escape local optima

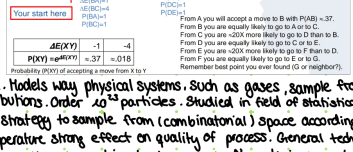
Temperature: hyperparameter, controlling how frequently "bad moves" allowed to escape local optima. Usually decays exponential over process.

Effectiveness: If lowered slowly enough → converges to global optimum (infinite search space → random guessing will find global optimum → weak claim) → can take very long

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. 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^(ΔE/T)
```

e.g.





Conclusion:

Superficially → local search with noise which starts high, slowly decreased. Models way physical systems, such as gases, sample from their statistical equilibrium distributions. Order 10^{23} particles. Studied in field of statistical physics.

True story more principled → general sampling strategy to sample from (combinatorial) space according to a well-defined probability distribution. Temperature strong effect on quality of process. General technique to sample from complex probability distributions by making local moves only. For optimization, chose close probability distribution which concentrates on optimum states for low temperature.

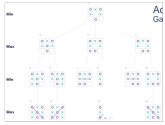
Game Trees: → to solve games

↳ arranged in levels corresponding to players

↳ root → active / current player

→ leaf nodes called terminal → each terminal node has utility value corresponding to game outcome.

Tic-tac-toe:



→ small tree

→ fewer than 9! = 362,880 terminal nodes

→ in chess → 10^{40}

Teaching AI to play games:

Games ...

1. are useful metric (allows improvement tracking, quantifiable)
2. provide safe place to train and evolve (low-stake environments, easier training than real world)
3. captures interest and imagination (showcase advancements, opens advancements to public (AlphaGo vs. Lee Sedol))

Games vs. Search Problems?

1. "Unpredictable" opponent (specify move for every possible reply, different opp. goals)
2. Time limits (unlikely enough time to find goal, needs approximation)
3. Most games are deterministic, turn-taking, two-player, zero-sum
4. But most "real" problems are stochastic, partial, multi-agent, utility based

Problem of massive game trees: → tree too big to fully traverse

(e.g. chess ($b \approx 35$) tree grows like 1, 35, 1225, 42875, 1500625, ... nodes. In Go $b \approx 250$ → "more possible Go positions than atoms in universe")

Heuristic value of functions → limit depth → for each leaf calculate value for current situation based on heuristics ↳ estimation of likely outcome if continue

Pruning → cut back tree → ignoring unwanted portions which make no difference to final result (e.g. Alpha-Beta Pruning)

(or randomly select subtrees → stochastic search)

Minimax Algorithm: → game tree → nodes represent states of game → edges // moves by players

player are:

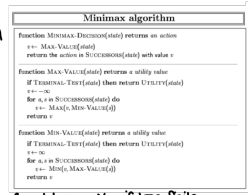
- MIN: Decrease chances of MAX to win game. (Opponent)
- MAX: Increases chances of winning

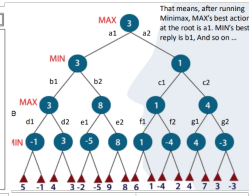
Game played turn by turn following above strategy

→ choose max position with highest minimax value → assuming opponent plays best response to action

follows DFS concept → each level either MIN o. MAX → you MAX & opponent MIN

→ on MAX level we maximize value → on MIN level we minimize





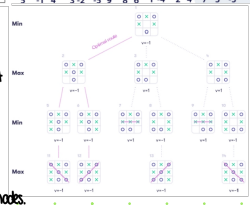
Completeness: Yes, if tree finite

Time Complexity: $O(b^m)$

Space Complexity: $O(b \cdot m)$

Optimal: Yes, assuming optimal opponent

(for chess $b \approx 35$, $m \approx 100$)



Alpha-Beta Pruning:

→ modified, optimized Minimax version

→ pruning to reduce amount of exploration without losing correctness of Minimax

Based on two parameters:

- Alpha: best (highest-value) choice found so far at any point along path of Maximizer to root. initial value of alpha $-\infty$.
- Beta: best (lowest-value) // // Minimizer // initial value of beta $+\infty$.

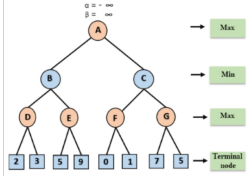
→ Remove all nodes not really affecting final decision but slowing algorithm, hence by pruning these nodes.

Key difference to Minimax: · Max player only update alpha value. Min player update beta value.

· While backtracking tree, node values passed to upper nodes instead of values of alpha and beta. → Only pass alpha, beta values to child nodes.

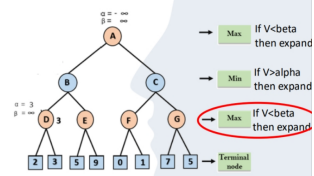
→ Video explanation link on slide 55.

Step 1



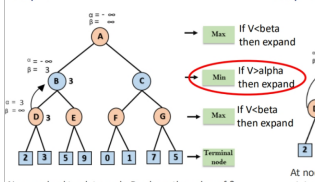
At the first step the, Max player will start first move from node A where $\alpha = -\infty$ and $\beta = +\infty$, these value of alpha and beta passed down to node B where again $\alpha = -\infty$ and $\beta = +\infty$, and Node B passes the same value to its child D.

Step 2



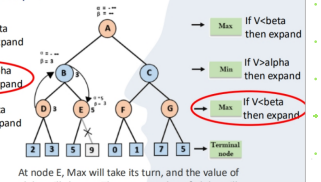
At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2= max(-∞,2) and then 3, and the max (2, 3) = 3 will be the value of α at node D. The node value will also 3.

Step 3



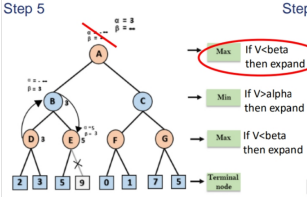
Now we backtrack to node B, where the value of β will change as this is a turn of Min. We will compare β= +∞ with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now $\alpha = -\infty$ and β= 3. We now dive into the subtree of E, since α<β

Step 4



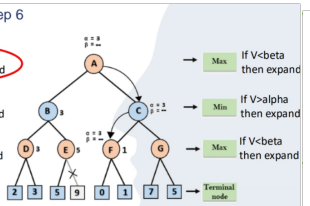
At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3. Since α>=β, the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.

Step 5



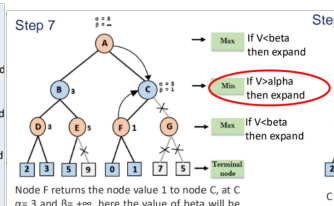
We again backtrack, from node B to node A. At node A, the value of alpha will be changed. Te maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

Step 6



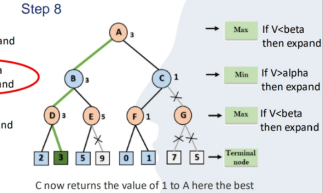
At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.

Step 7

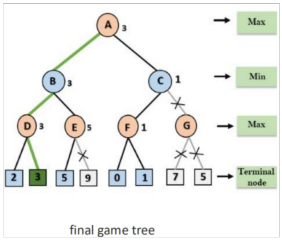


Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.

Step 8



C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Shown is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this exampl



final game tree

→ can save a lot of calculations, if tree is well ordered

→ in ideal ordering (best move first) reduce complexity to $O(b^{m/2})$ instead of $O(b^m)$

→ to optimize Alpha-Beta Pruning you have to predict interesting path to increase amount of cutoffs

Problems:

→ needs fast evaluation function

→ In games with large branching factors (e.g. Go) exploration of Alpha-Beta not good enough (to explore full minimax tree depth 3 → $300^3 = 27.000.000$ evaluations)

↳ even with pruning, too much too explore...

AlphaGo and AlphaZero:

→ In 2016, Deepmind introduced AlphaGo → combination of (Monte-Carlo) Tree Search and Neural Nets to beat Go)

slide 62 → outlook AlphaGo video

A lot of AI tasks are intractable → Search is "only" way to handle them

Many applications require some form of search (e.g. Learning, Reasoning, Planning, Natural Language Understanding, Computer Vision, ...)

↳ Even in current State-of-the-Art algorithm, search has still a key role in their success

A lot of problems can be abstracted or displayed as a search problem!

5 Constraint Satisfaction Problems

→ technique where problem is solved when solution satisfies certain constraints or rules of problem

Components: – state, defined by variables X_i with d values from domain D_i
– Goal test, defined as set of constraints C specifying allowable combinations of values for subsets of variables

Solving CSP requires: – state-space, notion of solution

Real World CSPs: Assignment problems, Timetabling problems, Hardware configuration, Spreadsheets, Scheduling, Floor planning, ...
↳ involve real-valued variables → Linear constraints solvable in polynomial time → linear programming
→ nonlinear constraints undecidable

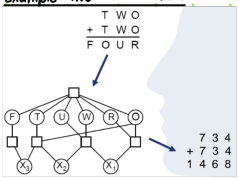
Example: n-Queens → place n queens on nxn chessboard → no two queens threaten each other
Variables: X = one variable for each row; Domain of variables: D = number between 1 and 8; Constraints: C = Enumeration of disallowed configurations

Constraint Graphs: → Abstraction of problem → easier to solve and understand
→ Every variable represented as node
→ Every edge indicates constraint between them

Example: "our Color Map "theorem

Problem: assign each territory a color such that no two adjacent territories have same color
Variables: X = {WA, NT, Q, NSW, V, SA, T}
Domain of variables: D = {r, g, b}
Constraints: C = {SA ≠ WA, SA ≠ NT, SA ≠ Q, ...}
→ Always possible to color map using 4 or less colors → ensuring regions sharing a border are different colors (videos on slides 13 & 14)

Example: TWO + TWO = FOUR



Connected nodes are involved in (in-)equations:
$2 \cdot O = 10 \cdot X_1 + R$
$2 \cdot W + X_1 = 10 \cdot X_2 + U$
$2 \cdot T + X_2 = 10 \cdot X_3 + O$
$F = X_3$
$F \neq T \neq U \neq W \neq R \neq O$

Example: Wildlife Corridor Design → slide 16

Types of constraints:
Unary constraints: involve a single variable (e.g. South Australia ≠ green)
Binary constraints: involve pairs of variables (e.g. South Australia ≠ Western Australia)
Higher-order constraints: involve 3 or more variables (e.g. $2 \cdot W + X_1 = 10 \cdot X_2 + U$)
Preferences (soft constraints): e.g. red better than green; not binding → respect as many as possible
→ constrained optimization problems

Two principal approaches to solve CSPs:
1. Search: → successively assign values to variable → check all constraints → if constraint violated → backtrack → until all variables have assigned values
2. Constraint Propagation: → maintain set of possible values D_i for each variable X_i → try to reduce size of D_i by identifying values that violate some constraints

Search:

Map CSPs into search problems: Nodes = assignments of values to subset of variables
Neighbors of a node = nodes in which values are assigned to one additional variable
Start node = empty assignment
Goal node = node which assigns value to each variable and satisfies all constraints

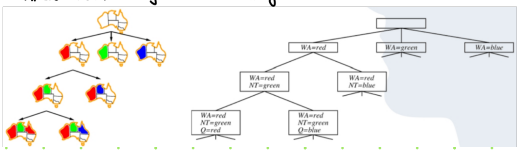
Build simple search space: Start node is empty assignment of values to variables
its successor are all possible ways of assigning a value to a variable (d=1)
their successors are those with two variables assigned (d=2)
...

Naïve Search: → slide 20 video
Complexity: n variables → all solutions are at depth n in search tree
all variables have v possible values
↳ at level 1: n·v possible assignments → chose one of n variables and one of v values for it
↳ at level 2: (n-1)·v possible assignments for each previously unassigned variable → chose one of remaining n-1 variables and one of v values for it
In general: branching factor at depth l: (n-l+1)·v
→ n·(n-1)·v·v leaves in total at d=2
...
→ Therefore, search tree has $n! v^n$ leaves

Do better?

Commutative Variable Assignments:

→ Not relevant if we do [WA=red then NT=green] or [NT=green then WA=red], assignments are commutative, order not important
→ At each node, we only need to make assignments for one of the variables → reduces total complexity to v^n



Backtracking Search → DFS with single variable assignment per level

↳ basic uninformed search algorithm for CSPs (add one constraint at a time without conflict, succeed if legal assignment is found → can solve n-queens for up to n = 25)
→ Complexity in worst case still exponential
→ Heuristics for selecting and ordering variables can improve performance

→ video link slide 25 & 27

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({}, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment given CONSTRAINTS[csp] then
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result ≠ failure then return result
      remove {var = value} from assignment
  return failure
```

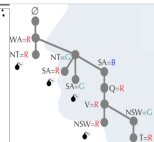
Example Sodoku: → CSP with 81 variables

Constraints: → some values are assigned in the start
→ 27 constraints on 9 values that must all be different
(9 rows, 9 columns, 9 squares)

Constraint Propagation: → list of possible values in empty fields
→ successively eliminate values

Status: Automated constraint solvers can solve hardest puzzle in no time

General-purpose methods can give huge gains in speed:
1. Next variable to assign?
2. Order in which its values be tried?
3. Detect inevitable failures early?
4. Take advantage of problem structure?



