Riassunto Intelligenza Artificiale

Prima parte (6 CFU): Dall’inizio fino al valore dell’informazione;

Seconda parte (3 CFU): Dalla teoria delle decisioni fino alla fine.

# Introduzione

L’intelligenza artificiale è definibile come la cooperazione tra le capacità umane e quelle dei programmi, tutt’oggi però non ha ancora una definizione precisa. Il primo a chiedersi se i computer fossero in grado di pensare in modo autonomo è Alan Turing ma la sua nascita effettiva è nel 1956 in un seminario in cui hanno partecipato i college, le università e le corporation più importanti del mondo. La definizione data all’intelligenza artificiale è l’abilità dei programmi di svolgere problemi e ragionamenti tipicamente umani in un dato dominio, da quello finanziario a quello medico. Da questa definizione si sono sviluppate due ipotesi:

* L’ipotesi forte è la creazione di una vera e propria mente artificiale in grado di pensare e agire come un essere umano;
* L’ipotesi debole è la creazione di programmi che hanno compiti intelligenti in ambiti ristretti e a supporto dell’attività umana.

La strada che tuttora si segue è l’approccio debole, a sua volta diviso in:

* approccio simbolico: ragiona manipolando simboli comprensibili all’uomo ed elaborandoli con appositi algoritmi;
* approccio sub-simbolico: cerca di emulare lo scambio di informazioni tra cellule, da esse vengono le reti neurali, dei neuroni artificiali che si scambiano informazioni tra loro, il problema è esse danno un risultato senza indicare come lo hanno ottenuto. E’ iper specializzata nel proprio campo di utilizzo.

## Categorie di un’IA

L’IA può essere vista come una definizione di 4 concetti:

* Azione umana (Imitation game): capacità di una macchina di confondersi per un umano, dimostrando la sua intelligenza;
* Ragionamento umano: viene preso in considerazione nei casi di studio riguardanti le neuroscienze;
* Ragionamento razionale (lettura della mente): date delle regole, vengono calcolate delle soluzioni razionali per risolvere un problema, non si può considerare un comportamento intelligente dato che si basa su procedimenti logici ( i quali non ammettono eccezioni);
* Azione razionale: è molto complessa e più contro il ragionamento umano/razionale stesso, quest’ultimo non è una necessità ma, se presente, deve essere al servizio di questo tipo di azione.

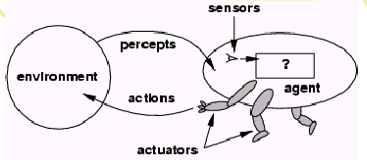
L’IA è a sua volta divisa in settori:

* Rappresentazione della conoscenza: rappresentazione di ciò che si sa e si impara;
* Ragionamento automatico: utilizzo della conoscenza per la produzione di conclusioni;
* Machine Learning: apprendimento di nuova conoscenza;
* Elaborazione del linguaggio naturale: comprensione e interazione direttamente nel linguaggio umano;
* Visione artificiale: riconoscimento di oggetti nell’ambiente in cui si opera;
* Robotica: manipolazione di oggetti nell’ambiente operativo;
* Riconoscimento e sintesi del parlato;

## Successi dell’intelligenza artificiale

Uno dei primi successi dell’IA è la vittoria negli scacchi e in altri giochi da tavolo o enigmistici, in particolare la dama è una gioco già risolto: la sua “bassa” complessità ha permesso ai programmi di considerare tutte le possibili mosse e le relative contromosse effettuabili dall’avversario, andando così a rispondere nel modo migliore e quindi a non permettere più nessuna vittoria da parte di umani. La guida automatica è basata su modelli sub-simbolici e permette il riconoscimento di pedoni, ciclisti e tutto ciò che è presente in strada, il riconoscimento tramite immagini permette il recupero di informazioni attraverso delle foto e la pubblicità dei contenuti streaming agli utenti dati i loro gusti, come nel caso di Netflix. Uno dei casi più famosi è Watson, un calcolatore di IBM che è riuscito a vincere un complesso gioco televisivo in cui bisognava dare delle domande a delle risposte date. Tutt’oggi è utilizzato in vari campi, medici e non. Ritornando a parlare dei giochi da tavolo, nel 2016 AlphaGo sconfigge il campione di Go (dama cinese), un gioco da tavolo orientale dalla grande complessità e quindi considerato irrisolvibile nel breve periodo, grazie al progresso tecnologico e a nuovi algoritmi.

# Agenti Intelligenti

Un’agente intelligente è un’entità che, data una sequenza di stimoli percepiti attraverso sensori, restituisce in output un’azione grazie agli attuatori, il suo obiettivo è massimizzare l’utilità e ridurre i costi. L’agente esegue i suoi compiti in un ambiente, esso può essere modificato e allo stesso tempo può modificare lo stato dell’agente stesso. Attraverso i sensori, l’agente raccoglie informazioni dall’ambiente, esse vengono processate per scegliere l’azione più adatta alla situazione presente.

## Agenti Razionali

Esempio: Vi sono due stanze A e B da pulire con un aspirapolvere, esso percepisce come input la stanza in cui è presente e se si trova in una zona sporca o meno, l’obiettivo è modellare una sequenza di azioni per pulire entrambe le stanze. L’aspirapolvere può andare a sinistra (LEFT), a destra (RIGHT), aspirare (SUCK) o non fare nulla (NOOP). Per prima cosa bisogna evitare che vada a sbattere contro i muri, quindi se si seleziona LEFT quando si è nella stanza A, non succede nulla (La stessa cosa vale anche quando si fa RIGHT in B). L’operazione SUCK si attiva quando l’aspirapolvere si accorge di essere sopra una zona sporca, tuttavia non comporta problemi anche se succede sopra una zona pulita. L’agente in questione è quindi razionale, ovvero che, data una sequenza di percept, seleziona l’azione che permette di massimizzare il risultato, tuttavia non sono perfetti dato che hanno risorse limitate (in questo caso la batteria), l’unica cosa che possono fare è “comportarsi bene”. Un agente razionale non è onnisciente, può eseguire azioni che permettono la modifica delle percezioni future (la cosiddetta “esplorazione”) in modo da ottenere informazioni utili ed è autonomo quando è in grado di apprendere, ricordarsi delle vecchie esperienze e adattarsi di conseguenza.

