I Search

- Informed search: uses evaluation function f to choose which nodes to expand
- Heuristic search: informed search but f is an estimate which contains heuristic h
  - (e.g.) Straight line distance

↓

Computing h(node) is usually much easier.

Greedy best-first search: make a locally optimal choice based on a heuristic h

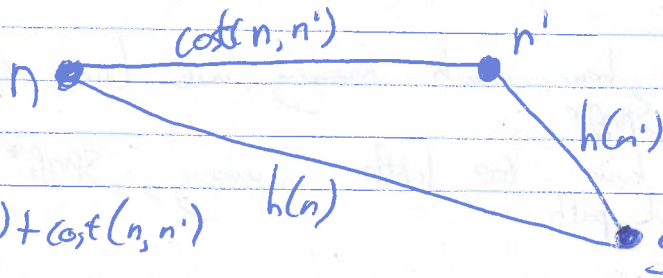
Problems: - can get trapped in local minimum
- can get trapped in endless cycle
- can terminate with sub optimal path

A* search: - uses h(node)
- uses the cost to get to node n from the starting point

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

usually 1

Admissible: h never overestimates
Consistent: h satisfies the triangle inequality:



$h(n) \leq h(n') + cost(n, n')$

A consistent heuristic is always admissible but not vice versa

A* search has a space complexity of $O(b^m)$

↓

We can get a linear complexity by using IDA*

Iterative deepening A*

→ keeps at most 1 path in memory ~~at the cost~~

Iterative depth first searches to an increasing limit.

→ Normal Iterative deepening DFS: increase depth limit by 1 until a solution is found

↓

IDA: depth limit is the path length f_{lim}
↳ Nodes with $f > f_{lim}$ will not be expanded.

Initial: $f_{lim} = f(\text{start-node}) = h(\text{start-node})$
Then: f_{lim} is increased to the lowest $f(\text{node}) > f_{lim}$

Tree-search A* is optimal if h is admissible.

Graph-search A* is optimal if h is consistent.

Simplified Memory-bounded A* (SMA*)

→ Used if we are very thight on memory or if we have more memory than IDA* uses.

↓

If we know how much memory we have available, we can use SMA*
↳ if we have too little memory, SMA* will not find an optimal path

→ The current f value of each node is continuously updated
  → unexpanded/putially expanded: $f(n) = g(n) + h(n)$
  → fully expanded: $f(n)$ = lowest f-value of any descendant
  → non-goal nodes for which path exceeds memory limit: $f(n) = \infty$

→ Nodes also store the lowest f-value of any forgotten descendant
→ If memory limit is reached, node with highest f-value is dropped
  ↓
  lowest f-value of children stored at parent node

→ When all actual $f(n)$ > forgotten $f(n)$, the node is re-expanded.

Minimal spanning trees: useful for finding the cheapest way to connect N vertices given possible connections between them.

Can also be used for cheapest path problems
↳ Some MSTs are paths

MST finding algorithms:

- Prim → Pick the minimal edge from the connected subset
- Kruskal → Add lowest cost edges without making a loop
  ↓

Greedy algorithms

II ~~Kostalin algorithm~~

Local search : do not search the entire problem space
↳ Start with a random "possible solution" and tweak it
↓ ↓
or chosen alter some variable

Restart: changing the entire set of variables
Walk: changing a few variables at a time

→ NOT complete (not guaranteed to find the optimal solution)

→ Local search is very inefficient
→ We need a ~~s~~something to make it more efficient
↓
evaluation function

Iterative best improvement

→ Much like greedy best-first search

1. We find the neighbour states of the current state (states with a one variable difference)
2. Select the neighbour with the best evaluation
3. Change the state to the neighbour state

→ Iterate

→ This method can get stuck in a local optimum
↳ No better neighbors and no adequate solution
↓
Can be fixed by random restarts

Simulated annealing: iterative improvement with decreasing shakes

→ We shake a lot in the beginning and hope to get out of large valleys
→ We shake less late on and hope to get out of all subvalleys

NOTE: only walks of decreasing randomness, no restart

1. Start with a random initialization
2. Pick a variable at random and change it
3. If this new assignment is better, change the current state
4. If this solution is worse, change the current state with a random probability

Probability for accepting worsening steps:

$$p(\text{accept}) = e^{(h(A)-h(A'))/T}$$

- $h(A)$ = evaluation of the state
- $h(A')$ = evaluation of the new assignment
- T = temperature

If T is high, the probability of acceptance is also high

Maximum flow networks

Flow network: Edges with weights, nodes, a source and a sink node

↓

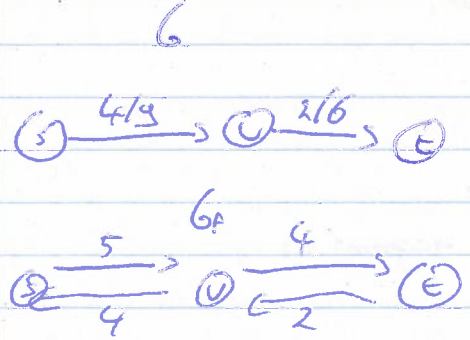
Directed graph

Maximum flow: maximum amount of outgoing and incoming flow from s to t

Ford-Fulkerson method

1. Start with $f(u,v) = 0$ for all $u,v \in E$
2. Iteratively increase $|f|$ by some augmenting path in the residual network G_f
3. Stop if G_f does not contain any augmenting paths anymore

Residual network:



Augmenting path: simple path in G_f from s to t

Cut: partition of V into S and T such that $s \in S$ and $t \in T$

Net flow: flow across cut (S,T)

Capacity of $c(S,T)$ is the maximum flow $S \to T$

Minimum cut: cut with the lowest capacity of all cuts

III Complexity analysis

Complexity of an algorithm: Function relating the input length to the number of steps or storage locations

Time complexity: numbers of lines of code executed

Worst case: upper bound
Average case: average
Best case: lower bound

We are not interested in the exact number of instructions
↓
We use the rate of growth

→ Big-Oh notation (asymptotic upper bound on running time.

Notation:

$f(n) = O(g(n))$

↳ f does not outgrow g modulo a constant factor or scalar

$f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

Constant time: $O(1)$
Logarithmic time: $O(\log_2(n))$
Polynomial time: $O(n^c)$ for some $c > 0$
Exponential time: $O(c^n)$ for some $c > 1$

Decision problems: problems that have yes or no as an answer

Complexity class P: decision problems that can be decided in time polynomial in the input size.

Complexity class NP: decision problems that can be verified in time polynomial in the input size

$P \subseteq NP$

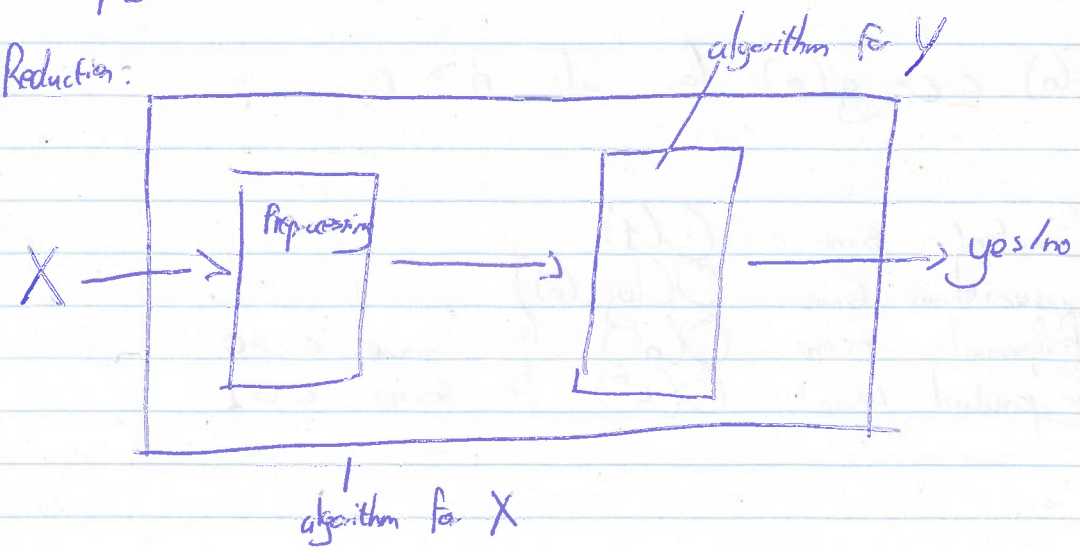
NP-hard: a problem that is at least as hard as any other problem in NP modulo a polynomial transformation

NP-complete: a problem is NP-complete if it is in NP and is in NP-hard

Proving a problem is in NP-hard:

→ Transform problem X to a problem Y that is NP-hard in polynomial time

Reduction:



IV Planning

CSP (Constraint satisfaction problem)

- Variables
- Domains (one for each variable)
- Constraints (predicate over the variables)
- Possible worlds

Eg. A sudoku puzzle:

- Cells that we can fill in (variables)
- Numbers of from 1-9 (domains)
- Restrictions on the values (constraints)
- Possible ways to assign numbers (possible worlds)

CSP solving:

1. list all possible worlds
2. For each world check the constraints

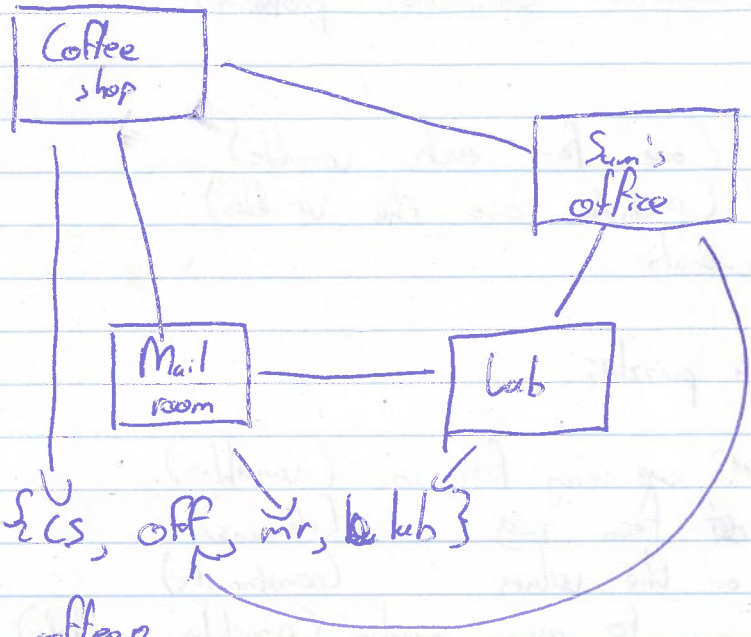
↳ Extremely time consuming

Planning: we want to find a sequence of actions that is: - possible
- result in a state satisfying the goal

→ Starting from an empty state

vb. Delivery robot example

Environment:



Features: Location
Domain: {cs, off, mr, lab}

(RHC) Rob has coffee
Domain: {true, false}

(SWE) Sam wants coffee
Domain: {true, false}

(MW) Mail is waiting
Domain: {true, false}

(RHM) Rob has mail
Domain: {true, false}

Actions:
- Move clockwise (mc)
- Move anti-clockwise (mac)
- Pick up coffee (PUC) - must be at cs
- Deliver coffee (DelC) - must be at off and RHC
- Pick up mail (PUM) - must be in mr and MW
- Deliver mail (DelM) - must be in off and RHM

Solve a planning problem: - See it as a pure search problem
- See it as a CSP problem

I Planning: intelligent decision making

Solution to a planning problem: paths from an initial state to a goal state

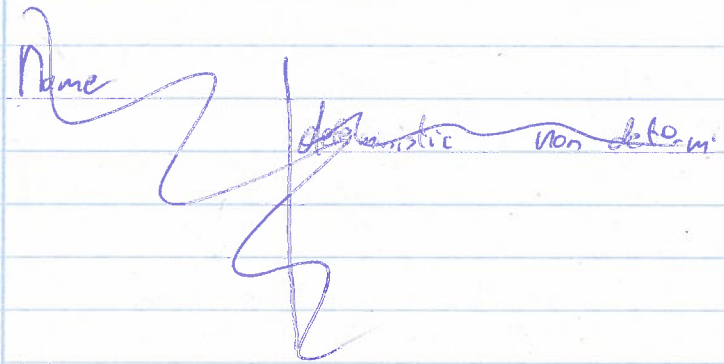
↓

Solvable by Dijkstra but is intractable for larger state spaces

constructing the transition graph

→ Planning algorithms try to avoid constructing whole graph

Classes of problems



1. Classical planning

Dynamics: deterministic
Observability: full
Horizon: finite

2. Conditional planning with full observability

Dynamics: nondeterministic
Observability: full
Horizon: finite

3. Conditional planning with partial observability

Dynamics: nondeterministic
Observability: partial
Horizon: finite or infinite

4. Conformant planning

Dynamics: deterministic or nondeterministic
Observability: none
Horizon: finite

5. Markov decision processes (MDP)

Dynamics: probabilistic
Observability: full
Horizon: finite or infinite

6. Partially observable MDPs

Dynamics: probabilistic
Observability: partial
Horizon: finite or infinite

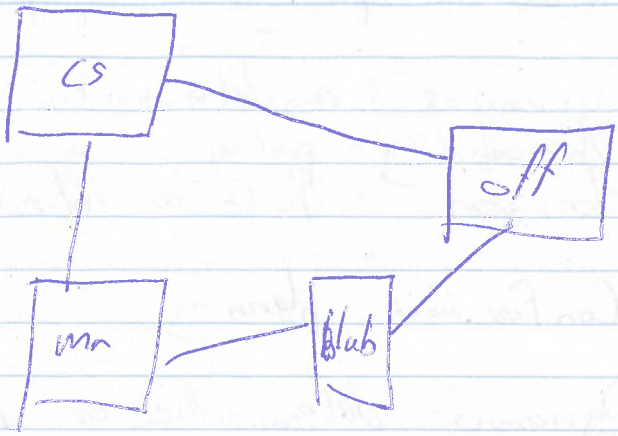
Explicit state representation: explicitly enumerate each state

4b. Delivery robot domain:

RLoc: {lab, cs, off, mr}
RHC: {rhc, $\overline{rhc}$}
SWC: {swc, $\overline{swc}$}
MW: {mw, $\overline{mw}$}
RHM: {rhm, $\overline{rhm}$}

State is a five tuple of the above variables

Actions:
- mc
- mcc
- pac
- dc
- pum
- dm



For each state specify the actions possible and the resulting state from that action.