Assignment of values to variables:
- state in state-space not "blackbox" anymore → defined by assigning values to some or all variables such as $X_1 = U_1, X_2 = U_2, \ldots$

Can be done:
1. Consistent/Legal Assignment → does not violate any constraint or rules
2. Complete Assignment → every variable is assigned with value and solution to CSP remains consistent
3. Partial Assignment → assigns values only to some variables

→ don't care about path to solution → solutions described by constraints → can do better than search trees → still maintain a data structure (compared to local search)

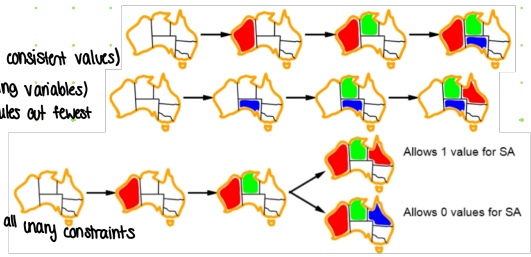
Heuristics for CSPs:

Domain-Specific Heu... ...s → depend on particular characteristics of problem

General-purpose Heuristics → for CSP good general-purpose heuristics:

- ↳ Minimum Remaining Values Heuristic (chose variable with fewest consistent values)
- ↳ Degree Heuristic (chose variable with most constraints on remaining variables)
- ↳ Least Constraining Value Heuristic (variable given, chose value that rules out fewest values in remaining variables)

→ Used in this order, these can greatly improve search speed

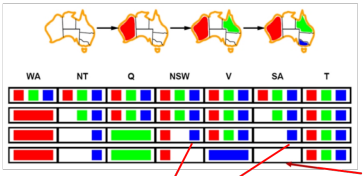


Node Consistency: → variable (node) is consistent if possible values of this variable are conform to all unary constraints

↳ Example: Sudoku → some nodes already filled out i.e. constrained to a single value

Local Consistency → defined by a graph where each of its nodes is consistent with its neighbors. Done by iteratively enforcing the constraints corresponding to edges.

Forward Checking: → Keep track of remaining legal values for unassigned variables and terminate search when any variable has no more legal values



only one of them can be blue → could have seen conflict one step earlier

no further assignment possible

Arc Consistency: → improve performance by looking at larger sets of constraints and how they interact

Arc: constraint involving two variables is called arc or binary constraint

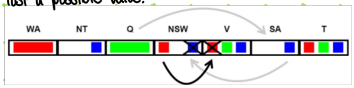
Arc Consistency: arc consistent if for each value of X in domain of X there is value Y in domain of Y such that constraint arc (X, Y) is satisfied.

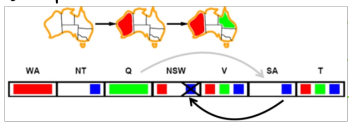
$\forall X \in dom(X), \exists Y \in dom(Y)$ such that arc (X, Y) is satisfied

→ can be generalized to n-ary constraints → each tuple involving X_i has to be consistent

Maintaining Arc Consistency (MAC): → after each new assignment of a value to a variable, possible values of neighbours have to be updated:

If one variable (NSW) looses value (blue), need to re-check its neighboors as well because they might have lost a possible value.





function AC-3(csp) **returns** the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
local variables: $queue$, a queue of arcs, initially all the arcs in csp

while $queue$ is not empty **do**
  $(X_i, X_j) \leftarrow$ REMOVE-FIRST($queue$)
  if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
    for each X_k **in** NEIGHBORS[X_i] **do**
      add (X_k, X_i) to $queue$

If X loses a value, neigbors of X need to be rechecked.

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff succeeds
  $removed \leftarrow false$
  for each x **in** DOMAIN[X_i] **do**
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint $X_i \leftrightarrow X_j$
      then delete x from DOMAIN[X_i]; $removed \leftarrow true$
  return $removed$

AC-3 algorithm

Path Consistency: Arc Consistency often sufficient to solve the problem (all variable domains reduced to size 1) or show that problem cannot be solved (some domains empty)

↳ not enough → always consistent value in neighboring region

↳ tightens binary constraints by considering triples of values

pair of variables (X_i, X_j) path-consistent with X_m if for every assignment that satisfies constraint on arc (X_i, X_j) there is assignment that satisfies the constraints on the arcs (X_i, X_m) and (X_j, X_m)

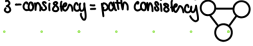
e.g. Sudoku: simple puzzles solved with AC-3 → 27 constraints → each requires that 9 values are all different

→ These 9-valued AllDif constraints can be converted into pairwise binary constraints ($9 \times 8/2 = 36$) → $27 \cdot 36 = 972$ arc constraints

→ not all problems can be solved with constraint propagation alone → search

k-Consistency: concept generalized → set of k values need to be consistent:

- 1-consistency = node consistency
- 2-consistency = arc consistency
- 3-consistency = path consistency



→ May lead to faster solution but checking for k-consistency is exponential in k in the worst case

→ Therefore arc consistency more frequently used in practice

Constraint Propagation and Backtracking Search:

→ each time a variable is assigned, constraint propagation algorithm run to reduce number of choice points in search

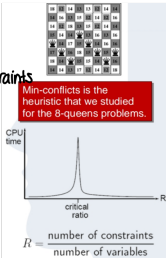
↳ improve speed of backtracking search further

Possible Algorithm: Forward Checking, AC-3 → initial queue of constraints only contains constraints with variable that has been changed

Example videos: slides 45+46

Local Search for CSPs:

- Modifications for CSPs: work with complete states, allow states with unsatisfied constraints, operators reassign variable values
- Min-conflicts Heuristic: random select conflicted variable, choose value that violates fewest constraints, hill-climbing with $h(n) =$ # of violated constraints
- Performance: can solve randomly generated CSPs with a high probability, except in a narrow range R



Power of Problem Decomposition:

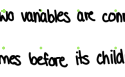
Assumption: Search space for a constraint satisfaction with n variables, each of which can have d values = $O(d^n)$

Idea: Decomposing problem into subproblems with c variables each → each problem has complexity = $O(d^c)$, there are n/c such problems →

Total complexity = $O(n/c \cdot d^c)$ → unconditional independence is rare!

Effect: total complexity can be reduced from exponential in n to linear in n (assuming c is constant) e.g. $n = 80$, $d = 2$, $c = 20$ → $2^{80} \neq 4 \cdot 2^{20}$

Tree-structured CSPs → tree structured if in the constraint graph any two variables are connected by a single path



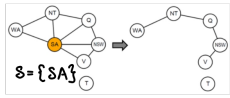
Theorem: any tree-structured CSP can be solved in linear time in number of variables ($O(n \cdot d^2)$)

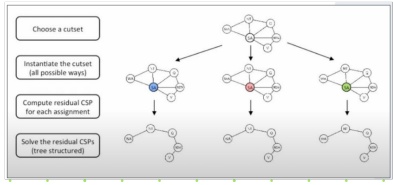
Linear Algorithm:
1. Choose variable as root → order nodes so that parent always comes before its children (each child only one parent)
2. For j=n downto 2 → Make arc(X_i, X_j) arc-consistent, calling Remove-Inconsistent-Value(X_i, X_j)
3. For i=1 to n → assign to X_i any value consistent with its parent

Problem: Tree-structures problems are rare → approaches for making problems tree-structured:
1. Cutset Conditioning → removing nodes so that remaining nodes form tree
2. Collapsing nodes together → Decompose graph into set of independent tree-shaped subproblems

Cutset Conditioning:
1. Choose subset S of variables such that constraint graph becomes tree after removal of S (= cycle cutset)
2. Choose (consistent) assignment of variables for S
3. Remove from remaining variables all values inconsistent with variables of S
4. Solve CSP problem for remaining variables
5. If no solution → choose different assignment for variables in 2.)

E.g. S = {SA}





6 Logic and AI 1: Propositional Logic → make agent smarter with logical reasoning

Syntax → Sequence of specific language which should be followed in order to form a sentence. Representation of language & related to grammar and structure.
Semantic → Sentence or syntax which logic follows should be meaningful. Defines sense of sentence which relates to real world.
e.g. Programming → Syntax: structural rules, i.e. seperate statements with semi-colon, ... (syntax correct if compiles); Semantics: Does program work? Makes sense?
Logic → key behind any (formal) knowledge. Allows person to filter necessary information and draw conclusion. In AI representation of knowledge done via logics.
Knowledge Base (KB) → represents actual facts existing in real world. central component of knowledge-based agent. Set of sentences describing information related to the world. = set of sentences in formal language
Inference Engine → engine of knowledge-based system which allows infer new knowledge in system.

Inference engine	← domain-independent algorithms
Knowledge base	← domain-specific content

- Declarative approach to building an agent (Tell it what it needs to know)
- "Ask" itself what to do → answers should follow from KB
- Agents can be viewed at knowledge level i.e. what they know

Agents:

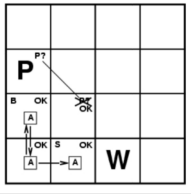
Intelligent Agents: goal-directed agent. Perceives its environment through its sensors using observations and built-in knowledge → acts upon environment through its actuators
Knowledge-based Agents: Represent states, actions, etc ...; Incorporate new percepts; Update internal representations of world; Deduce hidden properties of world & appropriate actions.