## Architettura di un’agente

L’architettura di un agente è dettata da PEAS essa indica tutto ciò che è necessario al funzionamento di un agente:

* P: indica la misura di performance, ovvero il criterio oggettivo per misurare il successo dell’agente;
* E: indica l’ambiente in cui opera l’agente;
* A: indica gli attuatori;
* S: indica i sensori.

Esempio: nel caso del taxi automatico, la misura di performance è la sicurezza dato che è adibito al trasporto di persone, esso deve essere anche abbastanza veloce da soddisfare le richieste degli utenti, tuttavia questo è in contrasto con la prima misura. L’ambiente in cui opera è la strada, in essa dei sensori devono misurare la presenza di cartelli o persone e quindi agire di conseguenza frenando (il movimento infatti è dato dagli attuatori).

## 

## 

## Tipologie di ambienti

Un ambiente può essere classificato in molti modi:

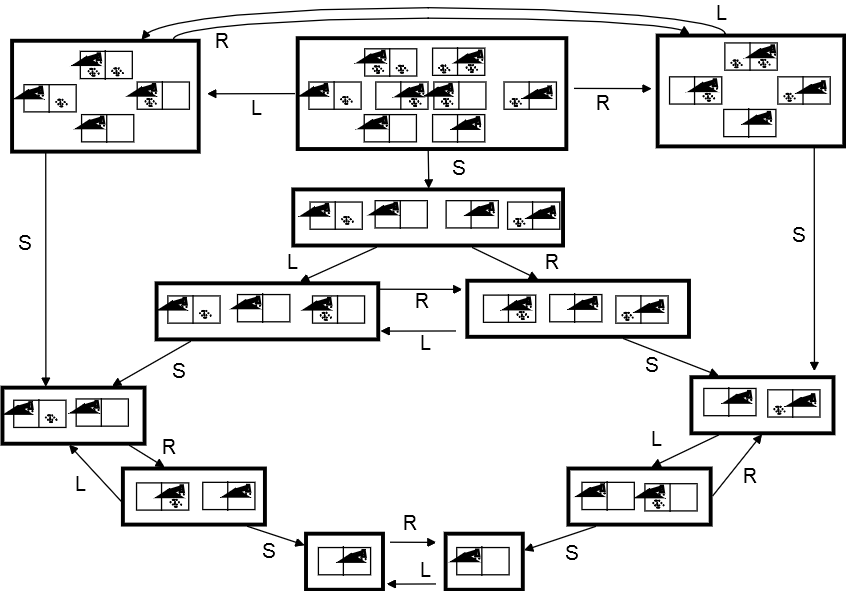
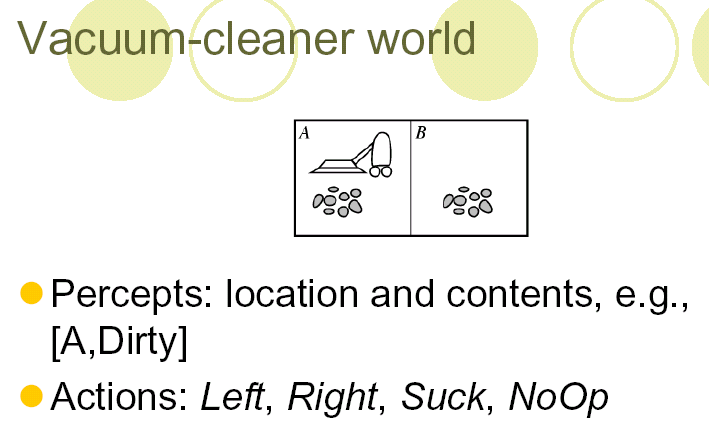
* Totalmente osservabile: l’ambiente è totalmente sensorizzato e quindi si ha una completa conoscenza dello stesso;
* Parzialmente osservabile: non si ha una completa conoscenza dell’ambiente in questione;
* Deterministico: gli stati successivi vengono determinati da quello corrente e dall’azione eseguita;
* Stocastico: lo stato successivo non è determinato dalle azione ma da probabilità;
* Episodico: le scelte non influiscono sulle situazioni future;
* Sequenziale: le scelte influiscono sulle situazioni future;
* Statico: l’ambiente non cambia nè nel tempo né a causa delle azioni dell’agente;
* Dinamico: l’ambiente cambia nel tempo e a causa delle azioni dell’agente (è detto semidinamico quando la causa dei cambiamenti è data solo dall’agente);
* Discreto: vi è un numero finito di cose chiaramente percepibili e modificabili con le azioni;
* Continuo: vi è un numero indefinito di cose chiaramente percepibili e modificabili con le azioni;
* Mono Agente: vi è un solo agente che opera su di esso;
* Multiagente: vi sono più agenti (anche di diversi tipi) che operano nell’ambiente;

Di seguito vi sono degli esempi:

| Ambiente\Esempio | Solitario | Backgammon | Shopping online | Taxi |
| --- | --- | --- | --- | --- |
| Completamente osservabile? | No | Sì | No | No |
| Deterministico? | Sì (carte tutte scoperte) | No | Parzialmente | No |
| Episodico? | No | No | No | No |
| Statico? | Sì | Parzialmente | Parzialmente | No |
| Discreto? | Sì | Sì | Sì | No |
| Monoreagente? | Sì | No | No | No |

## Ricerca nello spazio degli stati

Molti problemi in intelligenza artificiale posseggono configurazioni dette stati che lo descrivono, la ricerca di questi ultimi avviene utilizzando appositi algoritmi detti strategie di ricerca.

Esempio: un robot aspirapolvere deve pulire l’ambiente strutturato come in figura, supponendo che l’ambiente sia deterministico e completamente osservabile utilizzando sensori e videocamere, l’obiettivo è trovare la sequenza di azioni per la pulizia che permetta di massimizzare il risultato e/o diminuire i costi (la batteria in questo caso). Dal momento che l’ambiente è deterministico, l’aspirazione in una stanza determina che questa è pulita e quindi si può passare alla successiva, il risultato è [DirtyA,CleanA,DirtyB,CleanB] se l’agente parte dalla stanza A, se così non fosse, l’agente non aspira completamente e quindi dovrebbe domandarsi se la stanza è sporca fino a quando non è pulita. In ambiente deterministici e completamente osservabili la soluzione è “offline”, ovvero calcolata prima dell’esecuzione delle azioni. 