↓

State-space graph

Complexity: - finding a solution: polynomial
- finding the shortest solution: NP-hard

Feature based representation of actions: preconditions and effects

Causal rules: specify when a feature gets a new value

eg. $Rloc' = cs \leftarrow Rloc = off \wedge Act = mcc$

Frame rules: specify when a feature keeps its value

$Rloc' = cs \leftarrow Rloc = cs \wedge Act \neq mc \wedge Act \neq mcc$

Action centered representation: STRIPS

↓

Stan Ford research institution problem solver

Representation: - precondition
- effect

pac: - precondition: [cs, rhc]
- effect : [rhc]

If the effect is a boolean list:

effect := add-list & delete-list

→ Features not mentioned stay unchanged

STRIPS representation: - initial state
- goal state
- variables
- actions

Two types of goals: - Achievement goal (true in the final state)
- Maintenance goal (true in all states)

Two types of planning: - satisficing (any solution)
- optimal (optimal solution)

Forward planning: searches the state-space graph from the initial state

→ The relevant part of the graph can be created dynamically

Regression planning: searches the state-space graph from the goal states

VI CSP Planning

CSP: constraint satisfaction problem

Action features: variables over actions

PUC : Boolean
DelC : Boolean
...
Move : domain ({mc, mcc, nm})

} Robot

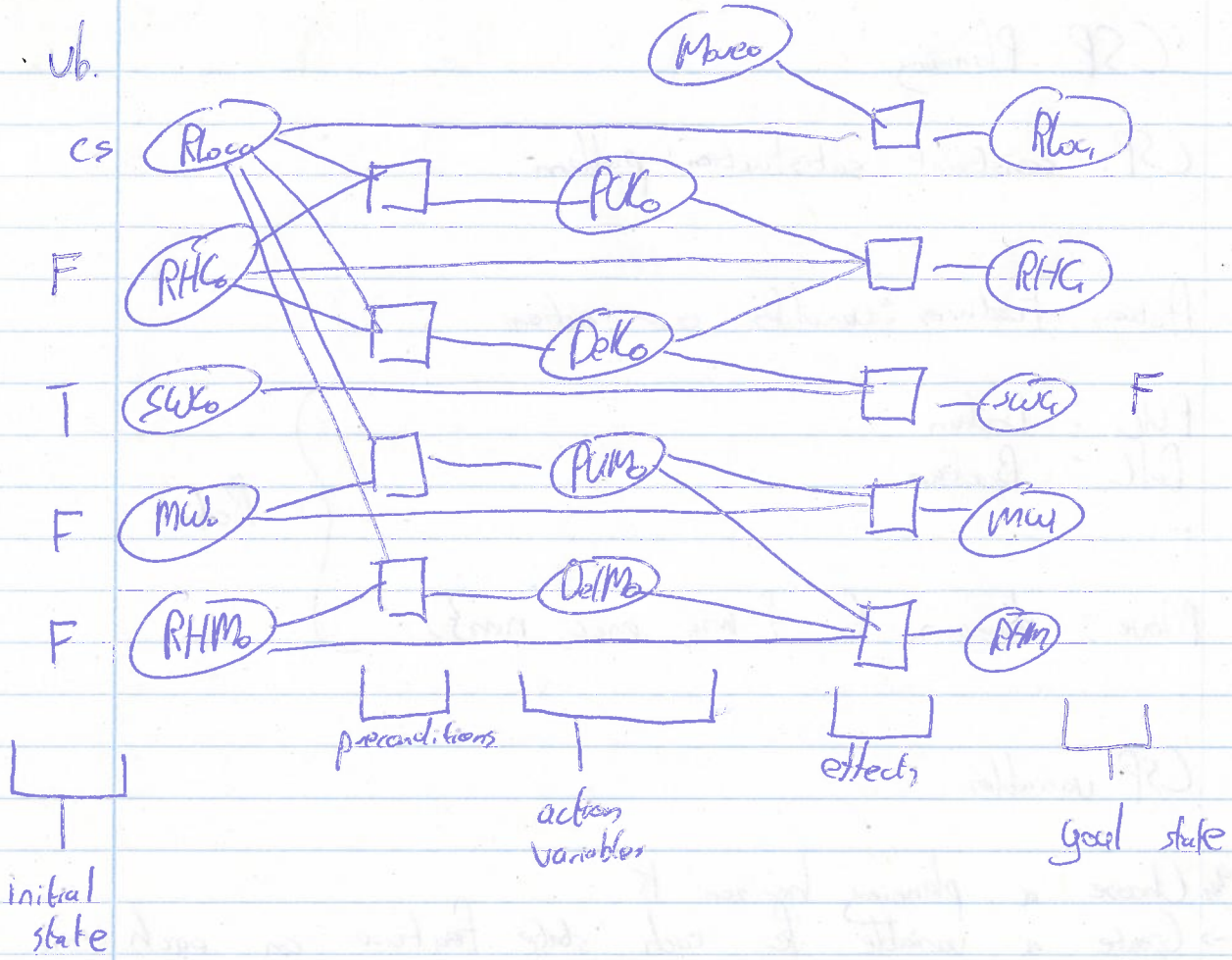
CSP variables

- Choose a planning horizon k
- → Create a variable to each state feature an each time from 0 to k
- → Create a variable for each action feature for each time in the range 0 to k-1

Constraint types:

- State constraint : constraint between variables in the same time step
- Precondition constraint : constraints between states at time t and actions at time t (what action is available at t?)
- Effect constraints : between state variables at time t, action variables at time t and state variables at time t+1
- Action constraint : Constraints between actions at time t (which actions can't co-occur)
- Initial state constraint : values of variables at time 0
- Goal constraint : values of variables at time t

Ub.



Solving a CSP can be done by logically conducting steps.

Mutex relations: two actions that can't co-occur:

- One action negates the effects of the other
- One of the effects of one action is the negation of a precondition of the other
- One of the preconditions of one action is mutex with a precondition of the other

Two literals that can't co-occur:

- One literal is the negation of the other
- Each pair of actions that could achieve the two literals is mutex

Solving a CSP:

1. Run the algorithm for $k=1$
2. Run the algorithm for $k = 2, 3, 4, \ldots$ until a solution is found or it is clear that no solution exists

Partial order planning: executing multiple (non exclusive) actions in the same time step

↳ (Move = mc {Del(= True, DelM = True})

↳ partial order plan, the order of Del(and DelM does not matter but must happen after mc

POP as a search problem

-> States are (mostly unfinished) plans

Each plan has 4 components:

- a set of actions
- a set of ordering constraints $act_0 \prec act_1$
- a set of causal links $act_0 \xrightarrow{p} act_1$
- a set of open preconditions

-> A plan is consistent if and only if there are no cycles in the ordering constraints and no conflicts with the causal links

-> A consistent plan with no open preconditions is a solution

-> A partial order plan is executed by repeatedly choosing any of the possible next actions

Solving POP

1. Assume propositional planning problem:
   Initial plan: = Start and Finish
   - the ordering constraint Start $\prec$ Finish
   - no causal links
   - all the preconditions in Finish are open

2. Successor function:
   - pick one open precondition p on an action act_1
   - generate a successor plan for every possible consistent way of chosing action act_0 that achieves p

3. Test the goal

> Generating a successor plan: - add causal link $act_0 \xrightarrow{p} act_1$
> - add the ordering constraint $act_0 \prec act_1$
> - resolve conflicts between new causal link and all existing actions
> - resolve conflicts between action act_0

VII Planning with costs as a search problem

- Nodes are planning states
- Edges correspond to operators

Optimal plan: plan with minimal cost

STRIPS

Problem $P = \langle F, I, G, O, c \rangle$

F: fluents (Boolean variables)
I: initial state $I \subseteq F$
G: goal state $G \subseteq F$
O: operators $\langle Pre(o), Add(o), Del(o) \rangle$ each $\subseteq F$
c: costs $c(o)$

States are subsets of fluents that have the value true

Heuristic function (h): estimate of the distance from a state to the goal

- $h^*(s)$ = the cost of the lowest-cost path from s to a goal node
- h is admissible if for all $s \in S$: $h(s) \leq h^*(s)$ (never overestimates)
- h is consistent if for all $s \in S$ and edge (s, s'): $h(s) \leq c(s, s') + h(s')$

STRIPS:
- uninformed heuristic

Relaxation:
- a less constrained version of the problem → heuristic

Classic STRIPS heuristic: uses the number of state variables that differ in the current state and a STRIPS goal.

→ Ignores almost all problem structure
→ Most successors have the same estimate

Relaxation: assumption about a problem that makes it easier to solve

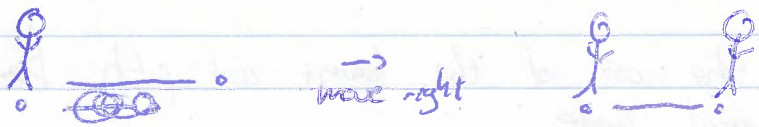
↓

ub. Manhatten distance: spots between two indices
↳ does not consider barriers

Blind heuristic: $h_{blind}(s) = 0$ for all s

Relaxation P^+: ignore bad effects of actions

→ Once a variable has a certain value, that value can always be used to satisfy preconditions



h^+ is admissible
→ any solution to P is a solution to P^+ as well

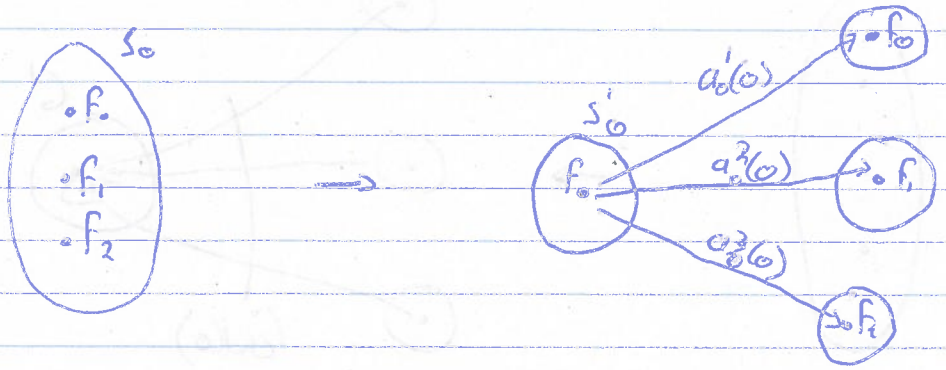
In strips, we need to remove the delete list:

$P = \langle F, I, G, O, c \rangle$
$P^+ = \langle F, I, G, O', c \rangle$ with $O' = \{Pre(o), Add(o), \emptyset\}$

Simplifying P^+

Ideally, $|s_0| = 1$ (number of fluents in a state equals 1)

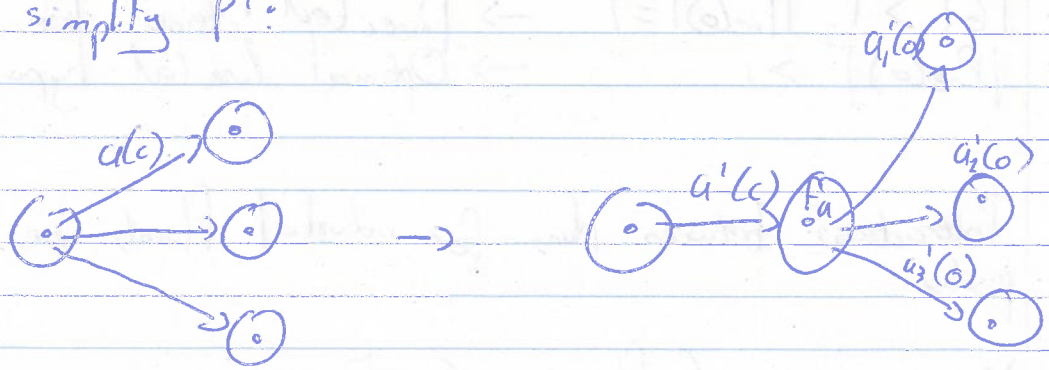
To simplify P^+:



A start state with size $|s_0| = n$ can be replaced with:
- a new start state containing a new fluent
  $s'_0 = \{F\}$
- a set of n zero-cost actions:
  $A_0 = \{\langle \{F\}, \{f_i\}, \emptyset \rangle \mid f_i \in s_0\}$

Ideally, $|Add(a)| = 1$

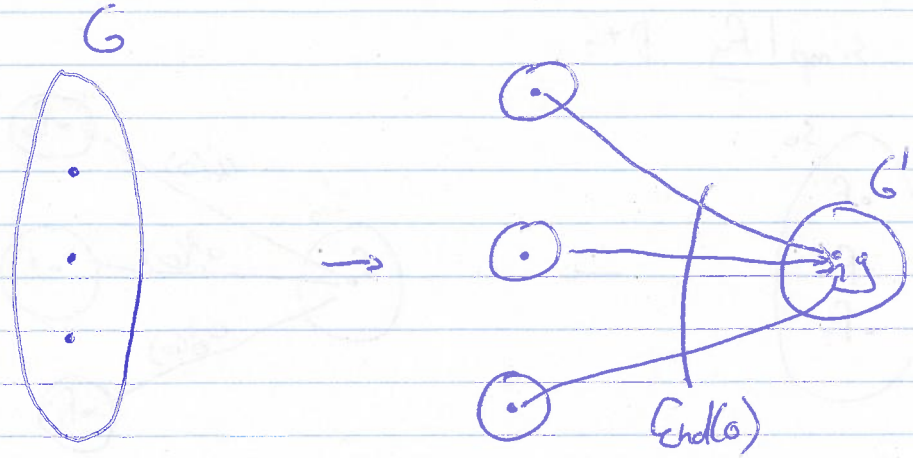
To simplify P^+:



An action a with $|Add(a)| = n$ can be replaced with:
- a new fluent f_a representing that the action has been executed
- an action $a' = \langle Pre(a), f_a, \emptyset \rangle$ with cost $c(a') = c(a)$
- a set of n zero-cost actions:
  $A' = \{\langle \{f_a\}, \{a_i\}, \emptyset \rangle \mid a_i \in Add(a)\}$

Ideally $|G| = 1$

To simplify P^+:



A goal with size $|G| = n$ can be replaced with:
- A new goal with a single newly-introduced fluent $G' = \{g\}$
- A single action $End = \langle G, g, \emptyset \rangle$

P^+ as a graph problem

Special cases of P^+ can be solved as graph problems:

- $|G| = |Pre(o)| = 1$ → Shortest path } tractable
- $|G| > 1$, $|Pre(o)| = 1$ → Directed Steine tree } NP-hard
- $|Pre(o)| > 1$ → Optimal directed hyperpath } NP-hard

Interactions between plans for individual fluents are always positive:

For any set of fluents $G \subseteq F$,

$$h^+(G) \leq \sum_{g \in G} h^+(g)$$

Difficulty of P^+: estimating cost of sets

Additive heuristic h^{add}: estimate the cost of a set as the sum of the costs of the individual fluents

Relies on the independence assumption for costs:

$$h^{add}(G) = \sum_{g \in G} h^{add}(g)$$

$$h^{add}(a;s) = cost(a) + h^{add}(Pre(a);s)$$

→ Cost to get to a from s is the cost to get to a plus the cost to get to all preconditions of a.

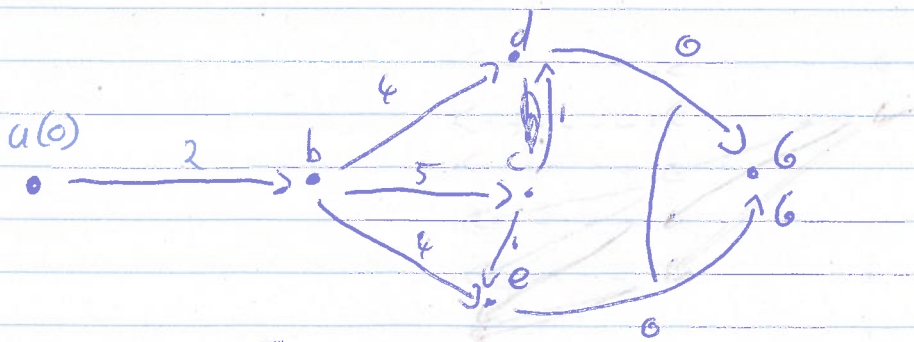
$$h^{add}(p;s) = \begin{cases} 0 & \text{if } p \in s \\ \min \text{ to get to } p \text{ from } s & \text{otherwise} \\ h^{add}(a_p(s)) & \end{cases}$$

$$a_p(s) = \min_{a \in O(p)} h^{add}(a;s)$$

$$h^{add}(P;s) = \sum_{p \in P} h^{add}(p;s)$$

$$h^{add}(s) = h^{add}(G;s)$$

We can use Dijkstra to compute h^{add}:



a	b	c	d	e	G
0	2	7	6	6	12

Problem with h_{add}:

- $h_{add}(G) = 12$

but a more optimal solution exists via c:

- $h^+(G) = 9$

↓

1. h_{add} counts edges twice (a → b)
2. Independence assumption:

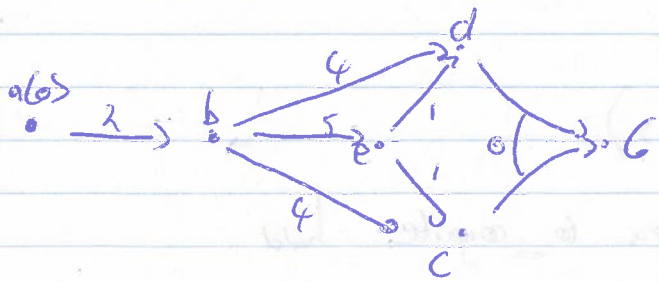
$$h_{add}(d, e) \sim h_{add}(d) + h_{add}(e)$$

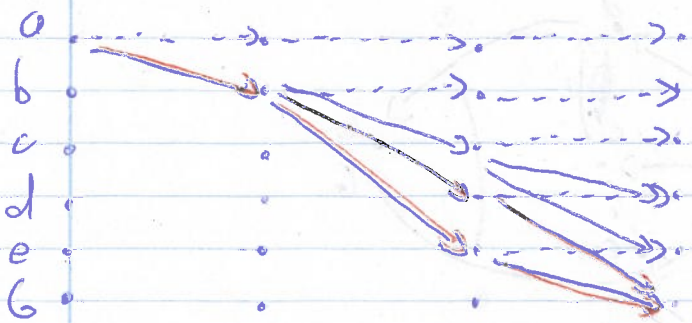
↳ not optimal

Heuristic h_{FF}

→ don't count edges twice

Relaxed planning graphs: graphically represent heuristic computation





Construct relaxed plan by starting from the goal, choose supports for each fluent at first layer in which it appears

$h_{FF}(s) = \{ a \to b, b \to d, b \to e, d, e \to G \} = 10$

h_{FF} is NOT admissible

h_{max} heuristic: replaces Σ in h_{add} with max:

$$h_{max}(p; s) = \max_{p \in P} (h_{max}(p; s))$$

Estimates cost of set as cost of most expensive fluent in set

→ Admissible

heuristic	description	admissible?
h_{add}	sum of cost precondition plus sum of cost	no
h_{max}	max cost of precondition plus sum of cost	Yes
h_{FF}	exploit positive interaction	no

VIII Bayesian networks

→ Also known as probabilistic / Belief networks

Probability:

$P(A=a)$ or $Pr(A=a)$ denotes: variable A takes value a

↓

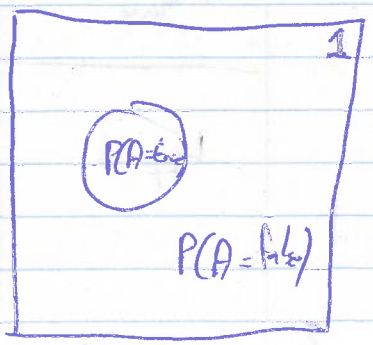
can also be non-binary

Meaning of probability:

- Frequentist: if A is repeated many times, the outcome a would occur $P(A=a)$ times the amount of repetitions (approximately)
- Bayesian / Judgmental / Subjective: The subjective degree of belief based on all available information, that one has in the event $A=a$

According to the information available to the subject

- Probabilities always sum up to one
- Prior and posteria probabilities are relative to an event



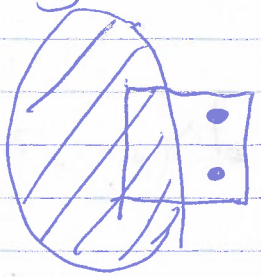
ex. X = "outcome of throwing a dice"

$\Omega(X) = \{1,\dots,6\}$

↓

values of X

Prior probability: $P(X=4) = 1/6$



Posterior probability: $P(X=4) = 1/3$

Event: showing you a partially occluded die

Joint probabilities: probabilities of multiple events happening at the same time

vb. $P(A = \text{true}, B = \text{true})$: proportion of outcomes where both A and B are true.

Conditional probabilities: probabilities of A conditioned on B

vb. $P(A = \text{true} \mid B = \text{true})$: out of all the outcomes where B is true, how many also have A equal to true

Joint probability distribution: distribution of probabilities over multiple variables

For each combination of variables, assign a probability:

vb.

A	B	P(A,B)
true	true	0.1
true	false	0.4
false	true	0.3
false	false	0.2

Product rule:

$$P(X_1, \dots, X_n) = \prod_{i=1}^{n} P(X_i \mid X_1, \dots, X_{i-1})$$

→ For k boolean values, we need a table of size 2^k and have to specify $2^k - 1$ numbers

↓

We can solve this using independance

Variables A and B are independant if any of the following holds:

- $P(A,B) = P(A) \cdot P(B)$
- $P(A \mid B) = P(A)$
- $P(B \mid A) = P(B)$

Then we say $A \perp\!\!\!\perp B$

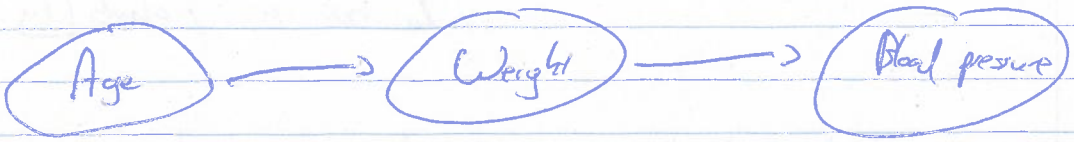
Conditional independance: two variables are conditionally independant if any of the following holds:

- $P(A,B \mid C) = P(A \mid C) \cdot P(B \mid C)$
- $P(A \mid B,C) = P(A \mid C)$
- $P(B \mid A,C) = P(B \mid C)$

Knowing C tells you everything about A's information about B is not relevant anymore.

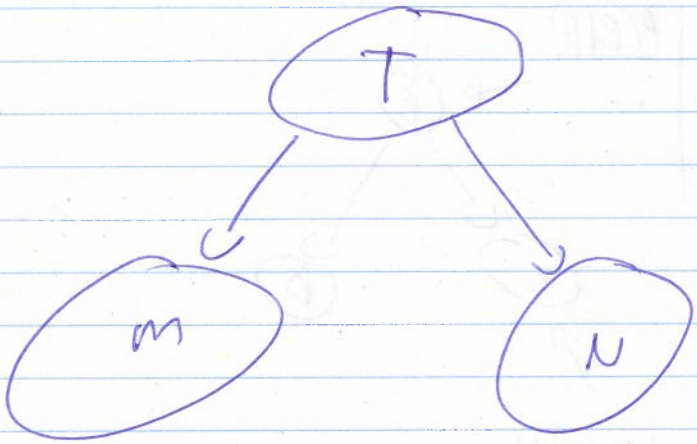
Then we say $A \perp\!\!\!\perp B \mid C$

Vb. Causal chain



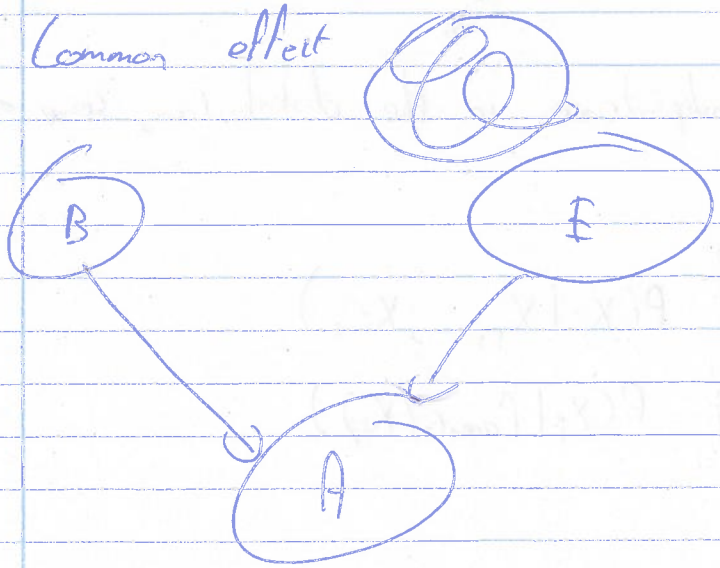
$A \not\perp\!\!\!\perp B$ but $A \perp\!\!\!\perp B \mid W$