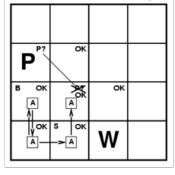
```
function KB-AGENT(percept) returns an action
  static: KB, a knowledge base
          t, a counter, initially 0, indicating time
  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Logic and AI

Wumpus:
Performance measure: gold +1000; death -1000; -1 per step; -10 for using arrow
Environment: Squares adjacent to wumpus smelly; Squares " " pit breezy; Glitter if Gold in same square; Shooting kills wumpus if you are facing it; Shooting uses up only arrow; Grabbing picks up gold if in same square; Releasing drops gold in same square
Sensors: Stench, Breeze, Glitter, Bump, Scream
Actuators: Left turn, Right turn, Forward, Grab, Release, Shoot

Reasoning turns observation into knoweldge





Wumpus in propositional logic:
Let $P_{i,j}$ be true if there is a pit in [i,j].
Let $B_{i,j}$ be true if there is a breeze in [i,j].

$\neg P_{1,1}$
$\neg B_{1,1}$
$B_{2,1}$

"Pits cause breezes in adjacent squares"
$B_{1,1} <=> (P_{1,2} \vee P_{2,1})$
$B_{2,1} <=> (P_{1,1} \vee P_{2,2} \vee P_{3,1})$

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	KB	α_1
false	false	false	false	false	false	false	false	true
false	false	false	false	false	false	true	false	true
:	:	:	:	:	:	:	:	:
false	true	false	false	false	false	false	false	true
false	true	false	false	false	false	true	true	true
false	true	false	false	false	true	false	true	true
false	true	false	false	false	true	true	true	true
false	true	false	false	true	false	false	false	true
:	:	:	:	:	:	:	:	:
true	true	true	true	true	true	true	false	false

Wumpus-World agent using propositional logic:
$\neg P_{1,1}$; $\neg W_{1,1}$
$B_{x,y} <=> (P_{x,y+1} \vee P_{x,y-1} \vee P_{x+1,y} \vee P_{x-1,y})$
$S_{x,y} <=> (W_{x,y+1} \vee W_{x,y-1} \vee W_{x+1,y} \vee W_{x-1,y})$
$W_{1,1} \vee W_{1,2} \vee \dots \vee W_{4,4}$
$\neg W_{1,1} \vee \neg W_{1,2}$
$\neg W_{1,1} \vee \neg W_{1,3}$
...

→ 64 distinct proposition symbols, 155 sentences

Syntax of Propositional Logic

- Backus-Naur Form (BNF)

↳ Symbol → P, Q, R, ..., RoommateWet, ...
Sentence → True | False | Symbol | NOT(Sentence) | (Sentence AND Sentence) | (Sentence OR Sentence) | (Sentence => Sentence)
e.g. Roommate Story → UmbrellaGone
→ UmbrellaGone => (YouCarryingUmbrella OR RoommateCarryingUmbrella)
→ NOT(YouCarryingUmbrella)

Semantics: Interpretation specifies which of proposition symbols are true and which false. Given Interpretation → able to tell whether sentence true or false. → Truth tables define semantic of operators

Model is an interpretation in which all sentences are true.

Caveats: TwoIsAnEvenNumber OR ThreeIsAnOddNumber is true (not excl. OR)
TwoIsAnOddNumber => ThreeIsAnEvenNumber is true (left false → always true)

Tautology: if sentence is true for any setting of its propositional symbols

P	Q	P OR Q	NOT(P) AND NOT(Q)	(P OR Q) OR (NOT(P) AND NOT(Q))
false	false	false	true	true
false	true	true	false	true
true	false	true	false	true
true	true	true	false	true

(P OR Q) OR (NOT(P) AND NOT(Q)) is tautology

(P => Q) OR (Q => P) ist tautology?

Logical equivalences: → two sentences are logically equivalent if they have same truth value for every setting of their propositional variables.

P	Q	P OR Q	NOT(NOT(P) AND NOT(Q))
false	false	false	false
false	true	true	true
true	false	true	true
true	true	true	true

(P OR Q) OR (NOT(P) AND NOT(Q)) is logically equivalent

(a OR b) ≡ (b OR a) | (a AND b) ≡ (b AND a) → commutativity
((a AND b) AND c) ≡ (a AND (b AND c)) | ((a OR b) OR c) ≡ (a OR (b OR c)) → associativity
NOT(NOT(a)) ≡ a → double-negation elimination

Consistency in Knowledge bases

Example: careless while specifying knowledge base:

PetOfRoommateIsABird => PetOfRoommateCanFly
PetOfRoommateIsAPenguin => PetOfRoommateIsABird
// => NOT(PetOfRoommateCanFly)
//

Problem: entails PetOfRoommateCanFly and NOT(PetOfRoommateCanFly) → technically, this KB implies anything

→ (Logical) KB must be consistent

↳ "there cannot be contradictions"

Reasoning Patterns and Modus Ponens:

→ obtain sentences directly from other sentences in KB according to reasoning patterns

↳ all of logical equivalences from before give us reasoning patterns

Modus Ponens: reasoning pattern → if we have a and a => b, we can correctly conclude b

(a => b) ≡ (NOT(b) => NOT(a)) → contraposition
(a => b) ≡ (NOT(a) OR b) → implication elimination
NOT(a AND b) ≡ (NOT(a) OR NOT(b)) | NOT(a OR b) ≡ (NOT(a) AND NOT(b)) → De Morgan
(a AND (b OR c)) ≡ ((a AND b) OR (a AND c))
(a OR (b AND c)) ≡ ((a OR b) AND (a OR c)) → distributivity

Inference / Entailment:

- knowledge base of things we know are true

- RoommateWetBecauseOfSprinklers => SprinklersOn

↳ SprinklerOn is entailed by knowledge base if, for every setting of propositional variables for which KB is true (model of knowledge base), SprinklersOn also true

RWBOS	SprinklersOn	Knowledge base
false	false	false
false	true	false
true	false	false
true	true	true

Algorithm to find out if sentence is entailed by KB
→ Go through possible settings of propositional variables
↳ if KB true and a false → return false → else return true.
Problem: not very efficient → Number of settings: $2^{\#\text{propositional variables}}$

Proof that the sprinklers are on

1) RoommateWet
2) RoommateWet => (RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers)
3) RoommateWetBecauseOfSprinklers => SprinklersOn
4) RoommateWetBecauseOfRain => NOT(RoommateCarryingUmbrella)
5) UmbrellaGone
6) UmbrellaGone => (YouCarryingUmbrella OR RoommateCarryingUmbrella)
7) NOT(YouCarryingUmbrella) — Knowledge Base
8) YouCarryingUmbrella OR RoommateCarryingUmbrella *(modus ponens on 5 and 6)*
9) NOT(YouCarryingUmbrella) => RoommateCarryingUmbrella *(equivalent to 8)* — Rephrasing/Reasoning
10) RoommateCarryingUmbrella *(modus ponens on 7 and 9)*
11) NOT(NOT(RoommateCarryingUmbrella)) *(equivalent to 10)*
12) NOT(NOT(RoommateCarryingUmbrella)) => NOT(RoommateWetBecauseOfRain) *(equivalent to 4 by contraposition)*
13) NOT(RoommateWetBecauseOfRain) *(modus ponens on 11 and 12)*
14) RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers *(modus ponens on 1 and 2)*
15) NOT(RoommateWetBecauseOfRain) => RoommateWetBecauseOfSprinklers *(equivalent to 14)*
16) RoommateWetBecauseOfSprinklers *(modus ponens on 13 and 15)*
17) SprinklersOn *(modus ponens on 16 and 3)*

Back to the Moon

1) PetOfRoommateIsABird => PetOfRoommateCanFly
2) PetOfRoommateIsAPenguin => PetOfRoommateIsABird
3) PetOfRoommateIsAPenguin => NOT(PetOfRoommateCanFly)
4) PetOfRoommateIsAPenguin
5) PetOfRoommateIsABird *(modus ponens on 4 and 2)*
6) PetOfRoommateCanFly *(modus ponens on 5 and 1)*
7) NOT(PetOfRoommateCanFly) *(modus ponens on 4 and 3)*
8) NOT(PetOfRoommateCanFly) => FALSE *(equivalent to 6)*
9) FALSE *(modus ponens on 7 and 8)*
10) FALSE => TheMoonIsMadeOfCheese *(tautology, i.e., always true)*
11) TheMoonIsMadeOfCheese *(modus ponens on 9 and 10)*

Conjunctive Normal Form (CNF)

→ Any KB can be written as formula in CNF → (... OR ... OR ...) AND (... OR ...) AND ... → "..." can be any Literal or its negation → multiple facts ANDed together

e.g. RoommateWet => (RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers) → (NOT(RoommateWet) OR RoommateWetBecauseOfRain OR RoommateWetBecauseOfSpr...)

RoommateWet
- RoommateWet

RoommateWet => (RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers)
- NOT(RoommateWet) OR RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers

RoommateWetBecauseOfSprinklers => SprinklersOn
- NOT(RoommateWetBecauseOfSprinklers) OR SprinklersOn

RoommateWetBecauseOfRain => NOT(RoommateCarryingUmbrella)
- NOT(RoommateWetBecauseOfRain) OR NOT(RoommateCarryingUmbrella)

UmbrellaGone
- UmbrellaGone

UmbrellaGone => (YouCarryingUmbrella OR RoommateCarryingUmbrella)
- NOT(UmbrellaGone) OR YouCarryingUmbrella OR RoommateCarryingUmbrella

NOT(YouCarryingUmbrella)
- NOT(YouCarryingUmbrella)

Unit resolution

→ implementing general reasoning patterns for normal form

→ l_1 OR l_2 OR ... OR l_k and NOT(l_i) → conclude: l_1 OR l_2 OR ... l_{i-1} OR l_{i+1} OR ... OR l_k
↳ modus ponens

1) RoommateWet
2) NOT(RoommateWet) OR RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers
3) NOT(RoommateWetBecauseOfSprinklers) OR SprinklersOn
4) NOT(RoommateWetBecauseOfRain) OR NOT(RoommateCarryingUmbrella)
5) UmbrellaGone
6) NOT(UmbrellaGone) OR YouCarryingUmbrella OR RoommateCarryingUmbrella
7) NOT(YouCarryingUmbrella) — Knowledge Base
8) NOT(UmbrellaGone) OR RoommateCarryingUmbrella (6,7) — Unit Resolution
9) RoommateCarryingUmbrella (5,8)
10) NOT(RoommateWetBecauseOfRain) (4,9)
11) NOT(RoommateWet) OR RoommateWetBecauseOfSprinklers (2,10)
12) RoommateWetBecauseOfSprinklers (1,11)
13) SprinklersOn (3,12)

Unit Resolution not enough :: → P OR Q , NOT(P) OR Q → can we conclude Q?

(General) Resolution:

if we have l_1 OR l_2 OR ... OR l_k
and
m_1 OR m_2 OR ... OR m_n where for some i, j, l_i = NOT(m_j) → we can conclude l_1 OR l_2 OR ... l_{i-1} OR l_{i+1} OR ... OR l_k OR m_1 OR m_2 OR ... OR m_{j-1} OR m_{j+1} OR ... OR m_n
→ Remove reappearing literals

1) RoommateWet
2) NOT(RoommateWet) OR RoommateWetBecauseOfRain OR RoommateWetBecauseOfSprinklers
3) NOT(RoommateWetBecauseOfSprinklers) OR SprinklersOn
4) NOT(RoommateWetBecauseOfRain) OR NOT(RoommateCarryingUmbrella)
5) UmbrellaGone
6) NOT(UmbrellaGone) OR YouCarryingUmbrella OR RoommateCarryingUmbrella
7) NOT(YouCarryingUmbrella) — Knowledge Base
8) NOT(RoommateWet) OR RoommateWetBecauseOfRain OR SprinklersOn (2,3) — Resolution
9) NOT(RoommateCarryingUmbrella) OR NOT(RoommateWet) OR SprinklersOn (4,8)
10) NOT(UmbrellaGone) OR YouCarryingUmbrella OR NOT(RoommateWet) OR SprinklersOn (6,9)
11) YouCarryingUmbrella OR NOT(RoommateWet) OR SprinklersOn (5,10)
12) NOT(RoommateWet) OR SprinklersOn (7,11)
13) SprinklersOn (1,12)

Horn Clauses: (special case)

→ implications with only positive literals X_1 AND X_2 AND X_4 → X_3 AND X_6
True → X_1

→ to tell whether x_j is entailed → follow implications (MP) as far as possible → reach x_j? from where WTF?
→ if can be reache x_j entailed
→ implement more efficiently by counting reached literals

Use Resolution for Systematic Inference:

Satisfiable: exists a model that makes modified KB true i.e. mod. KB consistent

Strategy: see if sentence entailed, add NOT(a) to KB an see if it becomes inconsistent

→ CNF for mod. KB satisfiable only if sentence a not entailed

Resolution Algorithm:

Given: formula in CNF
1. Find two clauses with complementary literals
2. Apply resolution
3. Add resulting clause (if not already there)
4. Test, if it results in empty clause, formula not satisfiable

Example: KB: 1) RoommateWetBecauseOfSprinklers
2) NOT(RoommateWetBecauseOfSprinklers) OR SprinklersOn
→ add: 3) NOT(SprinklersOn)
from 2 and 3 get: 4) NOT(RoommateWetBecauseOfSprinklers)
from 4 and 1 get: empty clause

Limitations of Propositional Logic:
→ some statements hard to model → e.g. any (count)
→ No notion of objects
→ No notion of relations among objects

→ Roommate Carrying Umbrella instructive to us suggesting:
objects: Roommate, Umbrella
relationship between objects: carrying

formally none of this meaning there → replaceable by any other placeholder names

⑦ Logic and AI 2: First-Order Logic

Elements of First Order Logic (FOL): Objects, Relations, Functions, Equality

functions: → used to encode integers and also data structures
→ could have seperate name for Roommate (Person0) e.g. Roommate(Person0) = Person1 → not necessary
→ can be applied to any object

video on slide 8

Universal and Existential Quantifier: → talk about many objects at once

→ Define variables → refer to one or multiple
for all: ∀ e.g. all lions are cats ∀x: Lion(x) => Cat(x)
there exists: ∃ e.g. there is a cat that is not a lion ∃x: Cat(x) => NOT(Lion(x))

"John has Jane's umbrella"
- Has(John, Umbrella(Jane))

"John has an umbrella"
- there exists y: (Has(John, y) AND IsUmbrella(y))

"Anything that has an umbrella is not wet"
- for all x: ((there exists y: (Has(x, y) AND IsUmbrella(y))) => NOT(IsWet(x)))

"Any person who has an umbrella is not wet"