### Problemi a stato singolo

I problemi a stato singolo sono problemi in cui all’esecuzione di date azioni si passa da uno stato a un altro, infatti a partire dagli stati di un dato problema è possibile costruire un grafo in cui i nodi sono gli stati stessi, gli archi sono le azioni che permettono la transizione da uno stato all’altro e i pesi di questi ultimi indicano il costo dell’azione associata, questa struttura è possibile quando il problema ha un ambiente completamente osservabile e deterministico. Un trucco simile è possibile anche quando l’ambiente non è completamente osservabile, quest’ultimo avrà molti più stati del precedente anche se, esplorando il problema, è possibile ridurre il numero, questo tipo di problemi è detto a stati multipli.

### Definizione di un problema

Un problema è definito da 4 elementi:

* *Stato iniziale*: stato che indica il punto di partenza del problema;
* *Azioni o operatori*: funzioni che, dato uno stato, ne restituiscono un altro (azioni deterministiche) o un’insieme di questi ultimi (azioni non deterministiche);

Azioni deterministiche: f:S→ S

Azioni non deterministiche: f:S→

* *Goal test*: Test che indica se lo stato in cui siamo è quello di arrivo, esso può essere esplicito quando è dichiarato e implicito quando vi sono condizioni che bisogna soddisfare;
* *Costo*: costo delle azioni che influenzano il modus operandi dell’agente, esse sono solitamente maggiori o uguali a 0;

Una soluzione è una sequenza di azioni che porta l’agente dallo stato iniziale a uno degli stati obiettivo, ognuna ha il suo costo e tra tutte si va a cercare quella ottimale, cioè con costo minimo. Di conseguenza, la soluzione ottimale in un grafo di stati è data dal cammino minimo dallo stato iniziale a uno stato finale, in questo modo è possibile risolvere in modo semplice tutti i problemi di questo tipo, tuttavia dato che lo spazio degli stati può essere enorme o infinito, costruire un grafo potrebbe risultare impossibile, in alternativa si può costruire solo il pezzo più rilevante attraverso particolari strategie di ricerca. La struttura dati che permette di visitare gli stati di un problema è il cosiddetto albero di ricerca: la root dell’albero è rappresentata dallo stato iniziale e non ha genitore, tutti gli altri ne hanno uno e hanno tanti figli quante possibili azioni, di ogni nodo dell’albero si tiene conto della profondità, ovvero il costo per arrivare lì dalla root. L’espansione di un nodo consiste nel generare figli applicando tutte le possibili azioni su esso, da esse vengono escluse tutte le azioni che portano al padre o allo stato iniziale per evitare ridondanze. L’albero viene continuamente espanso fino a quando non si raggiunge l’obiettivo richiesto,l’ordine di espansione determina la strategia di ricerca, ce ne sono molte e differenti caratteristiche. Le strategie sono valutate secondo 4 parametri:

* Completezza: una strategia è completa quando, se esiste una soluzione è in grado di trovarla, quindi non perdo nessuna soluzione.
* Complessità in tempo: numero di passi necessari per arrivare alla soluzione
* Complessità in spazio: numero massimo di nodi che dovrà tenere in memoria la strategia per arrivare alla soluzione. (vero problema)
* Ottimalità: se è sempre in grado di darmi la soluzione ottimale.

Le complessità sono il vero problema e dipendono dalla grandezza dell’input, la regola generale è che la complessità esponenziale in tempo è accettabile ( in quanto non si può fare nulla), quella in spazio invece non è accettabile, quindi preferisce sempre una soluzione con complessità in spazio lineare altrimenti la situazione diventa critica. La situazione migliore sarebbe avere una situazione con basse complessità e con una strategia completa e ottimale, ma questo non è possibile, quindi si devono trovare dei compromessi valutando i parametri con delle misure:

* b-branching maximum factor dell’albero di ricerca: fattore di diramazione massimo.
* d-depth : Profondità della soluzione a costo minimo.
* m-maximum: Profondità massima della strategia raggiunta durante la ricerca.

## Ricerca non informata (cieca)

Le ricerche cieche sono strategie di ricerca che scelgono il nodo da espandere in base al livello di profondità, esse possono essere:

* Ricerca in ampiezza (BFS)
* Ricerca in profondità (DFS)
* Ricerca ad apprendimento iterativo (IDS)

Vengono usate per confronto con strategie informate invece più utilizzate.

#### BFS (Breadth First Search)

BFS espende prima i nodi ai livelli più bassi, esso viene implementato con una coda, non permettendo in questo modo di espandere nodi appartenenti a livelli superiori se ve ne sono ancora da visitare in quello attuale (in caso di parità di livelli, la scelta dipende dall’implementazione). Questa strategia di ricerca è detta esaustiva in quanto cerca tutti i nodi su tutti i livelli per trovare la soluzione, per questo motivo è completa e ottimale (dato che i costi sono tutti uguali). BFS ha però problemi riguardanti le complessità: infatti è O(bd+1) sia in tempo che in spazio (equivalente al numero di nodi), questo succede perché nel caso peggiore la soluzione si trova nelle foglie dell’albero, quella in spazio è particolarmente critica perché rischia di saturare la memoria (il numero di nodi cresce esponenzialmente in base alla profondità). In conclusione, BFS è infattibile a causa delle complessità.

#### DFS (Depth First Search)

DFS espande prima le foglie dell’albero, partendo quindi dal nodo più profondo dato che viene implementata utilizzando uno stack in cui inserire i nodi da espandere. Questa strategia di ricerca può avere problemi a lungo termine in quanto è possibile andare in loop oppure di avere stati non più espandibili, per risolvere ciò si utilizza il backtracking, ovvero si torna a una possibilità precedente. Rispetto alla BFS, nella memoria sono presenti solamente i nodi di un dato cammino, lo spazio occupato è infatti b\*m. Al costo dell’ottimalità e della completezza (quest’ultima fallisce in caso di cammini infiniti), DFS permette uno spazio lineare ma un tempo terribile (equivalente a O(bm)) nei casi in cui m sia molto più grande di d, rimane tuttavia veloce nelle soluzioni dense.