Common cause



$M \not\perp\!\!\!\perp N$ but $M \perp\!\!\!\perp N \mid T$

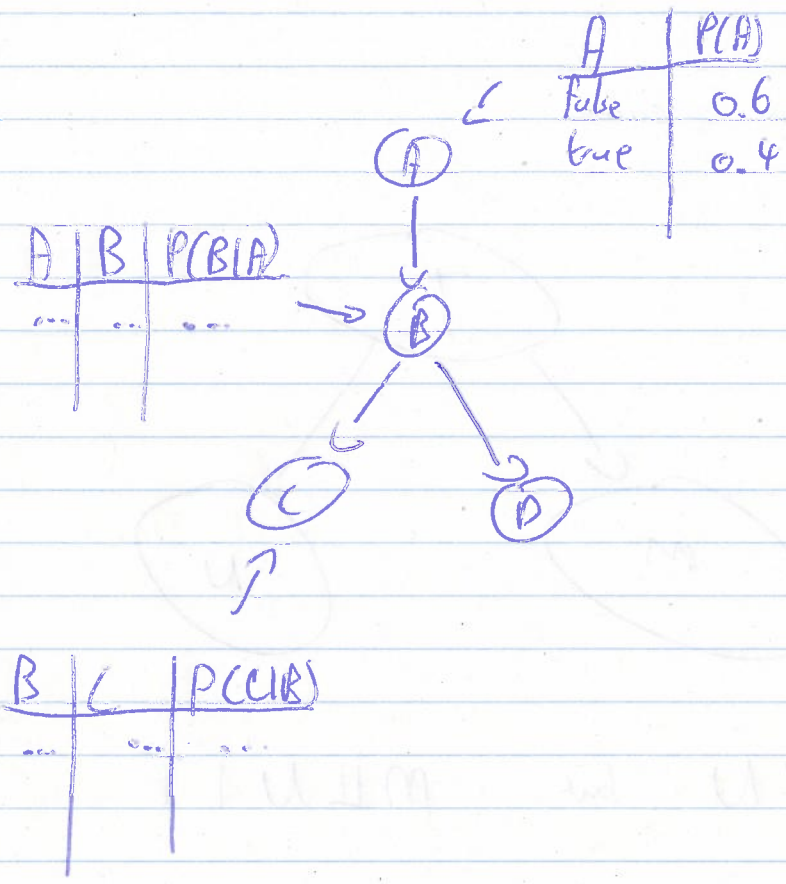
Common effect



$B \perp\!\!\!\perp E$ but $B \not\perp\!\!\!\perp E \mid A$

Bayesian network: a direct acyclic graph with nodes representing variables, each node has a probability table.

Each node has a conditional probability table based on the parent nodes:



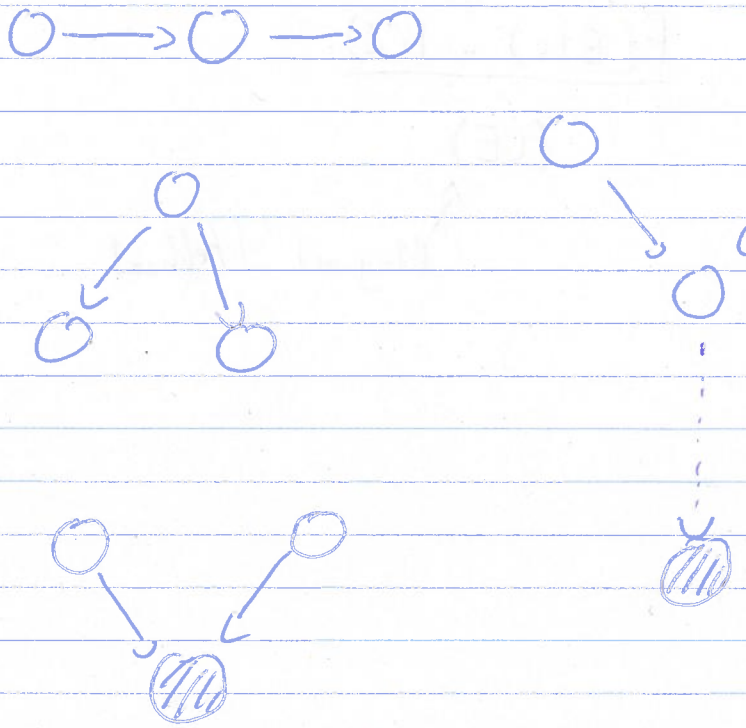
Because of the independences in the distribution, some arcs can be pruned:

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid X_1, \ldots, X_{i-1})$$

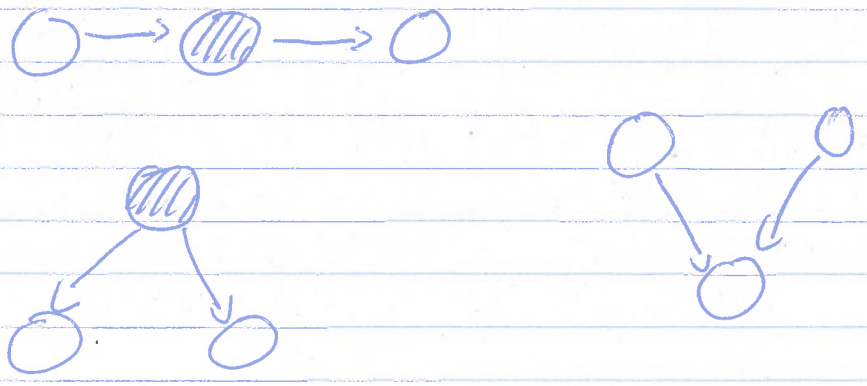
$$= \prod_{i=1}^{n} P(X_i \mid Parents(X_i))$$

D-separation (reachability): two nodes are D-separated if all chains connecting them are inactive

Active triplets:



Inactive triplets:



Bayes' rule

$$P(E|C) = P(E,C) \,/\, P(C)$$

Likelihood — Prior

$$P(C|E) = \frac{P(E|C) \cdot P(C)}{P(E)}$$

Posterior — Marginal likelihood

IX

Probability theory

- P is a probability measure over a set Ω
- P should obey three axioms:
  - $P(A) \geq 0$ for all events A
  - $P(\Omega) = 1$
  - $P(A \cup B) = P(A) + P(B)$ for disjoint events A and B

- Some consequences:
  - $P(A) = 1 - P(\Omega \setminus A)$
  - If $A \subseteq B$ then $P(A) \leq P(B)$
  - $P(A \cup B) = P(A) + P(B) - P(A \cap B) \leq P(A) + P(B)$

Probabilities over all disjoint subsets sum to 1:

$$\sum_{x \in X} P(X = x) = 1$$

Joint distribution: all possible combinations of events with their probability

	b_1	b_2	
a_1	0,05	0,54	0,59
a_2	0,08	0,04	0,12
a_3	0,17	0,12	0,29
	0,30	0,70	

$P(A)$

$P(B)$

Ub $P(a_1) = P(a_1, b_1) + P(a_1, b_2) = 0,05 + 0,54 = 0,59$

Conditional probability:

$$P(H|E) = \frac{P(H,E)}{P(E)}$$

Bayes rule:

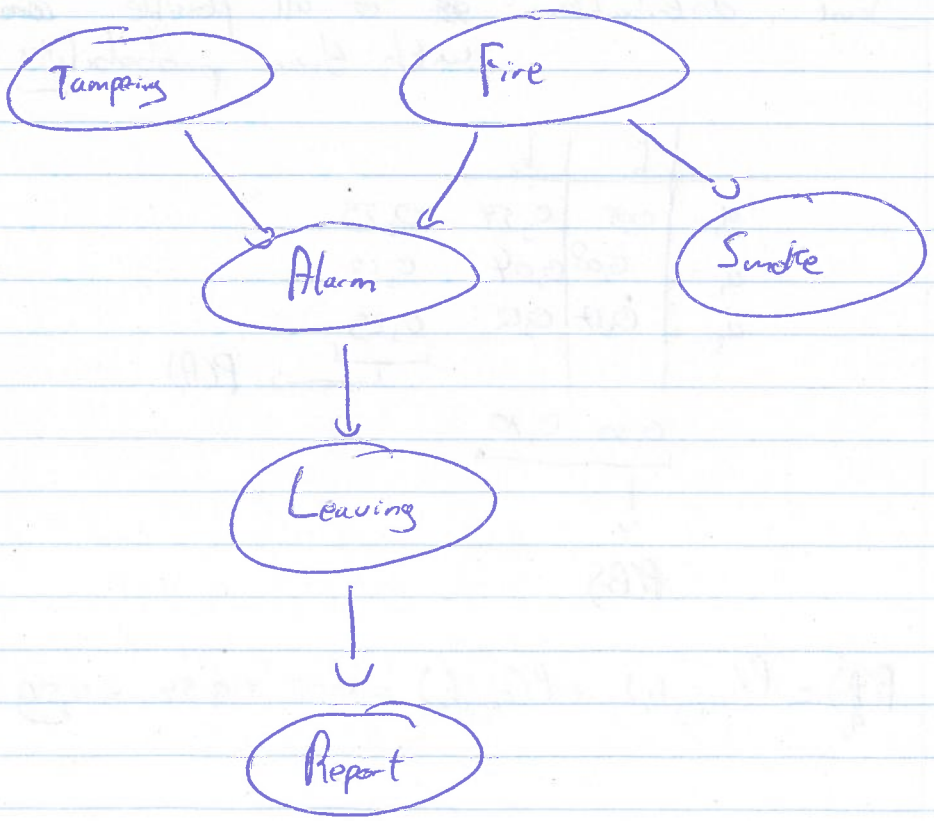
$$P(H|E) = \frac{P(H) \cdot P(E|H)}{P(E)}$$

Bayesian network: - directed acyclic graph
- edges from parents to children

The joint probability distribution in a Bayesian network:

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid Parents(X_i))$$

b. Bayesian network



$P(\text{tampering}) = 0,002$
$P(\text{fire}) = 0,01$
$P(\text{alarm} \mid \text{fire} \wedge \text{tampering}) = 0,5$
$P(\text{alarm} \mid \text{fire} \wedge \neg\text{tampering}) = 0,99$
$P(\text{alarm} \mid \neg\text{fire} \wedge \text{tampering}) = 0,85$
$P(\text{alarm} \mid \neg\text{fire} \wedge \neg\text{tampering}) = 0,0001$
$P(\text{smoke} \mid \text{fire}) = 0,9$
$P(\text{smoke} \mid \neg\text{fire}) = 0,01$
$P(\text{leaving} \mid \text{alarm}) = 0,88$
$P(\text{leaving} \mid \neg\text{alarm}) = 0,001$
$P(\text{report} \mid \text{leaving}) = 0,75$
$P(\text{report} \mid \neg\text{leaving}) = 0,01$

~~P(Alarm)~~ Computing $P(\text{Alarm})$:

→ We do not need leaving, report and smoke (between nodes)

$$P(\text{Alarm}) = \sum_{\text{tampering, fire}} P(\text{Alarm} \mid \text{Tampering, Fire}) P(\text{Tampering}) P(\text{Fire}) =$$

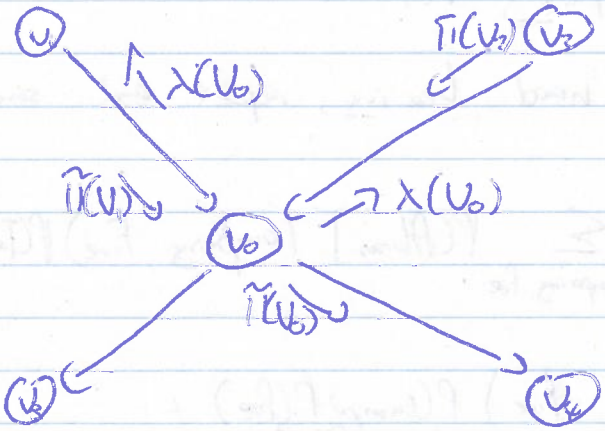
$P(\text{Alarm} \mid \text{tamp, fire}) P(\text{tamp}) P(\text{fire}) +$
$P(\text{Alarm} \mid \text{tamp}, \neg\text{fire}) P(\text{tamp}) P(\neg\text{fire}) +$
$P(\text{Alarm} \mid \neg\text{tamp, fire}) P(\neg\text{tamp}) P(\text{fire}) +$
$P(\text{Alarm} \mid \neg\text{tamp}, \neg\text{fire}) P(\neg\text{tamp}) P(\neg\text{fire}) = 0,00266$

And so: $P(\neg\text{alarm}) = 0,99734$

These computations can be done quicker:

- Look at nodes as objects that can pass around information to connected nodes. Keep computations local.
- Look at computations and find a smart way of reordering them, thus minimizing the amount of summations and multiplications

Message passing: focused on signals being sent between nodes



- $\tilde{\pi}_i(V)$ are messages send to a child node
- $\lambda(V)$ are messages send to a parent node

Every node knows its own local probability table

vb. Compute $P(V_0)$

$$P(V_0) = p(V_0 | V_1, V_2) \cdot P(V_1) \cdot P(V_2)$$

V_0 knows $P(V_0 | V_1, V_2)$ but does not know $P(V_1)$ and $P(V_2)$

V_1 sends $\tilde{\pi}(V_1) = P(V_1)$ to V_0
V_2 sends $\tilde{\pi}(V_2) = P(V_2)$ to V_0

Message passing only works when a network is a tree, no cycles.