- for all x: (IsPerson(x) => ((there exists y: (Has(x, y) AND IsUmbrella(y))) => NOT(IsWet(x))))

"John has at least two umbrellas"
- there exists x: (there exists y: (Has(John, x) AND IsUmbrella(x) AND Has(John, y) AND IsUmbrella(y) AND NOT(x=y))

"John has at most two umbrellas"
- for all x, y, z: ((Has(John, x) AND IsUmbrella(x) AND Has(John, y) AND IsUmbrella(y) AND Has(John, z) AND IsUmbrella(z)) => (x=y OR x=z OR y=z))

∀x: a equivalent to NOT(there exists x: NOT(a))

Resolution for FOL:

for all x: (NOT(Knows(John, x)) OR IsMean(x) OR Loves(John, x))

for all y: (Loves (Jane, y) OR Knows(y, Jane))

↳ substitution {x/Jane, y/John}

IsMean(Jane) OR Loves (John, Jane) OR Loves(Jane, John) → Complete (i.e. if not satisfiable, will find proof of this), if we can remove literals that are duplicates after unification → put everything in canonical form first

First-order CNF (canonical form used for resolution):
1. Convert to Negative Normal Form i.e. negation only before predicate symbols
2. Rename variable names used twice within scopes of different quantifiers to names not already used
3. Skolemize Statement: 1. Have quantifiers out so we get "for all $x_1, x_2 \ldots$ Exists $Y_1, Y_2, \ldots$ formula → Quantifiers only appear in initial prefix but never inside NOT, AND, OR
   2. Replace existentially quantified variables by Skolem functions
   3. Discard all universal quantifiers
4. Convert into clause set

e.g.

Example	Steps to CNF (see previous slide)
1. Anything anyone eats and is not killed, is food.	1. Text to FOL
2. For all x: For all y: eats(x,y) AND NOT(killed(x)) => food(y)	2. Eliminate all implications
3. For all x: For all y: NOT(eats(x,y) AND NOT(killed(x))) OR food(y)	3. Move Negations inwards
4. For all x: For all y: NOT(eats(x,y)) OR killed(x) OR food(y)	4. Rename variables
5. NOT(eats(x,y)) OR killed(x) OR food(y)	5. Eliminate existential quantifiers (Skolemizaiton)
	6. Drop universal quantifiers
	7. Convert into clauses

Axioms and Theorems:
Axioms → basic facts about domain → „initial" KB
Theorems → Statements logically derived from axioms

Substitution: → talk about smaller, more specific sets of objects
↳ SUBST replaces one or more variables with smth else in a sentence
e.g. SUBST ({x/John}, IsHealthy(x) => NOT(HasA Cold(x))) →
IsHealthy(John) => NOT(HasACold(John))

Instantiating quantifiers with SUBST:
from for all x: a → for any g SUBST({x/g}, a) — may add to KB any instantiation of a, where x is replaced by concrete ground term g

from there exists x: a → for one k SUBST({x/k}, a) — may add single instantiation of a to KB, where x is replaced by Skolem constant k → must not appear elsewhere in KB
Important: k is constan not reappearing in KB (Skolem constant)
original sentence no longer needed
Instantiate existentials after universals → more careful
for all x: there exists y: IsParentOf(x, y) → for all x: IsParentOf(x, k(x))
↳ Skolem constant depending on x i.e. becomes function

Generalized modus ponens:
→ for all x: Loves(John, x) → John loves every thing
→ for all y: (Loves(y, Jane) => FeelsAppreciatedBy(Jane, y))
↳ Jane feels appreciated by every thing that loves her
→ infer from: FeelsAppreciatedBy(Jane, John)

Used Substitution {x/Jane, y/John} and first get rid of existential quantified (moving them after the all quantifiers and then Skolemize)
→ Different variables for different sentences → general Unify algorithms for finding a good substitution

Unify: → keeping things as general as possible in unification
Consider EdibleByWith e.g. Edible_ByWith(Soup, John, Spoon)
for all x: for all y: EdibleByWith (Bread, x, y)
for all u: for all v: EdibleByWith (u, v, Spoon) => CanBeServedInBowlTo(u, v)
↳ Anything that is edible with a spoon by something can be served in a bowl to that something
Substitution: {x/z, y/Spoon, u/Bread, v/z} → for all z: CanBeServedInBowlTo(Bread, z)
Alternative {x/John, y/Spoon, u/bread, v/John} → CanBeServedInBowlTo (Bread, John) → not as general

Interference in FOL:
→ Deciding whether sentence is entailed is semidecidable: → algorithms that will eventually produce proof of any entailed sentence
↳ not decidable → cannot always conclude that sentence is not entailed

Program in Logic (Prolog)

objects: Person, Food

Person	Food
Sam	dal
Sam	curry
Josie	curry
⋮	⋮

Program: eats
↳ eats(Sam, dal)
eats(Sam, curry)
eats(Josie, curry)

→ acts like subroutine → type at Prolog prompt eats(Person1, 'food1') → % constraint over two variables
↳ conjoin two constraints with comma:
eats('Person1', 'food'), eats('Person2', 'food')
↳ solution: Person1 = Sam, Person2 = Sam, food = dal (press ;)
Person1 = Sam, Person2 = Sam, food = curry
Person1 = Sam, Person2 = Josie, food = curry

Variables start with A, B, ... Z or underscore
Constant "atoms" start with a, b, ... z or in single quote: curry, 'CS325'
↳ other constants: Integers, real numbers, empty list

compatible(Person1, Person2) :- eats(Person1, food), eats(Person2, food)
head "if" & body

→ Person 1 & 2 compatible if food exists they both eat
→ to satisfy head of rule → satisfy body
→ compatible(Josie, X) → answer: X = Sam → does not report food or Person 1 or 2 → local variables

Queries in Prolog: → things you type at prompt
↳ answers as yes or no depending if satisfying assignment exists
↳ prints first assignment before printing yes
↳ pressing enter = accepting pressing ";" → backtrack and look for another

Constants vs. Variables: → eats(Josie, food) → what does Josi eat
eats(Person, curry) → who eats curry
eats(Josie, food), eats(Person, food) → Who does Josie share food with?

Prolog Solver: depth-first search using unification and backtracking

Query → eats(sam,x) → Iterates in order through program's "eats" clauses
↳ first match: eats(sam,dal) → returns X=dal
↳ ";" → backtracks and continues → next eats(sam,curry) → returns X=curry

→ eats(..., ...) facts can be regarded as rows in database

↳ standard relational database operations: - eats(X,dal) % select.
- edible(Object) :- eats(Someone, Object). % project
- parent(X,Y) :- mother(X,Y). % union
parent(X,Y) :- father(X,Y).
- sister_in_law(X,Z) :- sister(X,Y), married(Y,Z). % join

→ Prolog's backtracking can be inefficient

→ video Prolog and Horn Clauses → slide 31

Applications: (FOL)

→ serious novel mathematical results proven

→ Verification of soft- and hardware → prove outputs satisfy required properties for all inputs

→ Synthesis of hard- and software → try to prove existance of program satisfying such and such properties → in a constructive way

Gödel's incompleteness Theorem: → video link on slide 32, 34

→ FOL cannot model basic arithmetic

→ for any consistent system of axioms that can capture basic arithmetic there exist true sentences not proveable by axioms

→ shows there is no chance to prove every truth in "universe"

Challenging exercise: → another exercise on slide 36

Suppose: → 3 objects in world → if x spouse of y, then y spouse of x

Prove: Something is its own spouse

there exist x,y,z : (NOT(x=y) AND NOT(x=z) AND NOT(y=z))

for all w,x,y,z : (w=x OR w=y OR w=z OR x=y OR x=z OR y=z)

for all x,y : ((Spouse(x) = y) => (Spouse(y) =x))

for all x,y : ((Spouse(x)=y) => NOT(x=y)) → contradiction

Forward and Backward Chaining:

Forward Chaining: Bottom-up approach → starting initial state and reaches goal state, Process of making conclusion based on known facts or data, data-driven → we reach goal using available data

Backward Chaining: Top-down approach, based on modus ponens inference rule, goal broken into sub-goal(s) to prove facts true, goal-driven approach → list of goals decides which rules are selected and used, depth-first search strategy for proof

Something we cannot do in FOL:

→ not allowed to reason in general about relations and functions

→ „If John is Jack's roommate, then any property of John is also a property of Jack's roommate" → higher order logic → (John = Roommate(Jack)) =>
for all p:(p(John) => p(Roommate(Jack))

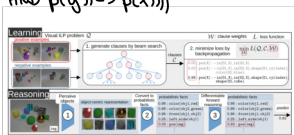
"If a property is inherited by children, then for any thing, if that property is true of it, it must also be true for any child of it" →
for all p:(IsInheritedByChildren(p) => (for all x,y: ((IsChildOf(x,y) AND p(y)) => p(x)))

AI Debate 2: link on slide 40

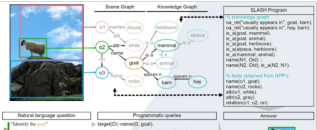
Symbols, Neurons and Neuro-symbolic?:

→ Symbol: „A physical symbol system has the necessary and sufficient means for general intelligent action" ~Allen Newell & Herbert Simon

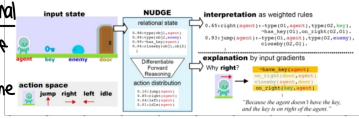
→ Neurons: „Symbols are luminiferous Aether of AI" ~Geoff Hinton



Differentiable inductive logic programming for complex visual scenes (Shindo, Pfanschilling, Dhmai, Kersting MLJ 2023)



Visual Question Answering using Neural-Probabilstic Answer-Set Programming (Skryagin, Ochs, Dahmi, Kersting JAIR 2023)



Explainable AND interpretable neuro-symbolic reinforcement learning agents using differentiable logics (Delfosse, Shindo, Dhami, Kersting NeurIPS 2023)

Focus on computational and algorithmic level not just implementational

Computational: Why do things work the way they do?; What is goal of computation?; What are unifying principles?

Algorithmic: What representations can implement such computations?; How does the choice of representations determine the algorithm?

Implementational: How can such a system be built in hardware? How can neurons carry out the computations?

8 Uncertainty → agents have to deal with uncertain situations / uncertainty

↳ qualification problem → impossibility to model all things that can go wrong

Probabilities: → way of handeling uncertainty

↳ may summarize effects that are due to laziness (don't want to list everything that can go wrong), Theoretical Ignorance (some things cannot be known), Practical Ignorance (things might not be known about situation)

→ can be related to (subjective) beliefs: → probability p means I believe statement will be true in p·100% of cases → e.g. traffic in 10% of cases → not traffic by degree of 10%!

▸ probability theory is about the degree of belief not degree of truth

↳ probability of propositions change with new evidence (e.g. $P(A_{45}$ gets me there in time | no reported accidents) = 0.06 but $P(A_{45}$ will get me there | no reported accidents, 5a.m) = 0.15)

↳ Consider probability that sun is still there tomorrow → difficult to observe by experiment

↳ Does patient have illness → Compare to other patients, but don't take too many information into consideration or else very hard to compare

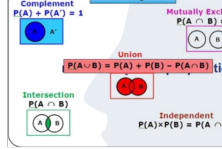
Basics:

▸ state or sample space → set of all samples (e.g. die → 1,2,3,4,5,6) Ω $\omega \in \Omega$

▸ probability space/model → sample space with assignment of probabilities per possible sample (e.g. $P(1)=P(2)=P(3)=\dots=P(6)=\frac{1}{6}$) $P(\omega)$ for every $\omega \in \Omega$; $\sum_\omega P(\omega) = 1$

▸ event → subset of sample space (e.g. P(roll greater 4) = $P(5)+P(6) = \frac{1}{6}+\frac{1}{6} = \frac{1}{3}$) $P(A) = \sum_{\omega \in A} P(\omega)$

Kolmogorov's Axioms of Probability:
1. All probabilities between 0 & 1 → $0 \leq P(a) \leq 1$
2. Necessarily true propositions have probability 1, ... have probability 0 → P(false) = 0, P(true) = 1
3. Probability of disjunction is $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$
4. Axioms restrict set of probabilistic beliefs a agent can (reasonably) hold → similar to logical constraints



Example why you shouldn't violate the axioms of probability:

Example:

- suppose Agent 1 believes the following: $P(a) = 0.4, P(b) = 0.3, P(a \vee b) = 0.8$
- Agent 2 can now select a set of events and bet on them according to these probabilities so that she cannot loose (Agent 2 offers e.g. $6 against Agent 1's $4 for proposition a)

Agent 1		Agent 2		Outcome for Agent 1			
proposition	belief	bet	stakes	$a \wedge b$	$a \wedge \neg b$	$\neg a \wedge b$	$\neg a \wedge \neg b$
a	0.4	a	4:6	-6	-6	4	4
b	0.3	b	3:7	-7	3	-7	3
$P(a \vee b) \neq P(a) + P(b)$	0.8	$\neg(a \vee b)$	2:8	2	2	2	-8