#### IDS (Iterative Deepening Search)

IDS è una strategia di ricerca nata per ovviare alle mancanze di DFS, l’idea infatti è mettere un limite non superabile alla lunghezza dei cammini, questo algoritmo è infatti chiamato ricerca con cut-off. Il funzionamento è analogo alla DFS con la differenza che, superato il limite, si torna indietro. Questo limite viene fatto crescere a ogni iterazione e viene nuovamente svolta la ricerca dall’inizio. Al costo di una maggior complessità in tempo (causato dall’ultima iterazione, ovvero quella dominante, tuttavia rimane nello stesso ordine di grandezza di DFS), si ottiene ottimalità e completezza, in quest’ultimo caso perchè, se la soluzione è a un dato livello, prima o poi verrà trovata impostando quel livello come cut-off, la complessità in spazio rimane invariata rispetto alla DFS.

### Ricerche informate (euristiche)

Le ricerche informate sono strategie in cui vengono utilizzate delle informazioni per trovare la soluzione (solitamente i costi per arrivare a essa) esse sono anche dette strategie Best First Search, ovvero che espando prima i nodi migliori. Ogni nodo ha una funzione di valutazione associata il cui scopo è quello di calcolare il valore reale di una soluzione, sono solitamente funzioni e quindi conviene espandere i valori più bassi/alti a seconda dell’applicazione.

f:S→ (data una funzione f e l’insieme di input S, vengono restituiti tutti i possibili valori razionali positivi)

L’implementazione di queste strategie avviene attraverso una coda di priorità, essa permette di scegliere i valori migliori in poco tempo. Le strategie di questo tipo sono;

* Costo uniforme: La soluzione viene calcolata basandosi sulla funzione f;
* Greedy: la soluzione viene calcolata attraverso una funzione euristica, permette di capire quanto manca alla soluzione;
* A\*: calcola la soluzione utilizzando sia f, sia la funzione euristica.

#### Strategia a costo uniforme

Nella strategia a costo uniforme, la funzione f determina il costo di un cammino a partire dalla radice fino a un dato nodo, si può quindi dire che: dati due nodi n, p tali che p=padre(n), f(n)= f(p) + costo(p,n), il costo è zero nel caso in cui n=p.

Questa strategia permette di scegliere un nodo avente la f più bassa, per farlo si utilizza la coda di priorità citata prima, essa estrae un nodo con una tempo O(logn). Questa strategia degenera in BFS nel caso peggiore dal momento che i nodi di un dato livello dipendono da quello precedente, nel caso medio invece i costi sono più contenuti. L’algoritmo è completo e ottimale dal momento che trova sempre la soluzione con costo minore, per quanto riguarda la complessità, invece, si ha O() (dove c è il costo della soluzione ottimale ed e è il numero di step) sia in tempo che in spazio, questo è un problema nel caso peggiore, dato che il numero di nodi saturerebbe la memoria. Questo algoritmo ha un problema: non è possibile capire quanto manca alla soluzione, per farlo infatti bisogna utilizzare le strategie greedy.

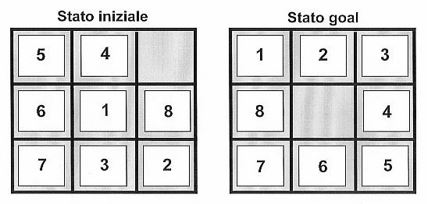
#### Strategia greedy

Le strategie greedy sono quelle in cui, per capire quanto manca alla soluzione, viene utilizzata una funzione euristica h, essa non è uno standard (dato che varia a seconda del problema) e quindi non vi è una soluzione comune, tuttavia vi è un approccio unico per trovarla:

* Si rilassano alcune regole per facilitare un dato problema, meno regole rilasso, più l’euristica sarà precisa;
* Si risolve il problema facilitato.

L’euristica è importante perché permette di diminuire la complessità di un problema, esse non sono funzioni di costo bensì delle stime, per trovarne di sensate si fanno delle sottostime, ovvero dei valori più bassi rispetto al costo vero. L’euristica si dice ammissibile quando è una sottostima della soluzione ottimale, in presenza di più di esse di un dato problema, quella più grande è quella che domina dato che è più simile al costo vero.

Esempio: gioco delle tessere

Una possibile regola da rilassare e quella riguardante il movimento delle caselle: supponiamo sia possibile il movimento libero delle stesse, se consideriamo una configurazione iniziale come in figura, per ottenere la configurazione finale si devono fare 7 mosse, questo è il numero minimo di mosse da fare per arrivare alla soluzione. Supponiamo ora di ridurre il rilassamento: le tessere possono ora muoversi in orizzontale o in verticale di uno senza considerare se si è vicini allo spazio vuoto, qui per calcolare l’euristica si utilizza la distanza di Manhattan, ovvero la somma della distanza orizzontale e verticale tra due punti. Con la configurazione iniziale data in precedenza si ottiene 13 come euristica, essa è una sottostima più grande della precedenza e quindi più precisa, di conseguenza viene preferita questa. Rispetto alla precedente, greedy tende ad andare in profondità, questa strategia non è però completa, dato che può incappare in loop, nè ottimale, la complessità è O(bm) in tempo e in spazio dato che rischia di andare in ampiezza, questo perchè tutti i nodi sono in memoria, tuttavia con delle buone euristiche è possibile diminuire il tempo.

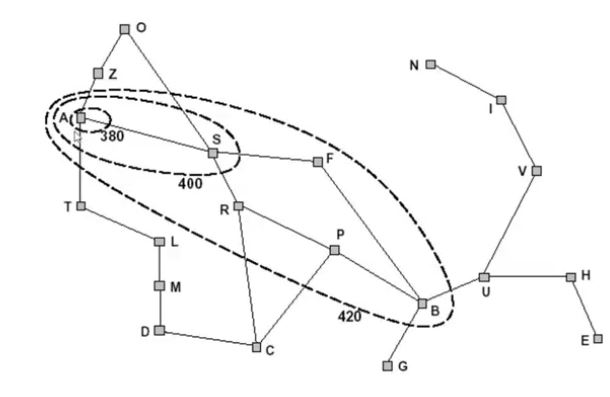
#### 

#### Strategia A\*

La strategia A\* è la combinazione delle prime due: data g(n) come la funzione di valutazione per arrivare a un nodo n partendo da quello iniziale e h(n) come euristica che da un nodo n stima il costo fino al goal, f(n)=g(n)+h(n) permette di calcolare un costo stimato dal nodo iniziale fino al goal passando per un nodo n. La funzione euristica utilizzata da A\* deve essere il più possibile vicina al costo reale, ad esempio: dato un goal G2 non ottimale e G1 ottimale, consideriamo un nodo n non espanso nel percorso di g1, esso verrà selezionato sempre perchè si trova nel percorso ottimale, si può quindi dire che:

* f(G2)=g(G2)dal momento che la funzione euristica ha valore 0 sui goal;
* g(G2)>g(G1) dato che G2 non è ottimale;
* g(G1)>=f(n) se l’euristica è ammissibile e quindi una sottostima della soluzione ottimale.