If a tree is cyclic:
- Loop-cut set conditioning → condition on non-loop variables
- Clustering of variables

Variable elimination: solve a query by summing out variables in a logical order

→ Works on arbitrary networks (not only trees)

Factor: a factor $f(X_1, \ldots, X_k)$ is a function from variables to a real value:

$$f(X_1, \ldots, X_k) = r \in \mathbb{R}$$

→ A conditional / joint / marginal probability can distribution can be represented by a factor.

↓

However, a factor does not need to represent a particular distribution.

Vb. $f_1(\text{Alarm}, \text{Tampering}) \stackrel{\text{def}}{=} P(\text{Alarm} \mid \text{Tampering})$

Vb. $f_2(\text{Alarm}) \stackrel{\text{def}}{=} P(\text{Alarm} \mid \neg\text{Tamp})$

Computations over factors:

$f_1(A,B) \times f_2(B,C) = f_3(A,B,C)$ where
$f_3(a,b,c) = f_1(a,b) \cdot f_2(b,c)$

ex. $F_1(a_1,b_1) = 0,6$
$f_2(b_1,c_2) = 0,7$

$f_3(a_1,b_1,c_2) = 0,35$

Factor marginalization: summing out a factor

$$\sum_B f_3(A,B,C) = f_4(A,C)$$

f_3

a_1	b_1	c_1	0,25
a_1	b_1	c_2	0,35
a_1	b_2	c_1	0,08
a_1	b_2	c_2	0,16
a_2	b_1	c_1	0,05
a_2	b_2	c_2	0,07
...	...	...	...

f_4

a_1	c_1	0,33
a_1	c_2	0,51
a_2	c_1	0,05
...	...	

Factor reduction: reducing the amount of variables

$f_3(A,B,c_1) = f_5(A,B)$

f_3

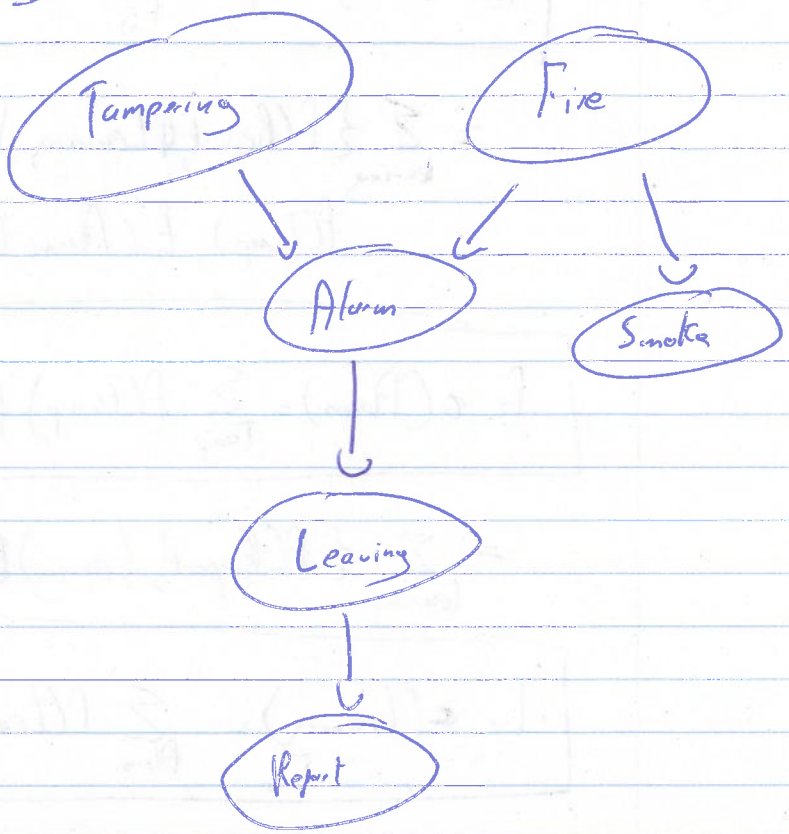
a_1	b_1	c_1	0,37
a_1	b_1	c_2	0,35
a_1	b_2	c_1	0,08
a_1	b_2	c_2	0,16
...	...	...	...

f_5

a_1	b_1	0,25
a_1	b_2	0,08
...	...	

Variable elimination algorithm

Ex. Compute P(Report)



Elimination ordering:

Smoke - Fire - Tampering - Alarm - Leaving

$$P(\text{Report}) = \sum_{\text{tamp, fire, alarm, smoke, leaving}} P(\text{Tampering}, \text{Fire}, \text{Alarm}, \text{Smoke}, \text{Leaving}, \text{Report})$$

$$= \sum_{\text{tamp, fire, alarm, smoke, leaving}} \{ P(\text{Tampering}) P(\text{Fire}) P(\text{Alarm} \mid \text{Tampering}, \text{Fire}) P(\text{Smoke} \mid \text{Fire}) P(\text{Leaving} \mid \text{Alarm}) P(\text{Report} \mid \text{Leaving}) \}$$

Set summations before parts that need them (marginalizing)

$$= \sum_{\text{leaving}} \{ P(\text{Report} \mid \text{Leaving}) \cdot \sum_{\text{Alarm}} \{ P(\text{Leaving} \mid \text{Alarm}) \cdot \sum_{\text{Tamp}} \{ P(\text{Tamp}) \cdot \sum_{\text{Fire}} \{ P(\text{Fire}) P(\text{Alarm} \mid \text{Fire}, \text{Tamp}) \sum_{\text{Smoke}} \{ P(\text{Smoke} \mid \text{Fire}) \} \} \} \} \}$$

apply f_1

$$f_1(\text{Fire}) = \sum_{\text{Smoke}} P(\text{Smoke} \mid \text{Fire}) = \begin{pmatrix} 1 \\ 1 \end{pmatrix} = P(\text{Smoke} \mid \neg\text{Fire}) + P(\text{Smoke} \mid \text{Fire}) = P(\neg\text{Smoke} \mid \neg\text{Fire}) + P(\neg\text{Smoke} \mid \text{Fire})$$

$$= \sum_{\text{leaving}} \{ P(\text{Report} \mid \text{Leaving}) \sum_{\text{alarm}} \{ P(\text{Leaving} \mid \text{Alarm}) \sum_{\text{Tamp}} \{ P(\text{Tamp}) \sum_{\text{Fire}} f(\text{Fire}) f(\text{Alarm}, \text{Tamp}, \text{Fire}) f_1(\text{Fire}) \} \} \}$$

$$f_2(Alarm, Tamp) = \sum_{Fire} f(Fire)\, f(Alarm, Tamp, Fire)\, f_1(Fire)$$

$$= \sum_{Leaving} \{ f(Report, Leaving) \sum_{Alarm} \{ f(Leaving, Alarm) \sum_{Tamp} f(Tamp)\, f_2(Alarm, Tamp) \} \}$$

$$f_3(Alarm) = \sum_{Tamp} f(Tamp)\, f_2(Alarm, Tamp)$$

$$= \sum_{Leaving} \{ f(Report, Leaving) \sum_{Alarm} f(Leaving, Alarm)\, f_3(Alarm) \}$$

$$f_4(Leaving) = \sum_{Alarm} f(Leaving, Alarm)\, f_3(Alarm)$$

$$= \sum_{Leaving} \{ f(Report, Leaving)\, f_4(Leaving) \}$$

$$f_5(Report) = \sum_{Leaving} f(Report, Leaving)\, f_4(Leaving)$$

$$= f_5(Report)$$

→ Normalize if f_5 is not a probability distribution

1. Write down query variables
2. Write down observed variables
3. Write down:
   - the product ⊗ to compute the query
   - the reduced formula based on the network structure
4. Identify factors and reduce observed variables
5. Fix an elimination ordering
6. For every variable Z in this ordering
   - Multiply factors containing Z
   - Sum out Z to obtain new factor f_Z
   - Remove the multiplied factors from the list and add f_Z

④ Normalize the result

Finding good elimination orderings

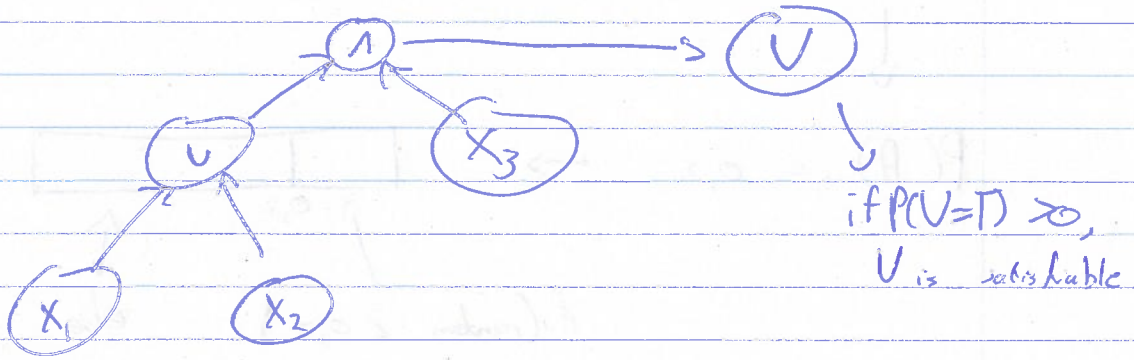
- NP-Hard (optimal)
- Bayesian inference, even with an optimal ordering, is NP-Hard)

↓

Reduction from SAT:

For any formula ℓ we can construct a Bayesian network B_ℓ with designated variable V such that $P(V = \text{true}) > 0$ if ℓ is satisfiable.

ex. $(X_1 \vee X_2) \wedge X_3$:



↓

if $P(V=T) > 0$, V is satisfiable

X Stochastic simulation: idea: get probabilities from samples

If we could sample from a variable's posteria probability distribution, we could estimate its posteria probability distribution

Generating samples: a bayesian network is a-cyclic and directed

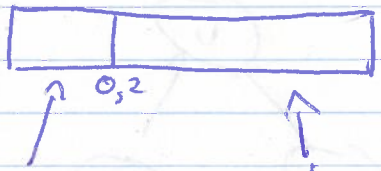
↓

variables can be ordered such that node X is indexed after its parents

We have to generate a random number for each variable in the Bayesian network.

↓

P(A) = 0,2 →

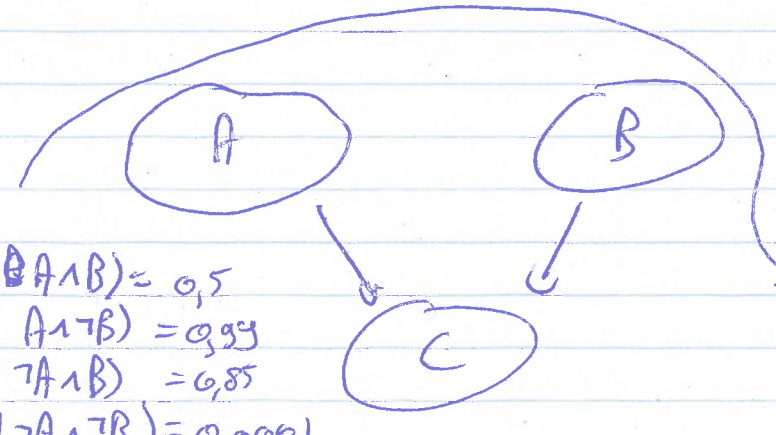


if (random < 0,2) else

↓

P(A) = True A = False

with dependance:



P(C | A ∧ B) = 0,5
P(C | A ∧ ¬B) = 0,99
P(C | ¬A ∧ B) = 0,85
P(C | ¬A ∧ ¬B) = 0,0001

1. First calculate A & B
2. Pick the probability from the options
3. Calculate C

Now we can repeat this a lot of times

↓

Forward sampling

How many samples do we need?

$$P(|p_{approx} - p_{real}| > \varepsilon) \leq 2 \cdot e^{-2 \cdot \#samples \cdot \varepsilon^2}$$

How many samples to ensure that in less than δ fraction of all cases, the difference between estimated and actual distribution $> \varepsilon$?