Agent 2 now designs a game, choosing to bet $4 on a, $3 on b and $2 on not(avb). E.g., if a holds, Agent 1 would have to pay $6, i.e. -6				-11	-1	-1	-1
				So summing up, Agent 1 always loses money			

Propositions, or towards uncertain knowledge:

→ Often in AI applications sample points are defined by values of a set of random variables
↳ i.e. Sample Space is Cartesian product of ranges of variables

→ With Boolean variables, sample points = propositional logic model (e.g. A = true, B = false, or $A \wedge \neg b$)

→ Proposition = disjunction of atomic events in which it is true (e.g. $(a \vee b) \equiv (\neg a \wedge b) \vee (a \wedge \neg b) \vee (a \wedge b) \rightarrow P(a \vee b) = P(\neg a \wedge b) + P(a \wedge \neg b) + P(a \wedge b)$

Random variable → function from atomic events to some range of values
e.g. Roulette → Atomic event: 0-36
Random variables with outcomes true or false: Rouge/Noir, Pair/Impair, Passe/Manque, Transversale, Carre, Cheval, ...
↳ rouge(36) = true

probability function P over atomic events induces a probability distribution over all random variables X
↳ $P(X = x_i) = \sum_{\omega : X(\omega) = x_i} P(\omega)$

Syntax:

▷ Propositional or Boolean random can be true or false (e.g. hasUmbrella (Proposition))
▷ Discrete random variables (finite or infinite) (e.g. Weather one of <sunny, rain, cloudy, snow>
↳ Weather = rain → proposition; values must be exhaustive and mutually exclusive
▷ Continuous random variables (bound or unbound) (e.g. Temp is unbound variable)
↳ Temp = 25.5, Temp > 23 are propositions

Joint Distributions: → gives probability of combined events

(e.g. probability that X = x and Y = y → true $P(x, y) \equiv P(X = x \wedge Y = y)$)

→ Reduce complexity by making use of independencies
X & Y independent from another if: $P(X|Y) = P(X)$, $P(Y|X) = P(Y)$, $P(X, Y) = P(X)P(Y)$
↳ not effected by the other variable → reduces amount of possible values
→ Absolute independent variables are rare i.e. no variables in cancer research are independent

Cancer

Smoking	no	benigne	maligne
no	0.768	0.024	0.008
few	0.132	0.012	0.006
many	0.035	0.010	0.005

$P(y = few) = P(no, few) + P(benigne, few) + P(maligne, few) = 0.15$

Marginalization:

for any set of X and Y → compute probability
↳ $P(Y) = \sum_{i=1}^{n} P(x_i, Y)$
→ resulting distribution → marginal distribution
↳ its probabilities are marginal probabilities

Conditional Probabilities: → probability of $X = x$ under assumption $Y = y$ is true

$P(x|y) = \frac{P(x \wedge y)}{P(y)}$ → alternative formulation Product rule → $P(x, y) = P(x|y)P(y) = P(y|x)P(x)$ → Chain rule derived by successive application of Product rule: $P(x_1, x_2, \ldots x_n) = P(x_1, \ldots, x_{n-1})P(x_n|x_1, \ldots, x_{n-1}) = P(x_1, \ldots, x_{n-2})P(x_{n-1}|x_1, \ldots, x_{n-2})P(x_n|x_1, \ldots, x_{n-1}) = \prod_{i=1}^{n} P(x_i|x_1, \ldots, x_{i-1})$

e.g. assume $P(Y = few) \xrightarrow{\text{calculate}} P(Y = few) = 0.15 \xrightarrow{\text{calculate}} P(X = maligne|Y = few) = \frac{P(maligne, few)}{P(Y = few)} = \frac{0.006}{0.15} = 0.04$
assume $P(X = maligne) \xrightarrow{\text{calculate}} P(maligne) = 0.019 \xrightarrow{\text{calculate}} P(Y = few|X = maligne) = \frac{P(maligne, few)}{P(maligne)} = \frac{0.006}{0.019} = 0.316$

Bayes Rule:

$P(x|y) = \frac{P(y|x)P(x)}{P(y)}$

Probability of hypothesis x after evidence y. Posterior
Probability of evidence y under any circumstance. Marginalization
Probability of hypothesis before considering evidence. Prior
Probability of evidence, given belief is true. Likelihood

e.g. AIDS-Test: Events: Aids = infected or not
Positive = positive test result
Probabilities: P(positive, aids) = 0.99
P(negative, aids) = 0.01
P(positive, ¬aids) = 0.005
P(negative, ¬aids) = 0.995

assume → P(aids) = 0.0001 (Prior) + positive test
Reliable?

$P(a|p) = \frac{P(p|a)P(a)}{P(p)} = \frac{P(p|a)P(a)}{P(p|a)P(a) + P(p|\neg a)P(\neg a)} = \frac{0.99 \cdot 0.0001}{0.99 \cdot 0.0001 + 0.005 \cdot 0.9999} = 0.0194$

Nobel Prize Economics (2002, Amos Tversky & Daniel Kahneman): Uncovered number of biases seeming to characterize human reasoning & decision-making, providing significant challenge to economic models assuming people simply apply statistical decision theory.

Uncertainty in AI:

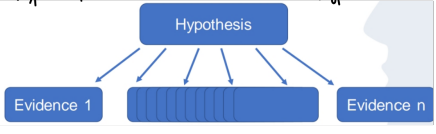
Recall Joint distribution is enumerating everything → Worst-case runtime: $O(2^n)$ (n = # of RVs), Space is $O(2^n)$ → size of joint distribution's table
↳ Main idea: make use of independencies to compress representation

9 Bayesian networks

→ simple, graphical notation for conditional independence assertions, hence for compact specifications of full joint distributions
→ directed acyclic graph with → Nodes (one for each variable) and Edges (directed edge from node N_i to node N_j → indicates X_i has direct influence upon X_j)

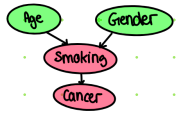
Naïve Bayes model: → assumes all effects are independent given cause

$P(hypothesis, evidence_1, evidence_2, \ldots, evidence_n) = P(hypothesis) \prod_i P(evidence_i|hypothesis)$



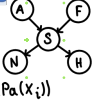
total # of parameters is linear in n.
Graphical encoding of conditional distributions.

(current) age and gender of person are independent



$P(G, A) = P(G) \cdot P(A)$
$P(A|G) = P(A)$
$P(G|A) = P(G)$

Absolute independencies are rare
↳ Less entries and consequently lower complexity
$P(C|S, G, A) = P(C|S)$



Set of random variables $[X_1, \ldots, X_n]$
Directed, acyclic graph (DAG)
To each RV X_i associate conditional probability distribution: $P(X_i|Pa(X_i))$
joint distribution $P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|Pa(X_i))$
↳ BN semantics

Local Markov Assumption
Each RV X independent of its „non-descendant" given its parents $(x_i \perp nonDescendants|Pa_{X_i})$

e.g. Smoking → Cancer

$S \in \{no, few, many\}$ $C \in \{no, benigne, maligne\}$

P(S=n)	0.80
P(S=f)	0.15
P(S=m)	0.05

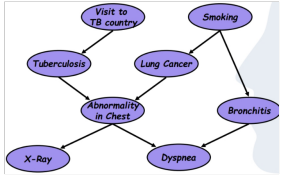
Smoking=	n	f	m
P(C=n)	0.96	0.88	0.60
P(C=b)	0.03	0.08	0.25
P(C=m)	0.01	0.04	0.15

Interference:

Query: $P(X|e)$
Definition of conditional probability $P(X|e) = \frac{P(X, e)}{P(e)}$
Up to normalization $P(X|e) \propto P(X, e)$
↳ rewrites to $P(Y) = \sum_{x_i \notin Y} \prod_{i=1}^{n} P(x_i|Pa(X_i))$
↳ BN semantics

$\sum_a (P_1 \times P_2) = (\sum_a P_1) \times P_2$ ↳ Marginalization if A not in P_2

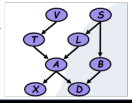
e.g. "Tuberculosis" network:



Variable Elimination:

→ compute P(d)

→ need to eliminate: v, s, x, t, l, a, b



$$P(v)P(s)P(t\mid v)P(l\mid s)P(b\mid s)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$

Eliminate: v

Compute: $f_v(t) = \sum_v P(v)P(t\mid v)$

$$\Rightarrow f_v(t)P(s)P(l\mid s)P(b\mid s)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$

Note: $f_v(t) = P(t)$

In general, result of elimination is not necessarily a probability term

$$P(v)P(s)P(t\mid v)P(l\mid s)P(b\mid s)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$
$$\Rightarrow f_v(t)P(s)P(l\mid s)P(b\mid s)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$

Eliminate: s

Compute: $f_s(b,l) = \sum_s P(s)P(b\mid s)P(l\mid s)$

$$\Rightarrow f_v(t)f_s(b,l)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$

Summing on s results in a factor with two arguments $f_s(b,l)$

In general, result of elimination may be a function of several variables

$$P(v)P(s)P(t\mid v)P(l\mid s)P(b\mid s)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$
$$\Rightarrow f_v(t)P(s)P(l\mid s)P(b\mid s)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$
$$\Rightarrow f_v(t)f_s(b,l)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$

Eliminate: x

Compute: $f_x(a) = \sum_x P(x\mid a)$

$$\Rightarrow f_v(t)f_s(b,l)f_x(a)P(a\mid t,l)P(d\mid a,b)$$

Note: $f_x(a) = 1$ for all values of a !!

$$P(v)P(s)P(t\mid v)P(l\mid s)P(b\mid s)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$
$$\Rightarrow f_v(t)P(s)P(l\mid s)P(b\mid s)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$
$$\Rightarrow f_v(t)f_s(b,l)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$
$$\Rightarrow f_v(t)f_s(b,l)f_x(a)P(a\mid t,l)P(d\mid a,b)$$

Eliminate: t

Compute: $f_t(a,l) = \sum_t f_v(t)P(a\mid t,l)$

$$\Rightarrow f_s(b,l)f_x(a)f_t(a,l)P(d\mid a,b)$$

$$P(v)P(s)P(t\mid v)P(l\mid s)P(b\mid s)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$
$$\Rightarrow f_v(t)P(s)P(l\mid s)P(b\mid s)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$
$$\Rightarrow f_v(t)f_s(b,l)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$
$$\Rightarrow f_v(t)f_s(b,l)f_x(a)P(a\mid t,l)P(d\mid a,b)$$
$$\Rightarrow f_s(b,l)f_x(a)P(d\mid a,b)$$

Eliminate: l

Compute: $f_l(a,b) = \sum_l f_s(b,l)f_t(a,l)$

$$\Rightarrow f_l(a,b)f_x(a)P(d\mid a,b)$$

$$P(v)P(s)P(t\mid v)P(l\mid s)P(b\mid s)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$
$$\Rightarrow f_v(t)P(s)P(l\mid s)P(b\mid s)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$
$$\Rightarrow f_v(t)f_s(b,l)P(a\mid t,l)P(x\mid a)P(d\mid a,b)$$
$$\Rightarrow f_v(t)f_s(b,l)f_x(a)P(a\mid t,l)P(d\mid a,b)$$
$$\Rightarrow f_s(b,l)f_x(a)f_l(a,b)P(d\mid a,b)$$
$$\Rightarrow f_s(b,l)f_x(a)P(d\mid a,b) \Rightarrow f_x(b,d) \Rightarrow f_x(d)$$

Eliminate: a,b

Compute: $f_a(b,d) = \sum_a f_l(a,b)f_x(a)p(d\mid a,b)$ $\quad f_b(d) = \sum_b f_a(b,d)$

Variable elimination:

→ Given: BN and query P(X|e) / P(X,e)

→ Instantiate evidence e

→ choose elimination order over variables, e.g. $X_1, \dots, X_n$

→ Initial factors $\{f_1, \dots, f_n\}$: $f_i = P(X_i \mid Pa_{X_i})$ (CPT for X_i)

for $i = 1$ to n, if $X_i \notin \{X, E\}$

→ Collect factors $f_1, \dots, f_k$ that include X_i

→ Generate new factor by eliminating X_i from these factors

$$g = \sum_{X_i} \prod_{j=1}^{k} f_j$$

→ Variable X_i has been eliminated! Add g to set of factors

Normalize (everything sums to 1) P(X,e) to obtain P(X|e)

Theorem: Inference (even approximate) in BN is NP-hard (#P; via reduction to 3-SAT)

$$\underbrace{(\bar{X}_1 \vee X_2 \vee X_3)}_{C_1} \wedge \underbrace{(\bar{X}_2 \vee X_3 \vee X_4)}_{C_2} \wedge \dots \underbrace{\quad}_{C_n}$$

Approximate Inference: → by stochastic sampling (from a BN)

1. Draw N samples from sampling distribution S
2. Compute approximate posterior probability P
3. Show this converges to the true probability P

Outline:

- Sampling from empty network
- Rejection sampling: reject samples disagreeing with evidence
- Likelihood weighting: use evidence to weight samples
- Markov Chain Monte Carlo (MCMC): Sample from a stochastic process whose stationary distribution is true posterior

Draw sample:

Given: Random variable X, D(X) = [0,1], P(X) = [0.3, 0.7]

Sample $X \leftarrow P(X)$: Draw random number $r \in [0,1]$, if $(r < 0.3)$ set $X = 0$, else $X = 1$ → can generalize of any domain size

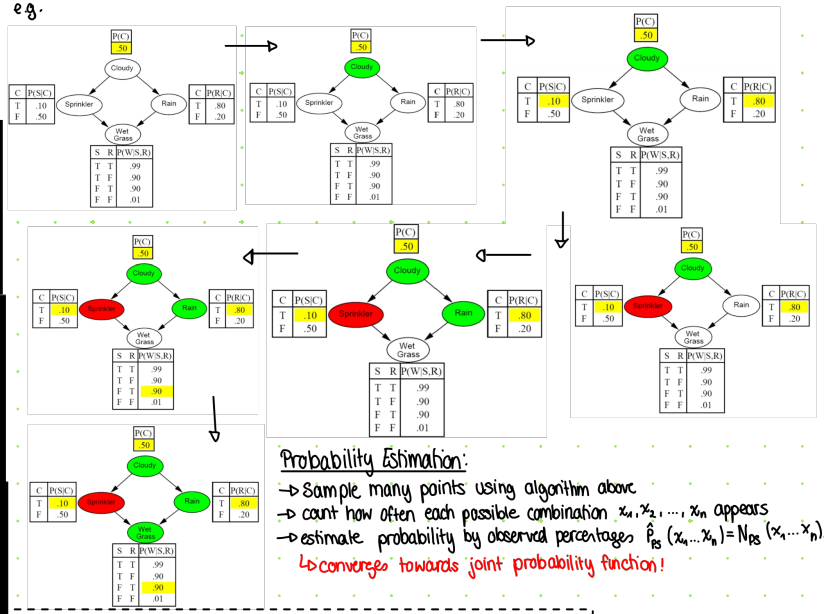
Sampling from an "Empty Network": -Generating samples from network having no evidence associated with it

→ sample value for each variable in topological order

→ using the specified conditional probabilities

function PRIOR-SAMPLE(bn) **returns** an event sampled from bn
  inputs: bn, a belief network specifying joint distribution $\mathbf{P}(X_1, \dots, X_n)$
  $\mathbf{x} \leftarrow$ an event with n elements
  for $i = 1$ **to** n **do**
    $x_i \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
      given the values of $Parents(X_i)$ in $\mathbf{x}$
  return $\mathbf{x}$

e.g.



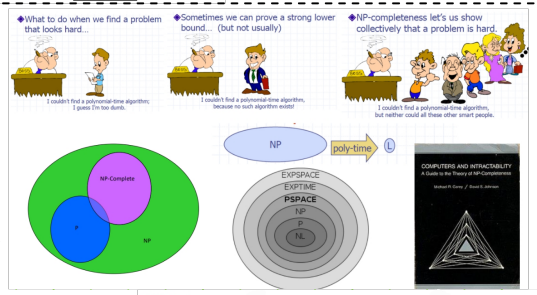
Probability Estimation:

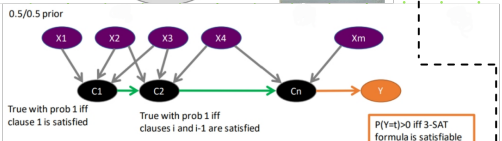
→ Sample many points using algorithm above

→ count how often each possible combination $x_1, x_2, \dots, x_n$ appears

→ estimate probability by observed percentages $\hat{P}_{PS}(x_1 \dots x_n) = N_{PS}(x_1 \dots x_n)/N$

↳ converges towards joint probability function!





Markov Chain Monte Carlo (MCMC) Sampling:

- "State" of network = current assignment to all variables
- Generate next state by sampling one variable given Markov blanket
- Sample each variable in turn, keeping evidence fixed