Da quest’ultimo punto si può capire che G2 non verrà mai preso in considerazione, dal momento che il suo costo è più alto di quello di n. Con un’ euristica ammissibile, la strategia A\* restituisce sempre la soluzione ottimale, tuttavia è bene anche avere la consistenza, ovvero che la disuguaglianza triangolare viene soddisfatta. Un’euristica consistente è ottimale (tuttavia non è vero il contrario) e permette di ottenere una f non decrescente, per renderla tale si utilizza l’equazione di Pathmax: data un h ammissibile e due nodi n e m in cui m è successore di n, se f(n) è maggiore di f(m) per via di una sottostima minore nel secondo, si calcola f(m) come il massimo tra f(n) e g(m)+h(m), in questo modo f diventa non decrescente: dato c=f(n) e d=g(m)+h(m), f(m)=max(c,d).

A\* è completo a meno che non vi siano infiniti nodi con costo f inferiore a quello ottimale, la complessità in tempo e in spazio è esponenziale ma dipende dall’ euristica scelta e, se quest’ultima è ammissibile, la soluzione trovata risulta sempre ottimale. Un'euristica più precisa permette una diramazione minore, quindi se equivale alla scelta esatta, verrà scelto sempre e solo il cammino minimo. Una cosa che è possibile notare in A\* è che vengono espansi prima i nodi con una f minore o uguale a un certo valore, quest’ultimo è la frontiera di quello che viene chiamato contorno di espansione, una regione in cui tutti i nodi all’interno vengono espansi prima degli altri, la sua grandezza dipende da quanto è focalizzata l’euristica. Il problema di A\* riguarda la complessità in tempo e in spazio, per risolverlo si utilizza una strategia IDS che utilizza l’informazione f=g+h, chiamata A\* iterativo (IDA\*).

#### Strategia A\* iterativo (IDA\*)

IDA\* è una visita DFS iterata in cui il limite non è nei livelli ma nella funzione f, infatti se viene superata si ferma ed esegue il backtracking, si passa all’iterazione successiva se non vengono trovate altre soluzioni, essa utilizzerà il valore più piccolo della f di tutti i nodi come nuovo cut-off (corrispondente alla funzione f che ha superato il limite attuale). Completezza e ottimalità vengono mantenute per via della funzione f, in spazio si occupa meno memoria dato che si basa sulla DFS e in tempo conta l’ultima iterazione.

##### Implementazioni di IDA\*

###### Ricerca con alberi

L’implementazione IDA\* su alberi permette di mantenere separati i cammini, infatti ogni nodo dell’albero rappresenta il cammino dalla radice fino a esso, tuttavia differenti nodi possono rappresentare lo stesso stato, essi vengono generati tramite differenti cammini.

| function tree\_search(problem)  open\_list ← initial\_state(problem)  Loop do  if empty(open\_list) then return failure  else node ← select\_and remove(open\_list  if goal(node) then return solution  else children ← expand(node)  add\_to(open\_list, children)  End loop |
| --- |

###### Ricerca con grafi

L’implementazione IDA\* con grafi permette di evitare gli stati ripetuti, per farlo si deve occupare della memoria perché si deve tenere conto dei nodi chiusi (oltre a quelli aperti) dato che possono essere generati una seconda volta, in più c’è bisogno di ulteriori ricerche per verificare questi ultimi (se viene trovato un cammino migliore per un nodo chiuso, lo devo riaprire per ricalcolare gli altri, in più si deve aggiornare il riferimento al padre). Un’euristica consistente permette di non riaprire più i nodi una volta chiusi, dato che il cammino migliore è già stato trovato.

| function graph\_search(problem)  open\_list ← initial\_state(problem)  closed\_list ← empty  Loop do  if empty(open\_list) then return failure  else node ← select\_and remove(open\_list)  if goal(node) then return solution  else add\_to(closed\_list,node)  children ← expand(node)  foreach n in children  if n not\_in (open\_list U closed\_list) then  add\_to(open\_list, children)  else update\_cost(n)  if n in closed\_list and improved\_cost then  move(n,closed\_list,open\_list) |
| --- |

### Algoritmi di ricerca locale

Ci sono problemi in cui il cammino è irrilevante e quindi lo stato obiettivo stesso è la soluzione. Esistono però problemi in cui non si ha lo stato obiettivo, quindi bisogna farlo trovare dall’algoritmo seguendo certi vincoli: a seconda del problema si può scegliere lo stato con funzione di valutazione più alta o più bassa. Tengo in memoria solo gli stati necessari a valutare le varie opportunità quindi non hanno bisogno di grandi quantità di memoria da occupare. Il più semplice è l’algoritmo hill climbing, l’idea è di muoversi da uno stato ad un altro quando migliora la sua utilità, considerando una situazione di partenza in cui si deve raggiungere un obiettivo ma si ha una visione limitata: è possibile vedere solamente se le direzioni vanno in salita,in discesa o se rimangono uguali. La scelta migliore è andare in salita, tuttavia ci si dimentica cosa si è fatto fino a quel momento. L’algoritmo va avanti finchè non vede solo discese, quindi cerca di raggiungere il punto più alto possibile:

| hillClimbing(Problem p)  node current, next  current=makeNode(StartState(problema))  while(true)  next=successor(current) con la valore max/min;  if(value(next) < value(current))  return current  else current=next; |
| --- |

L’algoritmo ha però un problema: esso dipende dallo stato iniziale in cui si trova, quindi rischia di fermarsi in un massimo locale e non in quello assoluto, in più può andare all’infinito se vi sono infiniti stati che hanno lo stesso valore. Si può quindi dire che questo algoritmo non garantisce di restituire la soluzione globale, tuttavia funziona correttamente quando vi è un solo punto più alto.

Esempio: problema delle otto regine. si devono piazzare su una scacchiera 8 regine in modo che nessuna sia sotto scacco. Partendo dall’angolo in basso a destra, si hanno 7 possibili posizioni.