↓

Solve $2 \cdot e^{-2 \cdot \#samples \cdot \varepsilon^2} < \delta$ for #samples

↓

$$\#samples > -\ln(\delta/2)/2 \cdot \varepsilon^2$$

vb. if $\varepsilon = 0,1$, $\delta = 0,05$ $\longrightarrow$ #samples > 184

Rejection sampling: reject samples that can't be used:

vb. Calculate $P(temp \mid \neg smoke, report)$

Every sample that does not have ¬smoke or report will be rejected

Drawback: a lot of samples will be rejected, which is very inefficient

Importance sampling : don't reject samples but weight samples according to how likely they are included

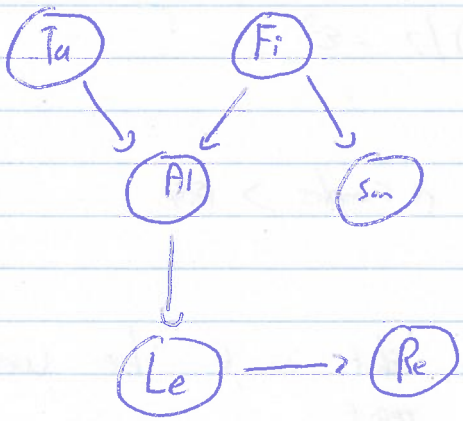
↓

Probability of a proposition is weighted average of samples from a distribution

$$P(\alpha \mid evidence) \approx \frac{\sum_{sample \models \alpha} weight(sample)}{\sum_{sample} weight(sample)}$$

Likelihood weighting : special case of IS, sample only non-evidence variables and weight each sample according to $P(evidence \mid sample)$

eg. We want to know $P(Ta \mid sm, re)$



$P(sm \mid fi) = 0,9$
$P(sm \mid \neg fi) = 0,01$
$P(re \mid le) = 0,75$
$P(re \mid \neg le) = 0,01$

	Ta	Fi	Al	Le	Weight
s_1	true	false	true	false	$0,01 \times 0,01$
s_2	false	true	false	false	$0,9 \times 0,01$
s_3	false	true	true	true	$0,9 \times 0,75$
s_4	true	true	true	true	$0,9 \times 0,75$

Drawback : many samples are needed

Approximate inference is in general NP-hard

Markov chain: special case of Bayesian network with chain structure

Thus $P(S_{t+1} | S_0, \dots, S_t) = P(S_{t+1} | S_t)$

Often S_t represents the state at time t

If we know S_t, we do not need to know $S_0 \dots S_{t-1}$ in order to compute S_{t+1}.

↓

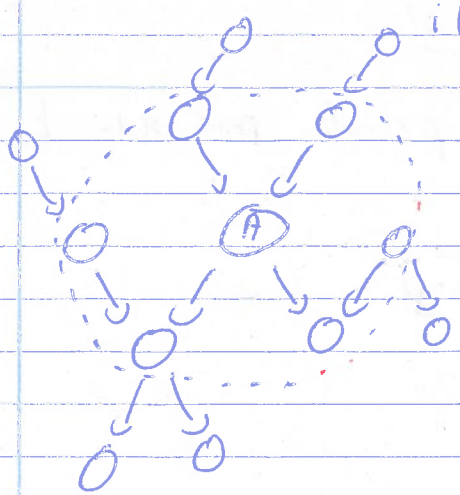
Conditionally independent

→ the future is independent of the past given the present

↓

Markov property

Markov blanket: a set of nodes around a node caring it from outside influences



- A is shielded from influences from outside by a joint value assignment to its parents, its children, and its co-parents

↓

when these are all observed, all nodes to A from nodes outside the blanket are d-seperated

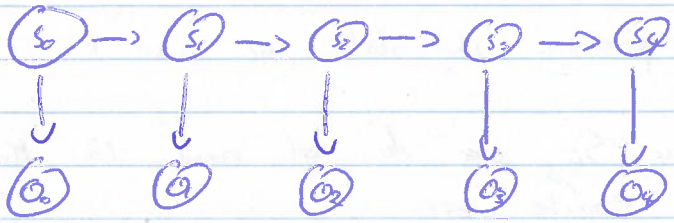
Stationary Markov chain: a markov chain is stationary if:

For all $t > 0$, $t' > 0$, $P(S_{t+1} | S_t) = P(S_{t'+1} | S_{t'})$

↓

We specify only: - $P(S_0)$ → initial conditions
- $P(S_{t+1} | S_t)$ → transition dynamics

Hidden Markov Model: special case of Bayesian network with the following structure



- $P(S_0)$ → initial conditions
- $P(S_{t+1} | S_t)$ → transition dynamics
- $P(O_t | S_t)$ → sensor model

Queries to a HMM:

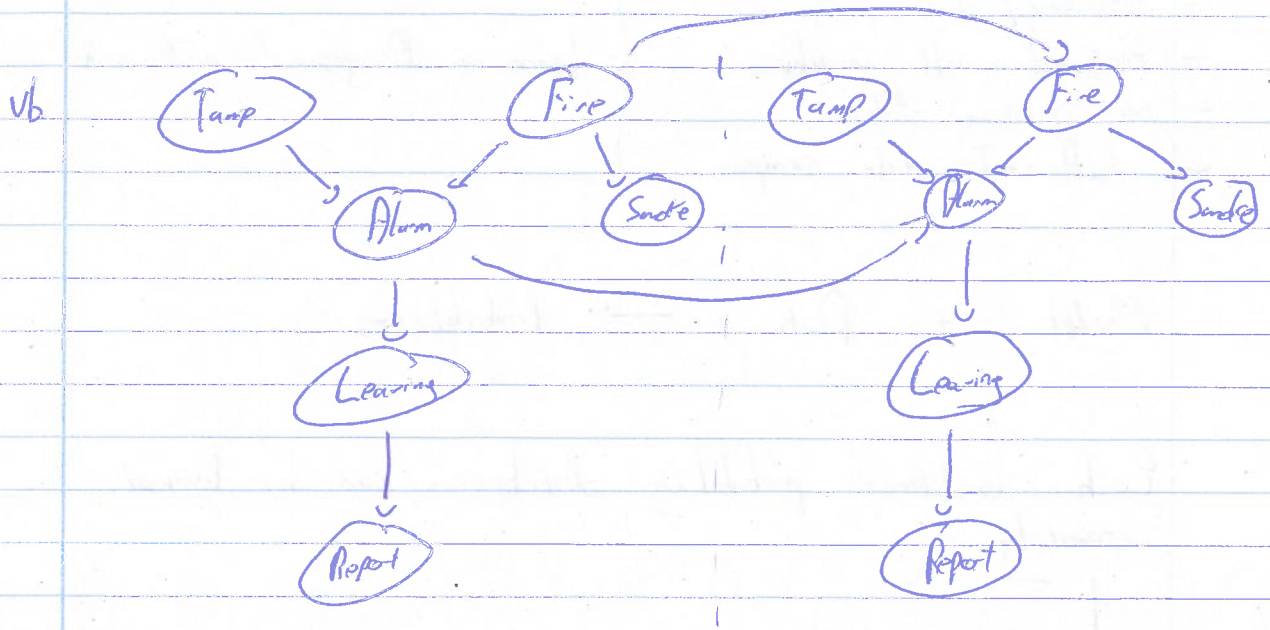
- What is the distribution of the current "true" state based on the observation history?

$P(S_t | O_0, \ldots, O_t)$ (Filtering)

- What was the distribution of a particular past state based on the observation history

$P(S_k | O_0, \ldots O_k, \ldots O_t)$ (Smoothing)

Dynamic Bayesian Network: a HMM but does not need to satisfy Markov property



$P(t_0)$ $P(t_{n+1} | t_n)$

Constructing Bayesian Networks: bayesian networks can be constructed or learned

Probability generation:

- Statistical domain knowledge
- Learning from data
- Knowledge from domain experts

↓

Conditional probabilities are very hard to obtain

Learning Bayesian networks:

- structure / domain
- observed all variables
- no missing data
- sufficient data samples

} learn a Bayesian network

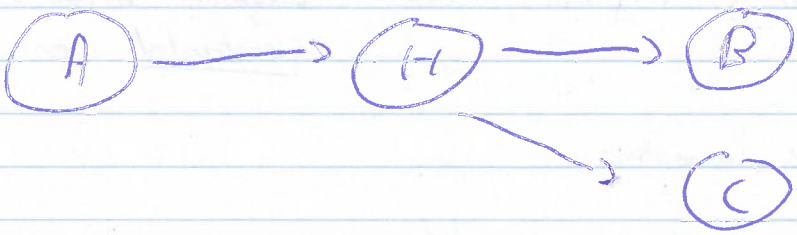
Model + Data ⟶ Probabilities

Each conditional probability distribution can be learned separately

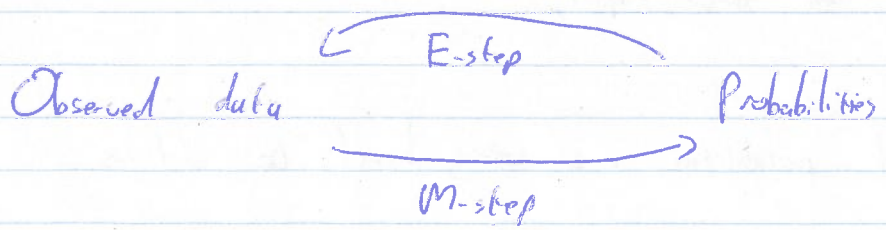
↓

When there are many parents for a node, there can be little or no data for each probability estimate

Unobserved variables



What if we had only observed A, B & C?



Expectation - Maximization algorithm: used for computing probabilities of unobserved variables

1. Start with made-up probabilities (in this case to P(H|A))

2. E-step : generate data points for unobserved variables, give the expected number of data points based on the given probability distribution

3. M-step : infe the probabilities from the generated data

4. Reput.

Structure learning: learning the structure from data

→ Independance tests

XI Utility theory

Actions result in outcomes

if O_1 and O_2 are outcomes:

- $O_1 \preceq O_2$ means O_1 is at least as desirable as O_2
- $O_1 \sim O_2$ means $O_1 \preceq O_2$ and $O_2 \preceq O_1$
- $O_1 \succ O_2$ means $O_1 \preceq O_2$ and $O_2 \preceq O_1$

An agent may not know the outcomes of its actions, but only have a probability distribution of the outcomes;

Lottery: probability distribution over outcomes:

$$[p_1 : O_1, p_2 : O_2, \dots, p_n : O_n]$$

with $p_1 \dots p_n$ being probabilities

$\rightarrow \sum_i p_i = 1$

and $O_1 \dots O_n$ being outcomes

Completeness: Agents have to act, so they must have preferences: $\forall O_1 \forall O_2 \; O_1 \preceq O_2$ or $O_2 \preceq O_1$

For any given pair of courses, passing one is at least as desirable as passing the other

Transitivity: preferences must be transitive

if $O_1 \preceq O_2$ and $O_2 \preceq O_3$ then $O_1 \preceq O_3$

Monotonicity: an agent prefers a large chance of getting a better outcome than a smaller chance

If $o_1 \succ o_2$ and $p > q$ then $[p: o_1, 1-p: o_2] \succ [q: o_1, 1-q: o_2]$

Continuity: suppose $o_1 \succ o_2$ and $o_2 \succ o_3$, then there exists a $p \in [0,1]$ such that:

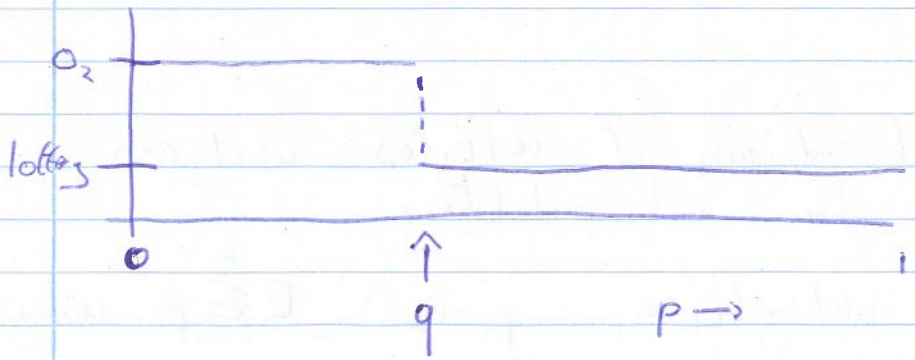
$$o_2 \sim [p: o_1, 1-p: o_3]$$

Vb. Suppose $o_1 \succ o_2$ and $o_2 \succ o_3$, consider the case where the agent would prefer:

- o_2
- the lottery $[p: o_1, 1-p: o_3]$

for different values of $p \in [0,1]$

Then:



Decomposability: an agent is indifferent between lotteries that have same probabilities and outcomes

$$[p: O_1, 1-p: [q: O_2, 1-q: O_3]] \sim [p: O_1, (1-p)q: O_2, (1-p)(1-q): O_3]$$

Substitutability: if $O_1 \sim O_2$ then the agent is indifferent between lotteries that only differ by O_1 and O_2

$$[p: O_1, 1-p: O_3] \sim [p: O_2, 1-p: O_3]$$

We need a measure of preference that can be combined with probabilities.

Preferences can be measured by a function:

$$\text{utility}: \text{outcomes} \to [0,1]$$

such that:

- $O_1 \succ O_2$ if and only if $\text{utility}(O_1) \geq \text{utility}(O_2)$
- utilities are linear with probabilities:

$$\text{utility}([p_1: O_1, \ldots, p_k: O_k]) = \sum_{i=1}^{k} p_i \cdot \text{utility}(O_i)$$

Allais paradox

What would you prefer?

A: $1m
B: lottery: [0,10 : $2,5m, 0,89 : $1m, 0,01 : $0]

What would you prefer?