```
function MCMC-Ask(X, e, bn, N) returns an estimate of P(X|e)
  local variables: N[X], a vector of counts over X, initially zero
                   Z, the nonevidence variables in bn
                   x, the current state of the network, initially copied from e
  initialize x with random values for the variables in Y
  for j = 1 to N do
    for each Z_i in Z do
      sample the value of Z_i in x from P(Z_i|mb(Z_i))
          given the values of MB(Z_i) in x
      N[x] ← N[x] + 1 where x is the value of X in x
  return NORMALIZE(N[X])
```

Gibbs Sampling

- Can also choose a variable to sample at random each time

Ordered Gibbs Sampler:

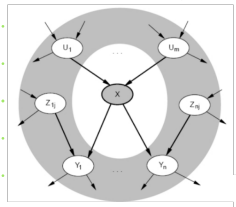
- Generate sample x^{t+1} from x^t: Process all variables in some order

$$X_1 = X_1^{t+1} \leftarrow P(x_1 \mid x_2^t, x_3^t, \dots, x_N^t, e)$$
$$X_2 = X_2^{t+1} \leftarrow P(x_2 \mid x_1^{t+1}, x_3^t, \dots, x_N^t, e)$$
$$\dots$$
$$X_N = X_N^{t+1} \leftarrow P(x_N \mid x_1^{t+1}, x_2^{t+1}, \dots, x_{N-1}^{t+1}, e)$$

in short, for i=1 to N: $X_i = x_i^{t+1} \leftarrow$ sampled from $P(x_i \mid x^t \setminus x_i, e)$

Markov Blanket: = parents + children + children's parents

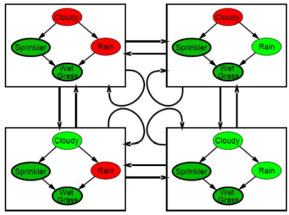
→ Each node conditionally independent of all other nodes given its Markov blanket



$$P(X \mid U_1, \dots, U_m, Y_1, \dots, Y_n, Z_{1j}, \dots, Z_{nj}) = P(X \mid \text{all variables})$$

Markov Chain:

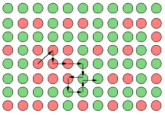
With $Sprinkler = true, WetGrass = true$, there are four states:



Wander about for a while, average what you see

Gibbs Sampling: Illustration

The process of Gibbs sampling can be understood as a *random walk* in the space of all instantiations with $Y = u$:



Reachable in one step: instantiations that differ from current one by value assignment to at most one variable (assume randomized choice of variable X_k).

Guaranteed to converge iff chain is:

- irreducible (every state reachable from every other state)
- aperiodic (returns to state i can occur at irregular times)
- ergodic (returns to every state with probability 1)

Get Probability Distribution from Sampling:

e.g. Task: Estimate P(Rain | Sprinkler = true, WetGrass = true)

1. Sample Cloudy or Rain given Markov Blanket, repeat n times
2. Cant # of times Rain is true and false in sample,

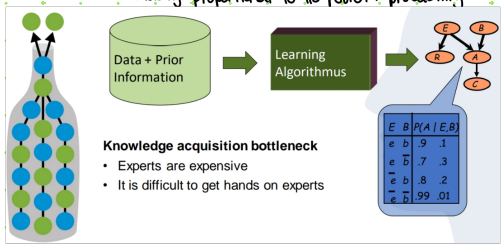
→ 100 states → 31 true, 69 false (Hahn 69)

$$\hat{P}(\text{Rain} \mid \text{Sprinkler} = \text{true}, \text{WetGrass} = \text{true}) = \text{Normalize}(\langle 31, 69 \rangle) = \langle 0.31, 0.69 \rangle$$

Theorem: Chain approaches stationary distribution:
long-run fraction of time spent in each state is exactly proportional to its posterior probability

10 Machine Ethics

↳ nicht klausurrelevant → lesen / Definitionen anschauen



11 Introduction into Machine Learning and Neural Networks

Learning:

1. essential for dealing with unknown environments (agent not omniscient?)
2. useful as system construction method (expose agent to reality)
3. modifies agent decision mechanism to improve performance

("insanity is doing same thing over and over again and expecting different results")

Methods → Memorization (Declarative Knowledge)
- ↳ Accumulation of individual facts
- ↳ Limited by time to observe & memory

→ Generalization (Imperative Knowledge)
- ↳ Deduce new from old facts
- ↳ Limited by accuracy of deduction process (predictive activity, assumes relations between past and future)

Inductive Learning: → Simplest form of learning. It learns function from examples.

→ f is (unknown) target function we want to learn. f(x) is target, label or y

→ Examples defined as (x, f(x)) → (1, Rain)

Problem: find hypothesis h, such that h ≈ f
- ↳ Given training set of examples
- ↳ On all examples

Example:
Day 1: 10h
Day 2: 10 h → f(2) = 10
Day 3: 9 h
Day 4: 2h
....
Day 31: 5h → f(32) = 6

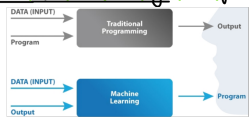
Predicting the future: → construct „rule set" h to agree with f on training set

→ h consistent → agrees with f on all examples

Ockham's Razor → best explanation is simplest explanation fitting the data

Overfitting Avoidance → Maximize combination of consistency and simplicity

1 Machine Learning: → Program algorithm to automatically learn a program from data or experience



→ ML often used as synonym for AI → not the same!
- ↳ AI not always imply learning-based system (Search, CSP, Logical Inference, ...)

→ ML focuses on learning based systems while often extracting knowledge from data

Human learning → (often) data- and knowledge-efficient; complete multitasking, multi-modal system; time-inefficient (takes long)

ML inspired by human learning → not goal to rebuild human learning
→ borrow ideas from biological systems (e.g. neural networks); may perform better/worse than humans

Applications: Web Search, Computational Biology, Speech Recognition → Machine Translations, Image Recognition, Robotics, Finance and Stock Market, Medical Diagnosis, Information Extraction → Visual Analytics, Traffic Prediction, Software development, ...

Designing learning System: ① Do I need learning approach for problem? (pattern?, analytical solving?, training data?)
② Type of problem? (representation? Algorithm choosing) ③ Gather and organize data (Processing important)
④ Fitting/training model ⑤ Optimization ⑥ Evaluate and iterate back to ② → ML not solution to all problems

2 Get to a solution:

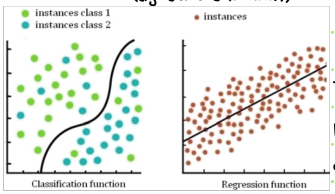
Types of Machine Learning:

- Supervised Learning (based on labeled datasets, learns to map inputs and outputs based on pairs in dataset used in learning process)
- Unsupervised Learning (unlabeled data → searches for patterns and similarities in data)
- Reinforcement Learning (in RL agent learns by interacting with its environment and getting positive or negative reward)

Supervised Learning: Classification and Regression Tasks

→ Given dataset: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ → goal: learn function $h(x)$ to predict y given x

- Regression task: → y is continious value e.g. temperature tomorrow
- Classification task: → y is discrete class label → algorithm tries predicting continious value describing label probability
  ↳ (e.g. over 0°C tomorrow:)



ML algorithms handle lot of different data (e.g. images, audio, ...)
→ Represent input as input vector (in $\mathbb{R}^n$)
↳ great as we can use linear algebra

Representation: Mapping input to another easy manipulatable space

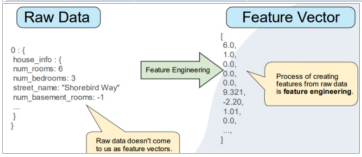
Feature: independent variables in machine learning models

Model: representation of what algorithm learned from daten used in training process. Model is output representation of learned "rule set"

Feature Engineering: process of selecting, manipulating and transforming raw data into features used within learning approach

Label / Features

Example	Weather	Location	Date	Temperature	Precipitation
1	Rainy	Darmstadt	11.04.22	12.3	44.2
2	Sunny	Hamburg	11.04.22	19.2	12.6
3	Rainy	Darmstadt	12.04.22	16.7	67.3
4	Cloudy	Heidelberg	11.04.22	17.3	22.2
...	...	...	...	...	...



Know data: distribution?, outliers?, reflect reality?, biased?, ...

→ using bad data results in bad models → noise per se might not be a problem

ML Common Approaches:

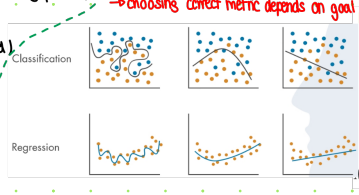
Regression: linear Regression, Multiple linear Regression, Regression Trees, Non-linear Regression, Polynomial Regression, ...

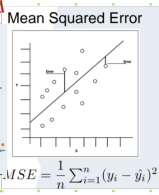
Classification: Random Forest, Decision Trees, Logistic Regression, Naive Bayes, Support Vector Machines, ...

→ no single best model working best for all problems.

3 Evaluating model: → to measure quality → know goal!

→ Accuracy, Precision, Recall, Mean Squared Error, ...
→ choosing correct metric depends on goal

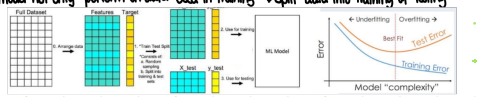




ML Overfitting: model has trained "too well" → memorized dataset while loosing ability to generalize i.e. perform on unknown / new data.

Deal with overfitting: Split data, Regularization, Use more data → augument data i.e. adding noise, Select different features, Cross-validation, Ensemble methods, ...

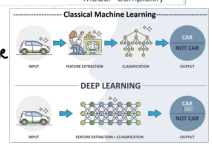
Train Test Split: → model not only perform on data used in training → Split data into Training & Testing



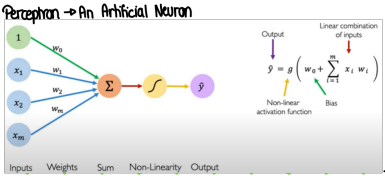
4 Artificial Neural Networks:

Deep Learning: learn underlying features directly from data without specifying them

Why Deep Learning?
↳ Hand-engineered features time consuming, brittle and not scalable in practice

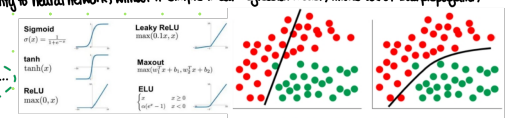


Perceptron → An Artificial Neuron



→ Neurons correspond to nodes or units
→ link from unit j to i propagates activation y from j to i
→ weight $w_{j,i}$ of link determines strength and sign of connection
→ all weights together → W / θ describe model
→ total input activation → sum of input activations
→ output activation determined by activation function g

Activation functions: → decides if neuron should be active → nowadays mostly non-linear function
↳ Adds non-linearity to neural network, without it simple linear regression model, Allows use of backpropagation

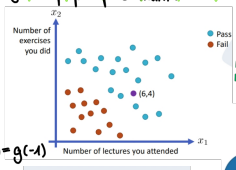


From perceptron to neural network

→ Perceptrons may have multiple output nodes → output nodes can be combined with other perceptrons