#### Approcci al problema dei massimi locali:

* Random restarting: si effettuano più esecuzioni ripartendo da uno stato diverso a caso, si ha meno probabilità di prendere un massimo locale, tuttavia non si ha ancora la garanzia di ottenere la soluzione ottimale.
* Approccio stocastico: invece che andare sempre a scegliere lo stato migliore tra i successori, si sceglie casualmente il successore a patto che sia migliore dello stato attuale, esso non deve essere necessariamente il massimo.
* Simulated Annealing: tecnica che approccia il problema in maniera migliore.

#### Approccio stocastico

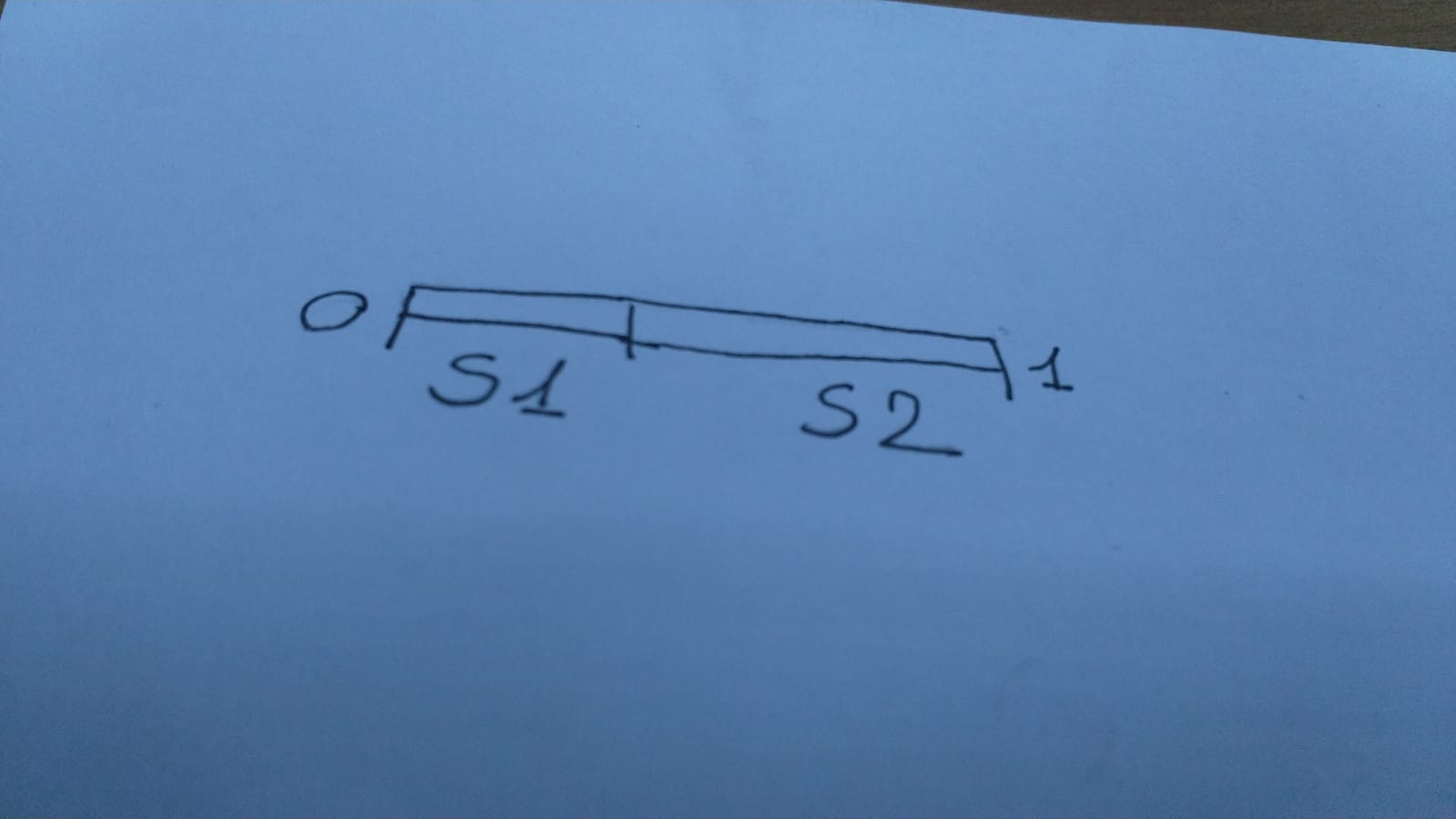
Nell’approccio stocastico si assegna una probabilità a ogni possibile successore il cui valore è migliore di quello attuale, essa viene calcolata nel seguente modo:

P=val(Si)/, detto anche normalizzazione a 1.

L’implementazione avviene generando in modo casuale e uniforme i numeri in un intervallo 0-1, in base a dove cade, si sceglie uno stato anziché gli altri.

Esempio: Dato S0 con valore 10, rappresentante lo stato attuale, e due successori S1 e S2 rispettivamente con valore 12 e 20, le probabilità di questi ultimi sono rispettivamente le seguenti:

P(S1)=12/(12+20)=0,38 P(S2)=20/(12+20)=0,62

L’intervallo 0-1 sarà quindi diviso in 2: una parte iniziale, corrispondente al 38%, in cui vi è l’intervallo di S1, il restante 62% indica invece quello di S2.

Dopo si genera un numero a caso compreso tra 0 e 1, in base a dove cade, si sceglie S1 o S2.

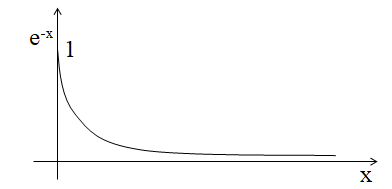
#### Simulated Annealing

L’idea del simulated annealing è quella di scegliere casualmente tra tutte le mosse, anche quelle peggiori, dato che scendendo e risalendo si può trovare la soluzione migliore. Più la mossa è cattiva, più bassa deve essere la probabilità di farla, questo perchè all’inizio dell’algoritmo facilito più le mosse cattive e ne diminuisco la possibilità col passare del tempo, quindi si esplora cosa c’è attorno allo stato di partenza e si prende una decisione verso la fine. Prima cosa da fare è valutare la differenza di energia tra uno stato e il successore, essa indica l’utilità del passaggio di uno stato. La mossa è cattiva se é minore di 0.