C: lottery: [0,11 : $1m, 0,89 : $0]
D: lottery: [0,10 : $2,5m, 0,9 : $0]

It is inconsistant with the axioms of preferences to have:

$A \succ B, D \succ C$

A, C : lottery [0,11 : $1m, 0,89 : X]
B, D : lottery [0,10 : $2,5m, 0,01 : $0, 0,89 : X]

Framing effect: by putting the question differently, people value certain choises more or less

What an agent should do depends on:

- The agents ability : possible options
- The agents belief : the way the world could be
- The agents preference

Decision variables: ~~var~~ variables an agent gets to choose a value for

~~Decision Trees~~

Expected value: the average value, weighting possible worlds by their probability

$f(\omega)$: value of f on world ω

- $E(f) = \sum_{\omega \in \Omega} P(\omega) \cdot f(\omega)$

$E(f \mid e) = \sum_{\omega \models e} P(\omega \mid e) \cdot f(\omega)$

Single decisions: agent makes all decisions before acting

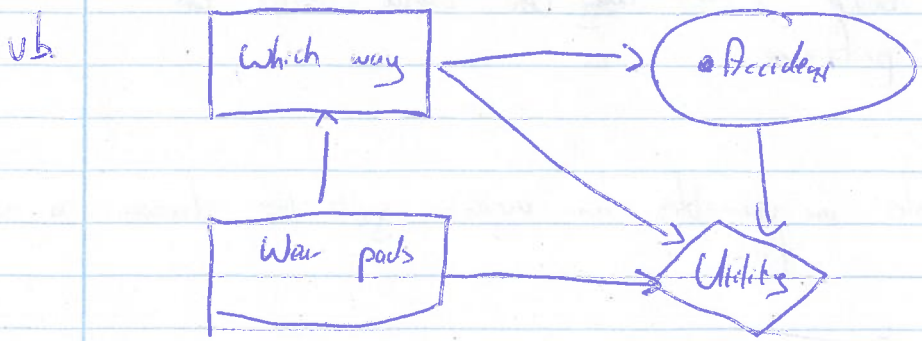
↓

The agent can choose a value for each decision variable

Expected utility of decision $D = d_i$: $E(U \mid D = d_i)$

Decision networks:

- Belief network
- □ – Decision nodes: the agent chooses a value for these
- ◇ – Utility nodes: parents are the variables on which the utility depends



A single-stage decision network consists of:

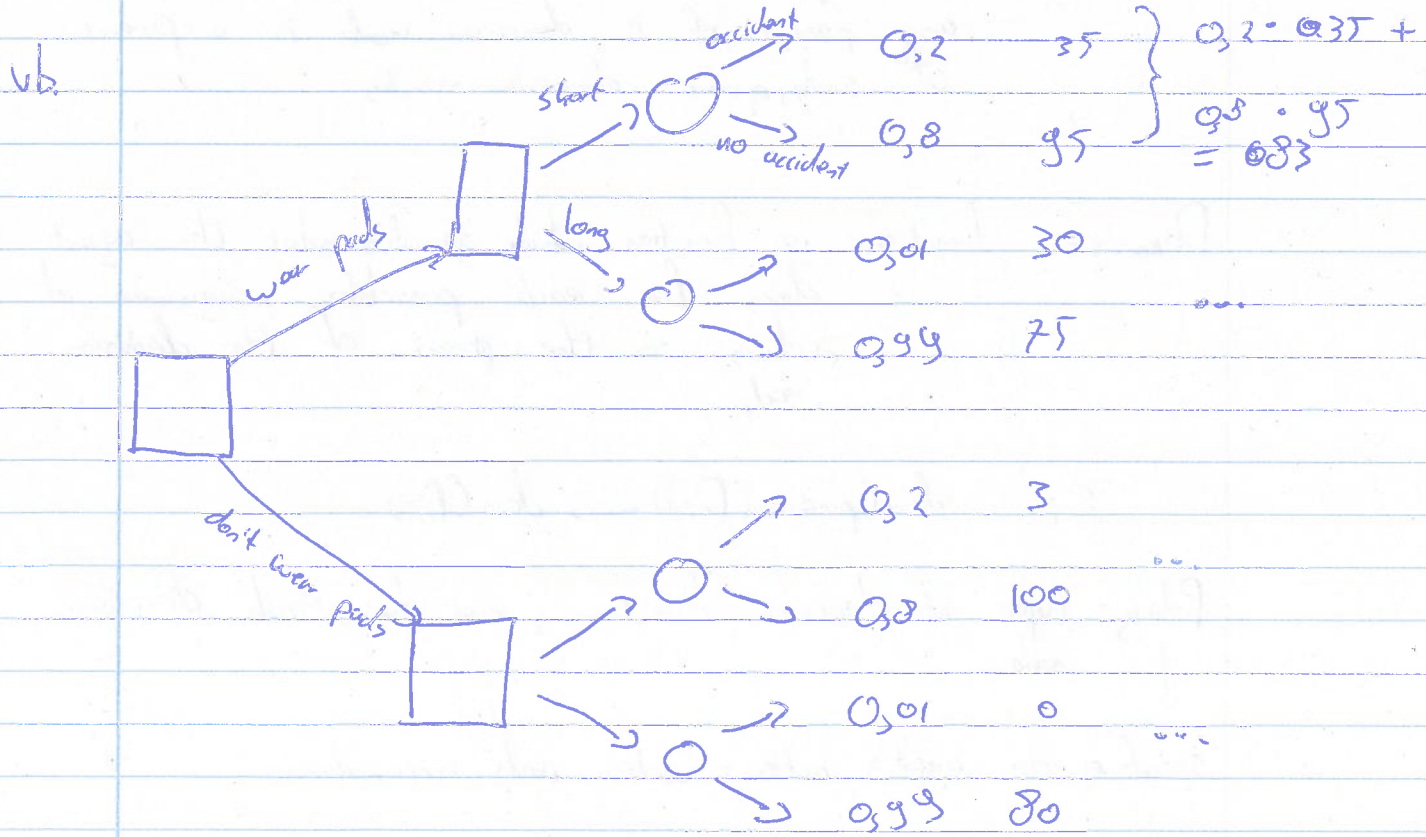
- Directed acyclic graph with decision, random and utility nodes
- A domain for each decision variable and each random variable
- A unique utility node

Factors:

- Utility function: factor on the parents of the utility node
- Conditional probability for each random variable given its parents

Finding an optimal decision:

- Create a factor for each conditional probability and for the utility
- Sum out all of the random variables
- This creates a factor on D that gives the expected utility for each value in the domain of D
- Choose D with the maximum value in the factor



Sequential decisions: actions can depend on what is observed

Sequential decision problem: sequence of decision variables $D_1, \ldots, D_n$

↓

Each D_i has an information set of variables $parents(D_i)$, whose value will be known at the time decision D_i is made

No-forgetting: a no-forgetting decision network is a network where:

- the decisions are totally ordered. This is the order in which the actions will be taken
- all decision nodes that come before D_i are parents of decision node D_i
- any parent of a decision node is a parent of subsequent decision nodes

Decision function: a function which specifies what the agent does for each possible assignment of values to the parent of the decision node.

$\delta_i : dom(parents(D_i)) \longrightarrow dom(D_i)$

Policy: tuple of decision functions, one for each decision node

↓

specifies an agent's actions under each circumstance

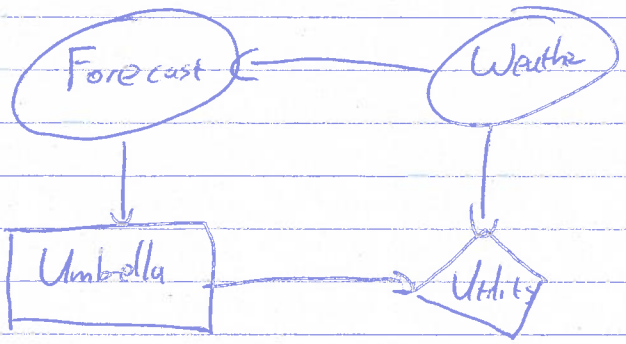
Optimal policy : policy with the highest utility

$$\mathcal{E}(u|\delta) = \mathcal{E}_{\omega \models \delta}\, u(\omega) \cdot P(\omega)$$

Finding an optimal policy

1. Create a factor for each conditional pobability table and a factor for the utility
2. Sum out random variables that are not parents of a decision node
3. Let D be the last variable - D is only in a factor f with its parents
4. Eliminate D by maximizing:
   - an optimal decision function for D
   - a new factor $\max_D f$
5. Repeat 2 until there are no more decision nodes
6. Sum out the remaining random variables. Multiply the factors: this is the expected utility of an optimal policy

vb.



Weather	Umbrella	Value
no rain	take	20
no rain	leave	100
rain	take	70
no rain	leave	0

Weather	forecast	value
no rain	sunny	0,7
no rain	cloudy	0,2
no rain	rainy	0,1
rain	sunny	0,15
rain	cloudy	0,25
rain	rainy	0,6

Weather	value
no rain	0,7
rain	0,3

forecast	umbrella	value
sunny	take	$0,7 \cdot 0,7 \cdot 20 + 0,3 \cdot 0,15 \cdot 0,7 = 12,95$
sunny	leave	$0,7 \cdot 0,7 \cdot 100 + 0,3 \cdot 0,15 \cdot 0 = 49$
cloudy	take	$0,7 \cdot 0,2 \cdot 20 + 0,3 \cdot 0,75 \cdot 70 = 8,05$
cloudy	leave	$0,7 \cdot 0,2 \cdot 100 + 0,3 \cdot 0,25 \cdot 0 = 14$
raining	take	$0,7 \cdot 0,1 \cdot 20 + 0,3 \cdot 0,6 \cdot 70 = 14$
raining	leave	$0,7 \cdot 0,1 \cdot 100 + 0,3 \cdot 0,6 \cdot 0 = 7$

Now maximazing D:

forecast	value
sunny	49
cloudy	14
raining	14

$\max_{\text{umbrella}} f$

forecast	umbrella
sunny	leave
cloudy	leave
raining	take

$\arg\max_{\text{umbrella}} f$

XII Decision processes

Utility and time: how to trade of pleasures today with pleasures in the future?

Ongoing processes: checking utility at the end might not be useful
↓
agent might not yet to the end

→ A sequence of rewards might be useful

Rewards:
- action costs
- prizes
- penalties

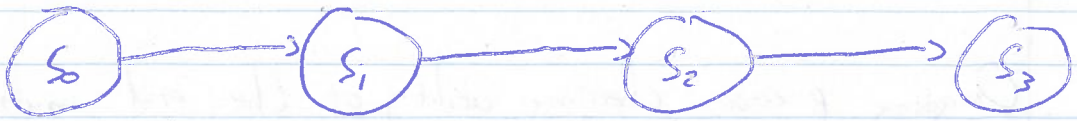
Finite horizon: fixed numbe of steps
Infinite horizon: neve ending numbe of steps
Indefinite horizon: agent will eventually stop but does not know when

World state: information such that if the agent knew the world state no information about the past is relevant to the future
↳ Markovian assumption

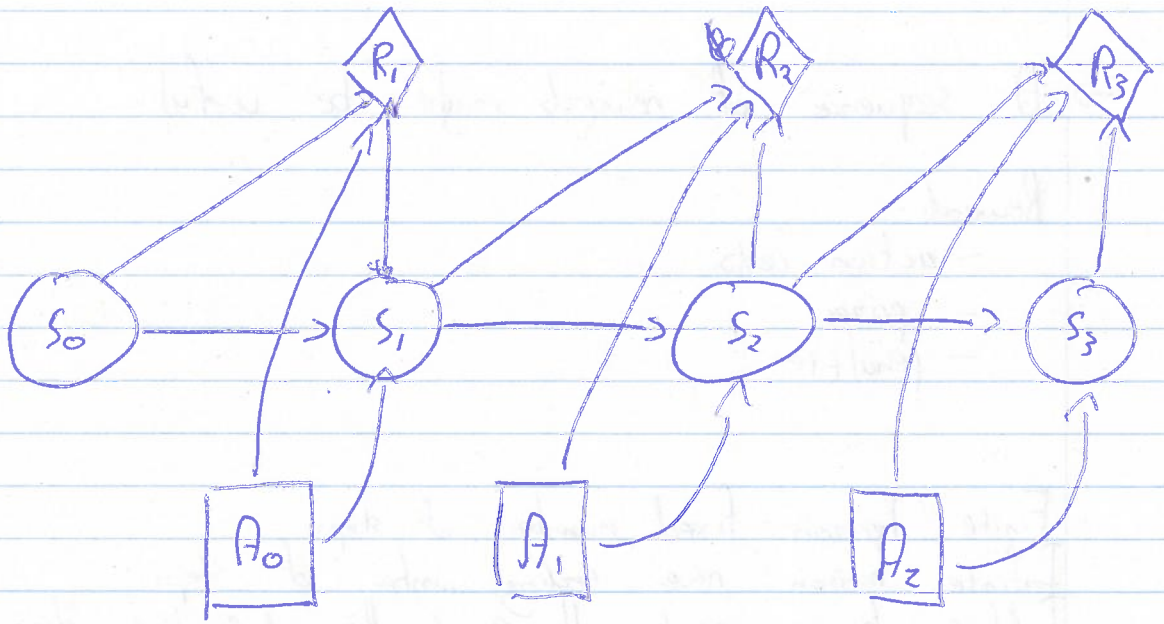
$$P(S_{t+1} \mid S_0, A_0, \ldots, S_t, A_t) = P(S_{t+1} \mid S_t, A_t)$$

$P(s' \mid s, a)$: probability that the agent will be in state s' after performing action a in state s

Markov chain: belief network in which with random variables where each variable only directly depends on its predecessor in the sequence



Markov Decision Process: augments a Markov chain with actions and values



At each stage, the agent decides what to do, the reward and resulting state depend on both the previous state and the action performed

MDP:
- set of States S
- set of Actions A
- $P(S_{t+1} \mid S_t, A_t)$ specifies the dynamics
- $R(S_t, A_t, S_{t+1})$ specifies the reward at time t
- $R(s, a, s')$ is the expected reward received
- $R(s, a) = \sum_{s'} P(s' \mid s, a) R(s, a, s')$
- γ is the discount factor

Planning horizon: how far ahead the planner looks to make a decision

Fully observable MDP: the agent gets to observe S_t when deciding on action A_t

Partially observable MDP: the agent has some noisy sensor of the state.