→ Build networks of these nodes i.e. Multilayer Perceptrons (MLP) → information flow unidirectional
→ Information distributed and processed in parallel
→ Use size (i.e # of layers) to model expressivness of MLP
→ Following definition of perceptron we know for each hidden node $z_{k,i} = \sigma(\omega_{0,i}^{(k)} + \sum_{j=1}^{s_{k-1}} z_{k-1,j} \cdot \omega_{j,i}^{(k)})$

5 Prediction or Forward Propagation:

e.g. Will purple pass exam?



Assume input: $x_1 = 6$, $x_2 = 4$

Weights: $\omega_{1,1}^{(1)} = 0.5$ $\omega_{1,2}^{(1)} = -1$ $\omega_{1,3}^{(1)} = -1$
$\omega_{2,1}^{(1)} = 0.\overline{33}$ $\omega_{2,2}^{(1)} = 0.5$ $\omega_{2,3}^{(1)} = -0.8$
$\omega_{3,1}^{(1)} = 0.1$ $\omega_{3,2}^{(1)} = 0.6$ $\omega_{3,3}^{(1)} = 2$

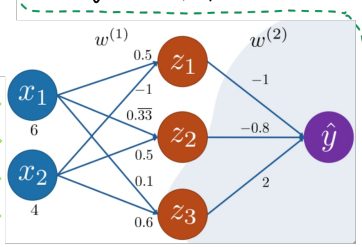
Assume bias $\omega_{0,i}^{(k)} = 0$

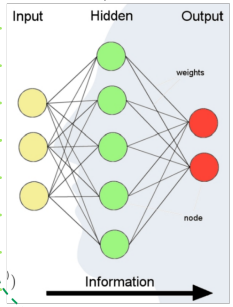
$\Rightarrow z_1 = g(\omega_{0,1}^{(1)} + \sum_{j=1}^{2} x_j \cdot \omega_{j,1}^{(1)}) = g(x_1\omega_{1,1}^{(1)} + x_2\omega_{2,1}^{(1)}) = g(3 + (-1)) = g(-1)$
$z_2 = g(1 + 2) = g(4)$
$z_3 = g(0.6 + 2.4) = g(3)$

Use sigmoid activation function: $\sigma(z) = \frac{1}{1+e^{-z}}$ →

$\Rightarrow z_1 = 0.26$, $z_2 = 0.98$, $z_3 = 0.95$

$\hat{y} = \sigma(\omega_{0,1}^{(2)} + \sum_{j=1}^{3} z_j \cdot \omega_{j,1}^{(2)}) = \sigma(\omega_{0,1}^{(2)} + z_1\omega_{1,1}^{(2)} + z_2\omega_{2,1}^{(2)} + z_3\omega_{3,1}^{(2)})$
$\hat{y} = \sigma((-1) \cdot 0.26 + (-0.8) \cdot 0.98 + 2 \cdot 0.95) = \sigma(0.856) = 0.7$ → Yes purple should pass



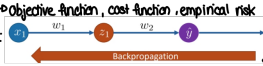


6 Training a NN or Backpropagation:

- Assume instead of 0.7 → 0.14 (fails exam) → How do we measure quality?

Loss function: Quantifying Loss $\mathcal{L}(f(x^{(i)}; W), y^{(i)})$ → cost of incorrect predictions
Empirical Loss $J(W) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; W), y^{(i)})$ → Measures total loss over dataset
↳ objective function, cost function, empirical risk

Compute $\frac{\partial J(W)}{\partial W}$:



$\frac{\partial J(W)}{\partial \omega_i}$

→ How much do weights affect outcome i.e. final loss?

Using chain rule describing problem: $\frac{\partial J(W)}{\partial \omega_2} = \frac{\partial J(W)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \omega_2}$ $\frac{\partial J(W)}{\partial \omega_1} = \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} \frac{\partial z_1}{\partial \omega_1}$

↳ repeat for every weight in network using gradients from later layers

→ Propagate the error back to all nodes, through the network

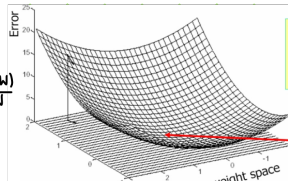
Training Network:

Overall goal: Minimize loss $W^* = \arg\min_W \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; W), y^{(i)})$

Gradient descent

Algorithm:

1. Initialize weights randomly
2. Loop until convergence
   → Compute gradient: $\frac{\partial J(W)}{\partial W}$
   → Update weights: $W \leftarrow W - \alpha \frac{\partial J(W)}{\partial W}$
3. Return Weights



Weight space is N-dimensional, where N is the total number of weights in the network

W^* describes the weight setting where the error or loss is minimal for the data set

Example: $x_1 \xrightarrow{0.5} y \xrightarrow{2} \cdots \rightarrow J(W)$

Given: $g() = \text{ReLU}(x) = \max(0, x)$, $x = 4$, $y = 1$, $\hat{y} = 2$, $\mathcal{L} = (\hat{y} - y)^2$, $J(W) = \mathcal{L}$

Wanted: $\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_1}$

Step 1: $\frac{\partial J(W)}{\partial \hat{y}} = 2(\hat{y} - y) = 2(2-1) = 2$

Step 2: $\frac{\partial \hat{y}}{\partial w_1} = \text{ReLU}'(w_0 + \sum_{i=1}^{n} w_i x_i) \cdot x_1 = 1 \cdot 4 = 4$

Step 3: $\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_1} = 2 \cdot 4 = 8$

Update the Weights: $W \leftarrow W - \alpha \frac{\partial J(W)}{\partial W}$

Step 4: $W_{new} = W_{current} - \alpha \frac{\partial J(W)}{\partial W}$ $\quad \alpha = 0.05$

$W_{new} = 0.5 - 0.05 \cdot 8$

$W_{new} = 0.1$

Alternative Step 4: $W_{new} = 0.5 - 0.5 \cdot 8$ $\quad \alpha = 0.5$

$W_{new} = -3.5$

→ explanation video link slides 51-53

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

$$\hat{y} = \text{ReLU}(w_1 x_1)$$

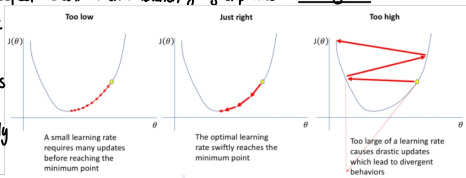
$$f = g(h(x))$$
$$f' = g'(h(x)) \cdot h'(x)$$

$$\text{ReLU}'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

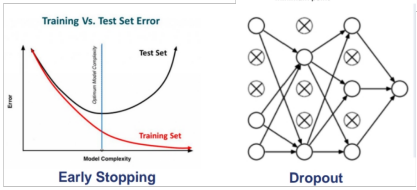
▸ Loss function can be difficult to optimize → Hard to find global minimum

→ Change weights into direction of steepest descent of error function giving step size → learning rate (α)

↳ Difficult finding a good learning rate



Regularization: Set of techniques to prevent overfitting in neural networks → improve accuracy of a Deep Learning model when facing completely new data from problem domain



7 What's next:

Further Architectures: MLP only first step in field of neural network architectures → How many layers? Complex inputs (images)? # features use? Use images and text as inputs at the same time? How can I do sequences of data e.g. sentences, time-series, ...? ...

Convolutional Neural Network (CNN): uses convolutional layers to extract features from input

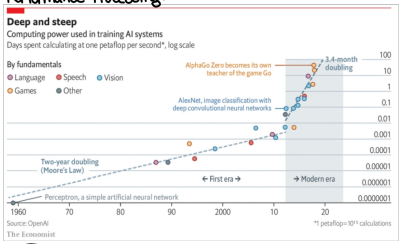
→ Features in first layers refer to edges, borders, shapes, ...

→ higher levels: pattern detection up to specific objects

→ uses pooling layers to decrease computational requirements and extract more dominant features

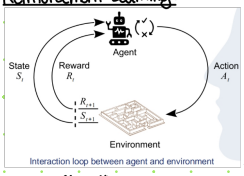
→ Compared to MLPs: fewer connections, i.e. weights, easier to train, can have a lot of layers, popular in fields like Computer Vision and NLP

Performance Processing:



(12) Introduction into Reinforcement Learning and AlphaZero

Reinforcement Learning:



→ RL algorithms attempt to find policy (what action in which state) for maximizing cumulative reward for agent over course of problem

→ Mathematically, typically represented by a Markov Decision Process (MDP)

→ RL differs from supervised learning in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected

Determine how system's performance success due to contribution of components

In RL, i.e. temporal CAP: → Given sequence of actions & states, final sum of time-discounted future rewards, how infer which actions were effective at producing lots of reward and which actions were not effective?

→ How assign credit for observed rewards given sequence of actions over time?

→ Every reinforcement learning algorithm must address this problem

Example: Fruit Fly

→ agent lives in a grid & walls block agent's path

→ agent's actions are noise, i.e. don't always go as planned

↳ 80% action North takes agent North

↳ 10% " " " West, 10% East (Stochastic movement)

↳ if there is wall → no movement

→ Small "living" reward each step → Big rewards come at the end → Goal: maximize sum of rewards

Deterministic grid world: 100% taking action supposed to take

Stochastic grid world: Because of noise multiple actions could happen

Markov Decision Processes:

Markov(ian) → given present state, future and past are independent

MDP defined by 4d-tuple (S, A, T_a, R_a) → S: a set of states $s \in S$

↳ start state (distribution)

↳ (optimal) terminal state

↳ discount factor: γ

A: a set of actions $a \in A$

T_a: a transition function $T(s, a, s')$ → Probability that a from s leads to s' i.e. $P(s' | s, a)$ → model

R_a: a reward function $R(s, a, s')$ → sometimes just $R(s)$ / $R(s')$

→ MDPs are a family of non-deterministic search problems

→ RL: MDPs where we don't know transition or reward functions

$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3$ with r_0, r_1, r_2

→ for Mdp → „Markov" = $P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots, S_0 = s_0)$

$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$

Solve MDPs: finding optimal plan

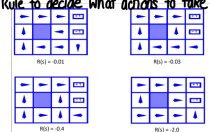
▸ in deterministic single-agent search problems we want optimal plan, or sequence of actions, from start to goal

▸ in MDP compute optimal policy $\pi^*: S \rightarrow A$: π gives an action for each state, optimal policy maximizes expected utility if followed, defines reflex agent

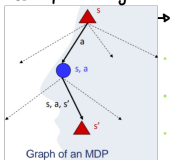
▸ Policy π: Rule to decide what actions to take

$a_t = \mu(s_t)$ Deterministic policy

$a_t \sim \pi(\cdot | s_t)$ Stochastic policy



Depending on the action rewards/costs, optimal policies may look differently



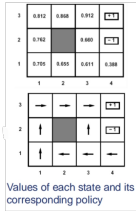
Intuition of Action Selection:

At time t, agent tries to select an action to maximize the sum G_t of discounted rewards received in the future. $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$

- future rewards worth less than an immediate reward. Helps ensuring convergence of expectations.
- Discount factor $\gamma \leq 1$ (often $\gamma = 0.9$) → assume reward n years in future is only worth $(\gamma)^n$ of the value of immediate reward → $(0.9)^6 * 10{,}000 = 0.531 * 10{,}000 = 5311$
- For each state → calculate utility value equal to the sum of future Discounted Rewards

Optimal Utilities: Fundamental operation of computing the values (optimal expectimax utilities) of state s (based on the rewards)

- Optimal values define optimal policies!
- Value function: defines value of state s: $V^*(s)$ = expected utility starting in s and thereafter acting optimally
- Action-Value function: defines value of a q-state (s,a): $Q^*(s,a)$ = expected utility starting in s, taking action a and thereafter acting optimally