*=Value(S’)-Value(S)*

Per diminuire le scelte cattive, si calcola la probabilità con Poisson, in questo modo la probabilità di scegliere queste mosse tenderà a 0 (come ben visibile dal grafico).

***p=***

Per capire se eseguire una mossa cattiva o meno, si calcola , se questo numero è maggiore di 0, vuol dire che la mossa non è cattiva e quindi la eseguo, altrimenti lo si fa con una probabilità p estraendo un numero tra 0 e 1, quest’ultimo caso è simile all’approccio stocastico citato in precedenza. Per diminuire la probabilità delle mosse cattive col passare del tempo, si introduce un altro parametro: la temperatura, inizialmente vengono dati valori molto alti, diminuendo a ogni passo secondo uno scheduling di raffreddamento. Quando la temperatura arriva a 0, l’algoritmo si trasforma in un hill-climbing puro. La probabilità si calcola quindi così:

p=

L’implementazione della temperatura più semplice è data da un array con valori decrescenti, il raffreddamento non deve essere né troppo veloce, dato che in questo caso si esplora poco e non si garantisce che la soluzione trovata sia quella giusta, nè troppo lenta, in tal caso si esplora troppo e potrebbe metterci molto tempo a trovare la soluzione, dato che diventa una ricerca esaustiva.

#### Beam local search

Un’alternativa alla ricerca locale è la Local Beam Search dove si esegue una sorta di hill climbing dove invece di tenere lo stato migliore si tengono k stati migliori in memoria. L’algoritmo funziona restartando su ogni stato dei k scelti, ognuno in modo indipendente dall’altro, la sua versione stocastica seleziona casualmente k stati con una probabilità proporzionale al loro valore.

#### Algoritmi genetici

Gli algoritmi genetici sono algoritmi ispirati alla natura,la ricerca dello stato ottimale non avviene da uno stato solo ma generando i possibili successori da una coppia di stati, essi avranno caratteristiche comuni a quelle dei genitori. Gli stati sono rappresentati come stringhe di bit, rappresentante i cromosomi, i bit sono i geni, i valori di questi ultimi sono gli alleli, uno stato è detto individuo e un insieme di stati è una popolazione. Si fa evolvere la popolazione per ottenerne una migliore con degli operatori in modo da massimizzare la funzione di utilità:

* Selezione: si prende un dato numero di individui (possono anche essere ripetuti) con una probabilità proporzionale alla fitness function, utilizzando la normalizzazione a 1;
* Crossover: gli individui vengono accoppiati, ogni coppia genera due figli con alcuni geni incrociati, solitamente metà da un genitore e metà dall’altro;
* Mutazione: per ogni gene vi è una bassa probabilità che permette di cambiare il suo allele.

Questi algoritmi non sono ottimali, tuttavia hanno la loro utilità.

### Giochi a due concorrenti

I giochi a due concorrenti sono giochi in cui vi è una ricerca di tipo avversario, infatti bisogna considerare che i due avversari cercano di massimizzare la propria utilità utilizzando strategie opposte e spesso imprevedibili, inoltre le mosse di un giocatore dipendono da quelle dell’altro. Dato che può esserci un tempo limite, può essere utile avere delle approssimazioni perché è impossibile, con le risorse di tempo e spazio a disposizione, esplorare tutto lo spazio delle possibili soluzioni. I giochi possono essere:

* deterministici: giochi in cui le mosse hanno risultati univoci;
* chance (con elementi probabilistici): la mossa dipende dal risultato dell’evento probabilistico associato, ad esempio dal lancio dei dadi;
* dimensione perfetta delle informazioni: i giocatori hanno accesso all’intero stato del gioco;
* dimensione imperfetta delle informazioni: non ho accesso all’intera configurazione del gioco.

Alcuni giochi sono detti a zero-sum, essi sono quei giochi in cui la somma dei pay-off dei due avversari è uguale a zero, quindi cosa guadagna uno, perde l’altro e viceversa, l’importante è che esse si annullino. Per valutare una configurazione di un gioco, si utilizzano i seguenti valori;

* Utilità: indica quanto una configurazione è gradita al giocatore, può essere una vincita o un gradimento ed è legata al vincolo di zero-sum;
* Strategia: è la definizione di una mossa per ogni contromossa dell’avversario, esse possono essere pure (deterministiche) o miste, in quest’ultimo caso vi è una distribuzione di probabilità su strategie pure.

Esempio: dato un gioco G e due giocatori M e m aventi rispettivamente le strategie S e s, l’utilità U(S,s) è l’utilità di M, quindi se G è zero-sum, -U(S,s) è l’utilità di m. In parole semplici, M vuole massimizzare mentre m vuole minimizzare.

## Teorema di Minimax

Dato un gioco G finito e zero-sum e due giocatori M e m, esistono due strategie S\* e s\* ottimali e un numero Vg detto valore di Minimax (cioè la scelta quando M ed m giocano entrambi al massimo delle loro possibilità).Se m gioca al massimo (quindi utilizza s\*) e M no, l’utilità U(S,s\*) di M è minore o uguale a Vg, al contrario se M gioca al massimo (utilizzando S\*) e m no, l’utilità U(S\*,s) è maggiore o uguale a Vg. Si può quindi dire che Vg=U(S\*,s\*), quindi i giocatori utilizzando le strategie ottimali e quindi giocano al meglio. Nel caso in cui G sia a informazione perfetta, le strategie S\* e s\* sono pure.

#### Alberi di giochi

Per risolvere questo tipo di giochi si utilizzano gli alberi di giochi, in essi i nodi sono delle possibili configurazioni iniziali, intermedie e finali del gioco e gli archi che li connettono sono le mosse che mi portano da un nodo all’altro. Ogni livello è associato a un giocatore, essi sono alternati e contengono tutte le possibili mosse che possono fare. Quando si raggiungono dei nodi terminali si decide chi vince e si può valutare l’utilità di questi nodi e quindi decidere quanto valgono per il gioco utilizzando i valori 1, -1 e 0 in base al vincitore.