To decide what to do, agent compares different sequences of rewards

↓

Agent receives sequence of rewards $r_1, r_2, r_3, \ldots, r_n$

↓

combining rewards

Total reward: $V = \sum_{i=1}^{\infty} r_i$

Average reward: $V = \lim_{n \to \infty} (r_1 + \ldots + r_n)/n$

Discounted reward: $V = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \ldots$

↓

where γ is the discount factor

$0 \leq \gamma \leq 1$

Policy: specification of what the agent should do as a function of the state it is in.

Stationary policy: $\pi : S \to A$

An agent following π is state s will execute $\pi(s)$ next

Optimal policy: a policy with the maximum expected discounted reward

↓

For a Fully observable MDP with stationary dynamics and rewards with infinite or indefinite horizon, there is always an optimal stationary policy

Vb. Policy one:

States: {healthy, sick}
Actions: {party, relax}

Available policies: $|\text{Actions}|^{|\text{states}|} = 2^2 = 4$

1. π_1[healthy] = party, π_1[sick] = party
2. π_2[healthy] = relax, π_2[sick] = relax
3. π_3[healthy] = party, π_3[sick] = relax
4. π_4[healthy] = relax, π_4[sick] = party

- $Q^{\pi}(s,a)$ is the expected value of doing action a starting in state s, then following policy π
- $V^{\pi}(s)$ is the expected value of following policy π in state s

Bellman equations:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \cdot \sum_{s'} P(s'|s,a) \cdot V^{\pi}(s')$$

where $$R(s,a) = \sum_{s'} P(s'|s,a) \cdot R(s,a,s')$$

and $$V^{\pi}(s) = Q^{\pi}(s, \pi(s))$$

- $Q^*(s,a)$: expected value of doing a in state s then following the optimal policy
- $V^*(s)$: expected value of following the optimal policy in state s
- Bellman optimality equations:

$$Q^*(s,a) = R(s,a) + \sum_{s'} P(s'|s,a)\gamma V^*(s')$$

$$V^*(s) = \max_a Q^*(s,a)$$

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

Value iteration : algorithm for finding the optimal policy

Inputs: - set S of states
- set A of actions
- P is a state transition function specifying $P(s'|s,a)$
- R is a reward function $R(s,a)$

Outputs: - $\pi^*(s)$, the optimal policy
- $V^*(s)$, the optimal value function

} approximately

Algorithm:

1. Assign $V_0^\pi(s)$ arbitrarily
2. For each k and s:

$$V_k^\pi(s) = \max_a Q_k^\pi(s,a)$$

$$Q_k^\pi(s,a) = R(s,a) + \gamma \cdot \sum_{s'} P(s'|s,a) \cdot V_{k-1}^\pi(s')$$

For each s:

$$\pi(s) = \arg\max_a Q^{\pi}_{\pi_{max}}(s,a)$$

with $$Q^{\pi}_{\pi_{max}}(s,a) = R(s,a) + \gamma \cdot \sum_{s'} P(s'|s,a) \cdot V^{\pi}_{\pi_{max}}(s')$$

b)

s	A	$P(\text{healthy}'|s,A)$	$P(\text{sick}'|s,A)$	R
healthy	party	0,7	0,3	10
healthy	relax	0,95	0,05	7
sick	party	0,1	0,9	2
sick	relax	0,5	0,5	0

1. Assign $V^{\pi}_0(s) = 0$

$V^{\pi}_0(\text{healthy}) = 0$
$V^{\pi}_0(\text{sick}) = 0$

2. For each k and s: $V^{\pi}_k(s) = \max_a Q^{\pi}_k(s,a)$

where $$Q^{\pi}_k(s,a) = R(s,a) + \gamma \cdot \sum_{s'} P(s'|s,a) \cdot V^{\pi}_{k-1}(s')$$

$Q_1(\text{healthy}, \text{party}) = R(\text{healthy}, \text{party}) + \gamma \cdot (P(\text{healthy}|\text{healthy}, \text{party}) \cdot V_0(\text{healthy}) + P(\text{sick}|\text{healthy}, \text{party}) \cdot V_0(\text{sick}))$

$= 10 + 0{,}8 \cdot (0{,}7 \cdot 0 + 0{,}3 \cdot 0) = 10$

$Q_1(\text{healthy}, \text{relax}) = 7$

$V_1(\text{healthy}) = \max(Q_1(\text{healthy}, \text{party}), Q_1(\text{healthy}, \text{relax})) = \max(7, 10) = 10$

$\pi(\text{healthy}) = \arg\max(Q_1(\text{healthy}, \text{party}), Q(\text{healthy}, \text{relax}))$
$= \arg\max(7, 10) = \text{party}$

Policy iteration

1. Set π_0 arbitrarily, let $i = 0$
2. Repeat:
   - evaluate $Q^{\pi_i}(s,a)$
   - let $\pi_{i+1}(s) = \underset{a}{\operatorname{argmax}}\, Q^{\pi_i}(s,a)$
   - set $i = i+1$

   until $\pi_i(s) = \pi_{i-1}(s)$

Finding a solution to a set of $|S| \cdot |A|$ unknowns

Solve:

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \cdot \sum_{s'} P(s' | s, \pi(s)) \cdot V^{\pi}(s')$$

for each s:

$$V^{*}(s) = \max_{a} Q^{\pi}(s,a)$$

where $\hat{a} = \underset{a}{\operatorname{arg\,max}}\, Q^{\pi}(s,a)$

where $Q^{\pi}(s,a) = R(s,a) + \gamma \cdot \sum_{s'} P(s' | s, a) \cdot V^{\pi}(s')$

Vb.

1. Assign $\pi(s)$ arbitrarily:

$\pi_1(\text{healthy}) = \text{party}$
$\pi_1(\text{sick}) = \text{relax}$

2. $V^{\pi}(\text{healthy}) = R(\text{healthy}, \text{party}) + \gamma \cdot (P(\text{healthy} | \text{healthy}, \text{party}) \cdot V^{\pi}(\text{healthy}) + P(\text{sick} | \text{healthy}, \text{party}) \cdot V^{\pi}(\text{sick}))$

$= 10 + 0{,}8 \cdot (0{,}7 \cdot V^{\pi}(\text{healthy}) + 0{,}3 \cdot V^{\pi}(\text{sick}))$

$V^{\pi}(\text{sick}) = R(\text{sick}, \text{relax}) + \gamma \cdot (P(\text{healthy} | \text{sick}, \text{relax}) \cdot V^{\pi}(\text{healthy}) + P(\text{sick} | \text{sick}, \text{relax}) \cdot V^{\pi}(\text{sick}))$

$= 0 + 0{,}8 \cdot (0{,}5 \cdot V^{\pi}(\text{healthy}) + 0{,}5 \cdot V^{\pi}(\text{sick}))$

derive them both

$$Q^{\pi}(\text{healthy}, \text{party}) = R(\text{healthy}, \text{party}) + \gamma \cdot (P(\text{healthy} \mid \text{healthy}, \text{party}) \cdot V^{\pi}(\text{healthy}) + P(\text{sick} \mid \text{healthy}, \text{party}) \cdot V^{\pi}(\text{sick}))$$

$$= 10 \cdot 0{,}8 \cdot (0{,}7 \cdot 35{,}71 + 0{,}3 \cdot 23{,}81)$$

QED

$$= 35{,}71$$

$$Q^{\pi}(\text{healthy}, \text{relax}) = 35{,}10$$

$$Q^{\pi}(\text{sick}, \text{party}) = 22$$

$$Q^{\pi}(\text{sick}, \text{relax}) = 23{,}81$$

$$\downarrow$$

$$\hat{a}_{\text{sick}} = \text{relax}$$

XIII

Learning types:

- Supervised learning: learning from a teacher
  ↳ training data includes desired outputs
- Unsupervised learning:
  ↳ training data does not include desired outputs
- Reinforcement learning: learning to act under evaluative feedback

↓

- most like natural learning
- trial and error

→ Reward good behaviour, punish bad behaviour

Reinforcement learning agent

- The learning agent is given the possible states and set of actions it can carry out
- The agent observes the state of the environment and the reward received
- The goal of the agent is to maximize its reward

Experiences: each time the agent has to choose its action requires it to:
- explore to gain knowledge
- exploit discovered knowledge

- The agent initially knows all states S and all actions A.
- The dynamics $P(s'|s,u)$ and $R(s,u)$ are not given to the agent

↓

works like MDP

Reinforcement learning is hard

- A reward may be caused by a lot of actions before
- A long-term effect of an action depend on what the agent will do in the future
- The explore-exploit dilemma: at what time should the agent be greedy or inquisitive?

Main approaches to RL:

- search through a space of policies
- learn a model consisting of state transition function $P(s'|a,s)$ and reward function $R(s,a,s')$, solve this as an MDP
- learn $Q^*(s,a)$, use this to guide action

Suppose we have a sequence of values: $v_1, v_2, \ldots, v_k$ and we want a running average of the first k

$$A_k = \frac{v_1 + \ldots + v_k}{k}$$

TD Formula:

$$A_k = (1-\alpha_k) A_{k-1} + \alpha_k v_k$$
$$= A_{k-1} + \alpha_k (v_k - A_{k-1})$$

$\alpha_k = \frac{1}{k}$

$(v_k - A_{k-1})$: TD Error: specifies how different the new v_k is

Q-learning: an agent tries to learn the optimal policy from its history

$$\langle s_0, a_0, r_1, s_1, a_1, r_2, \ldots \rangle$$

Experience $= \langle s, a, r, s' \rangle$

Q-learning uses temporal differences to estimate the value of $Q^*(s,a)$

↓

the agent maintains a table $Q[S,A]$ where $Q(s,a)$ represents its current estimate of $Q^*(s,a)$

↓

$\langle s, a, r, s' \rangle$

an experience provides one data point for $Q(s,a)$

↓

shows that the agent received the future value of $r + \gamma V(s')$

$V(s') = \max_{a'} Q(s',a')$ → the actual current reward plus the discounted estimated future value

↳ data point is called a return

↓

We can also use the experience directly on $Q(s,a)$:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

Algorithm:

1. initialize $Q(s,A)$ arbitrarily
2. Observe s
   Repeat forever:
   - select and carry out an action a
   - observe reward r and state s'
   - $Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$
   - $s \leftarrow s'$

It does not yet explore knowledge

Exploration strategies:

- ε-greedy strategy: choose random action with probability ε or best action with probability $1-\varepsilon$
- softmax: choose action a with probability:

$$\frac{e^{Q(s,a)/T}}{\sum_u e^{Q(s,a)/T}}$$

where T is the temperature

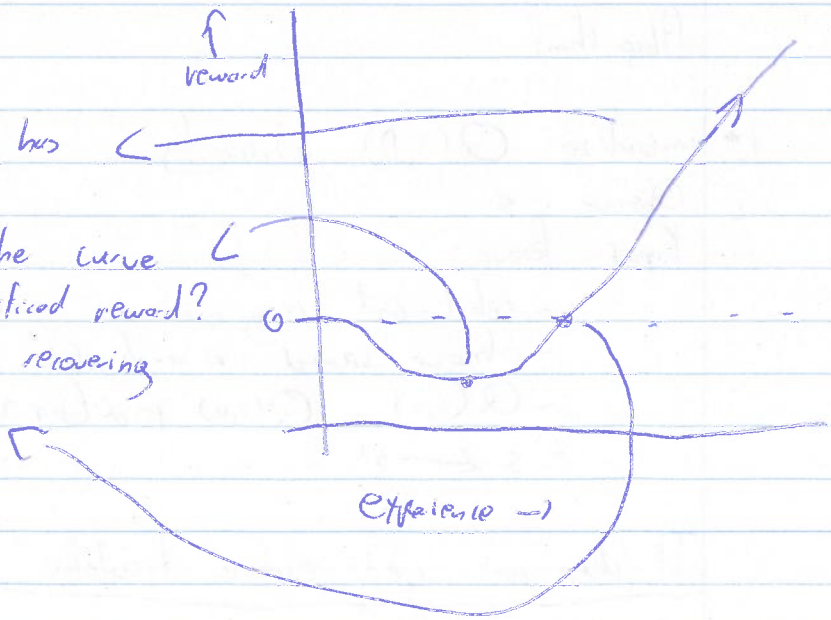
- optimism in the face of uncertainty: initialize Q to values that encourage exploration
- upper confidence bounds: take into account average + variance

Problems with Q-learning:
- it does one backup between each experience
- it learns separately for each state

Evaluating RL algorithms: use the cumulative reward (plot)

Three statistics:

- slope after algorithm has stabilized
- minimum of the curve
  ↳ how much sacrificed reward?
- time it takes for recovering (0-crossing)



SARSA: on-policy learning

↓

also uses a' in experience: $\langle s, a, r, s', a' \rangle$ to update $Q(s,a)$

↳ remember previous action a'