- Define optimal policy: $\pi^*(s)$ = optimal action from state s



Values of each state and its corresponding policy

Optimality:

- Policy defined to be better than or equal to another policy if its expected return is greater than / equal to that of the other policy for all states
- Always at least one optimal policy π^* that is better than or equal to all other policies
- All optimal policies share same optimal state-value function V^* → gives max. expected return for any state s over all possible policies
- All optimal policies " " " action-value function q^* → " " " " " " state-action pair (s,a) over all possible policies

Formalization: $V^\pi(s) = R(s,\pi(s)) + \gamma \sum_{s'} T(s' \mid s,\pi(s)) V^\pi(s')$ **Bellman Equation of Optimality**

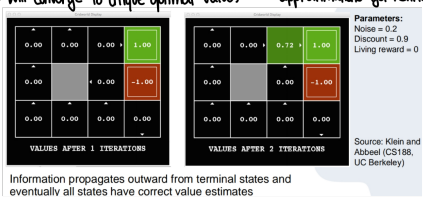
- V^π: value of state s when acting according to π
- $R(s,\pi(s))$: immediate reward
- γ: discount factor
- $T(s' \mid s,\pi(s))$: transition probability to arrive in s'

Dynamic Programming: Value Iteration:

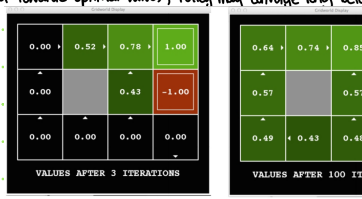
- Start with $V_0^*(s) = 0$
- Given V_i^*, calculate values for all states for depth $i+1$: $V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$ → value update/ Bellman update
- Repeat until convergence
- Complexity of each iteration: $O(S^2A)$

Theorem: will converge to unique optimal values → approximations get refined towards optimal values, Policy may converge long before values do

Example:



Information propagates outward from terminal states and eventually all states have correct value estimates



RL assumes an MDP: set of states $s \in S$, set of actions (per state) A, model $T(s,a,s')$, reward function $R(s,a,s')$, discount factor γ (could be 1) → seek for policy $\pi(s)$

↳ we don't know $T(s,a,s')$ or $R(s,a,s')$ i.e. don't know which states are good or what actions do → must actually try actions and states out to learn

Model-based approach RL → learn (use) model and use it to derive optimal policy
Model-free approach RL → Derive optimal policy without learning model
On-policy RL → learns only about policy it's executing
Off-policy RL → learns about a policy or policies different from the one it's executing
Passive Learning → agent watches world going by → tries to learn utilities of being in various states → goal: execute fixed policy and evaluate it
Active Learning → agent watches and acts → goal: act and learn optimal policy

Passive Learning:

Simplified task:
- You don't know transition $T(s,a,s')$
- You don't know rewards $R(s,a,s')$
- You are given a policy $\pi(s)$
- Goal: learn state values of policy

→ Learner "along for the ride"
→ No choice about what actions to take
→ Just execute policy and learn from experience
→ get to active learning soon
→ NOT offline planning! → take actions in world and see what happens

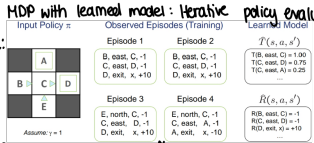
Model-Based Learning:

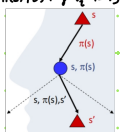
→ Learn model empirically through experience → Solve for values as if learned model were correct

Simple empirical model learning: Count outcomes for each s,a → Normalize to give estimate of $T(s,a,s')$
→ Discover $R(s,a,s')$ when we experience (s,a,s')

Solving MDP with learned model: Iterative policy evaluation e.g. $V_{i+1}^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_i^\pi(s')]$

Example:





Graph of an MDP

Sample-Based Policy Evaluation → improve estimate of V by computing these averages: $V_{i+1}^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_i^\pi(s')]$

↳ Take samples of outcomes s' (by doing action!) and average

$\text{sample}_1 = R(s,\pi(s),s'_1) + \gamma V_k^\pi(s'_1)$
$\text{sample}_2 = R(s,\pi(s),s'_2) + \gamma V_k^\pi(s'_2)$
$\text{sample}_n = R(s,\pi(s),s'_n) + \gamma V_k^\pi(s'_n)$

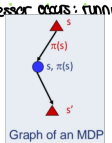
$V_{k+1}^\pi(s) \leftarrow \frac{1}{n} \sum_i \text{sample}_i$

Temporal Difference (TD) Learning: → learn from every experience!

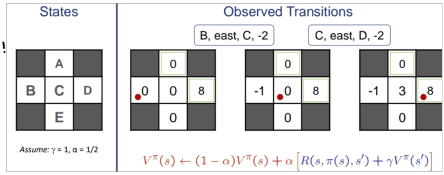
↳ Update V(s) each time we experience (s,a,s',r)
↳ Likely s' will contribute updates more often

=> Policy still fixed! Move values toward value of whatever successor occurs: running average!

Sample of V(s): $\text{sample} = R(s,\pi(s),s') + \gamma V^\pi(s')$
Update to V(s): $V^\pi(s) \leftarrow (1-\alpha)V^\pi(s) + (\alpha)\text{sample}$
Same update: $V^\pi(s) \leftarrow V^\pi(s) + \alpha(\text{sample} - V^\pi(s))$
(Explains term TD)



Graph of an MDP



Problems: TD value learning is a model-free way to do policy evaluation. However, if want to turn values into a (new) policy, sink:

$\pi(s) = \arg\max_a Q(s,a)$
$Q(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V(s')]$

→ learn Q-values directly
→ Makes action selection model-free too!

Active Learning:

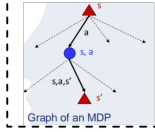
Full reinforcement learning: don't know transitions $T(s,a,s')$, rewards $R(s,a,s')$
can choose any action
Goal: learn optimal policy

→ learner makes choices! → Fundamental trade-off: exploration vs. exploitation
→ NOT offline planning! → take actions and find out what happens

Q-Learning: → sample based Q-value iteration

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right]$$

Learn $Q^*(s,a)$ values: → Receive sample (s,a,s',r)
→ Consider old estimate: $Q(s,a)$
→ Consider new sample estimate:
sample $= R(s,a,s') + \gamma \max_{a'} Q(s',a')$
→ Incorporate new estimate into running average $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)[\text{sample}]$



Graph of an MDP

Exploration / Exploitation:

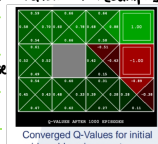
→ Several schemes for forcing exploration (Simplest: random actions (ε-greedy))
↳ every time step, flip a coin; probability ε, act randomly; probability $1-\varepsilon$, act according current policy

Problems: explore space but keep thrashing around once learning done
↳ solution: lower ε over time or exploration functions

Properties:

Amazing result: Q-learning converges to optimal policy
↳ explore enough, learning rate small enough ... but not decrease too quickly → basically, doesn't matter how you select actions

Problems: → keeps table of all q-values → not possible to learn about every state realistically, too many states to visit in training & to hold in q-tables in memory



Converged Q-Values for initial grid-world environment
Source: Klein and Abbeel (CS188, UC Berkeley)

Deep Q-Networks (DQN) → Q-learning with deep neural function approximator (Q-network)
↳ discrete and finite set of actions A, uses epsilon-greedy policy to select actions

Overview → so far:

Things we know	Techniques
if we know MDP	▷ Model-based DPs
→ compute V^*, Q^*, π^* exactly	↳ Value and policy iteration
→ evaluate a fixed policy π	↳ Policy evaluation
if we don't know MDP	▷ Model-based RL
→ we can't estimate MDP then solve	▷ Model-free RL
→ we can estimate V for a fixed policy π	↳ Value learning
→ we can estimate $Q^*(s,a)$ for optimal policy while executing exploration policy	↳ Q-learning

Policy Search:

Problem: often well working feature-based policies don't approximate V/Q best
Solution: learn policy that maximizes rather than predicts rewards
→ May parameterize policy and run some gradient optimization e.g. REINFORCE

$$\pi(a|s,\theta) \doteq \frac{\exp(h(s,a,\theta))}{\sum_b \exp(h(s,b,\theta))} \qquad \theta_{t+1} \doteq \theta_t + \alpha \gamma^t G_t \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$$

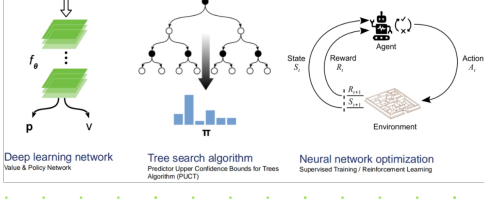
Simplest policy search: → start with initial linear value function or q-function
→ nudge each feature weight up and down and see if policy better than before

Problems: → How judge if policy better?
↳ run many sample episodes! → impractical if there are many features

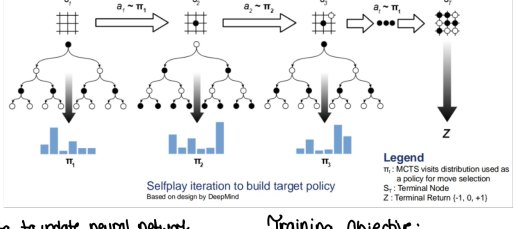
Deep neural networks:

→ deep policy network trained to produce probability map of promising moves. Deep value network used to prune (many) search tree

Components of AlphaZero:

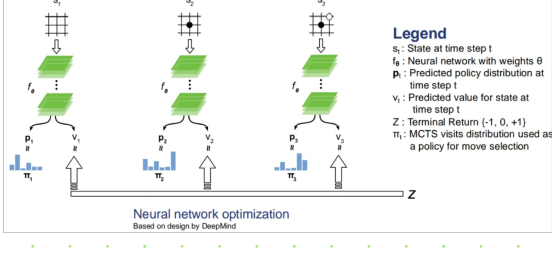


Combining Tree Search with Neural Networks:



Generating games as training data

Using Search as an Improvement Operator: → using generated data to update neural network



Training Objective:

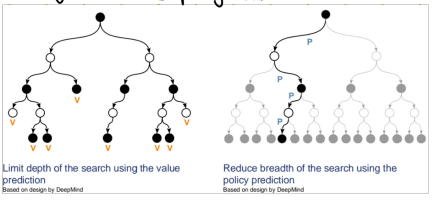
Loss formulization: $L = \alpha(z-v)^2 - \pi^T \log p + c\|\theta\|^2$

- α: value loss factor
- z: target value
- v: predicted value
- π: MCTS simulation distribution
- p: policy head output
- c: L_2 regularization constant

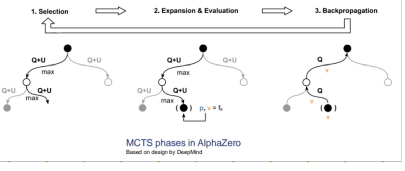
$\alpha(z-v)^2$: Mean Squared Error
$-\pi^T \log p$: Cross Entropy
$c\|\theta\|^2$: Regularizer

Search Principles of AlphaZero:

↳ using shared value / policy neural network



Monte-Carlo Tree Search (MCTS):



PUCT-Algorithm: → Predictor Upper Confidence Bounds for Tree Algorithm (PUCT)

→ Exploration by rollouts

→ Move selection: $a_t = \text{argmax}_a(Q(s_t,a) + U(s_t,a))$, where $U(s_t,a) = c_{puct}\, P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$

→ Update Q-values by simple moving average (SMA): $Q'(s_t,a) \leftarrow Q(s_t,a) + \frac{1}{n}[v - Q(s_t,a)]$

c_{puct}: scalar value to manage exploration

$U(s_t,a)$: utility values

$N(s_t,a)$: number of visits of action a

$Q(s_t,a)$: state action values

s_t: state s at time step t

a: action a

v: predicted value by neural network in $(-1,+1)$