Dopo aver costruito l’albero, si effettuano delle strategie, esse non possono essere dei cammini come soluzione dal momento che i giocatori non hanno tutti i livelli dell’albero, di conseguenza si rappresentano con dei sottoalberi in cui nei livelli di un giocatore si sceglie un ramo, corrispondente alla mossa migliore, in tutte quelle del nemico invece si considerano tutte quelle che può scegliere. Essendo un albero, abbiamo a che fare col branching factor medio b e con la massima profondità h raggiungibile da una data situazione di partenza, le strategie di M e m hanno ordine di grandezza di bh nodi da esaminare. La soluzione sarà un sottoalbero perché ad ogni mossa di un giocatore corrisponde una contromossa dell’altro.

| utility minimax(node s)  if(terminalState(s)) return payoff(s)  else if(M.move==s) return max{minimax(result(a,s)) con a applicabile a s}  else return min{minimax(result(a,s)) con a applicabile a s} |
| --- |

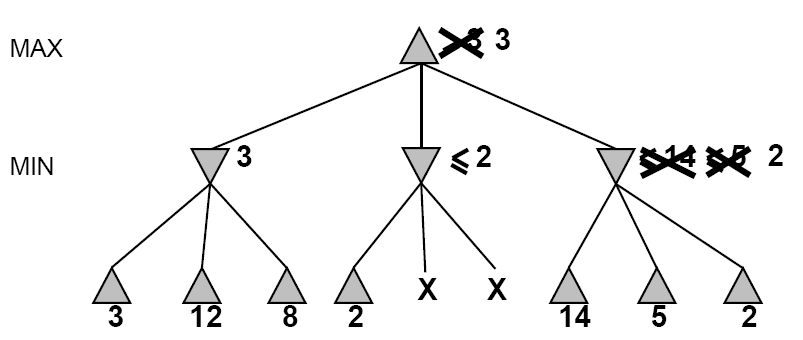
Esempio: dati 2 giocatori M e m che giocano al massimo e l’albero in figura, per determinare una strategia di M, si utilizza l’algoritmo di Minimax: per ottenere il massimo risultato possibile, M e m sintetizzano il valore dei figli sui nodi in cui possono scegliere, scegliendo la mossa che cerca di massimizzare la loro utilità e minimizzando il più possibile quella dell’avversario. L'algoritmo é completo dato che viene costruito l’albero ed in seguito visitato, è esponenziale in tempo ma dipende dall'albero stesso mentre in spazio vengono utilizzati b\*h nodi grazie allo sviluppo in profondità. In presenza di alberi molto grossi non è possibile memorizzarlo interamente, negli scacchi infatti non ci si può permettere di generarlo tutto dato il grande numero di mosse possibili, quindi bisogna utilizzare un cut-off, eseguendo l’algoritmo fino a tale limite (chiamato anche look ahead, ovvero il numero di mosse in avanti che si guardano per fare la strategia). Le foglie ottenute non sono tuttavia valutabili, di conseguenza bisogna utilizzare delle apposite funzioni di valutazione, esse sono solitamente lineari: dato uno stato s, f ne determina le caratteristiche, ognuna con un peso w, ovvero la sua importanza, si può quindi dire che:

E(s)=

Dopo che l'avversario ha scelto la mossa, l'albero di gioco viene ricostruito dal punto in cui l’avversario si era fermato, ora però si ha un livello in più e quindi vi sono più possibilità, l'unico svantaggio é che tutte le mosse dipendono da una data stima, esse però diventano più precise man mano che si va avanti.

#### α-β pruning

Per migliorare ancora, bisogna cercare di tagliar via configurazioni che sicuramente non verranno considerate per far spazio in memoria e utilizzarlo per andare in profondità, per farlo si utilizza la strategia α-β pruning in cui esegue una ricerca in profondità.

Esempio: dato l’albero come in figura, il primo nodo minimo sceglie itera uno alla volta sui figli prendendo il minimo (in questo caso 3), la radice può avere quindi un valore che può essere maggiore o uguale a quest’ultimo dato che è il più piccolo trovato finora. Proseguendo con il secondo nodo minimo, si considera il 2 e quindi il suo valore può essere minore o uguale a questo numero, questo crea una disgiunzione tra quest’ultimo nodo e la radice, quindi gli altri figli non vengono considerati dato che potrebbero avere valori maggiori di 2. Un altro possibile caso di pruning si poteva manifestare nel caso in cui il figlio 2 del nodo minimo più a destra fosse stato a sinistra o al centro, tuttavia non è così e itera tutti i figli scegliendo il minimo. Una volta che i nodi minimi hanno scelto i valori, la root itera su essi scegliendo il massimo, esso corrisponde al valore Vg, Nel caso migliore i nodi massimi sono in ordine decrescente e quelli minimi in ordine crescente, quindi con un pruning fatto in questo modo è possibile persino raddoppiare la profondità di ricerca, quindi il look ahead.

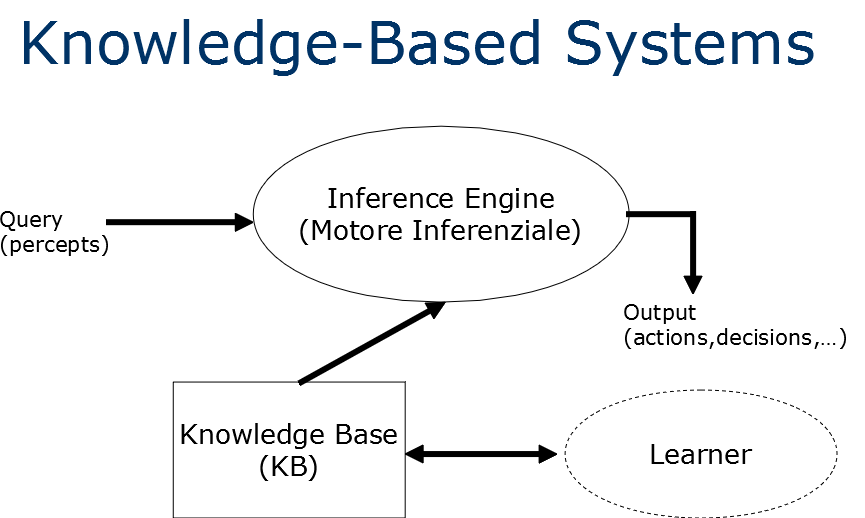
## Rappresentazione della conoscenza e ragionamento (KRR)

La KRR serve per rappresentare la conoscenza attraverso dei modelli di dati e a utilizzarla attraverso date tecniche. Un agente intelligente utilizza KRR per conoscere cose riguardanti un problema e ragionare su esso, per questo può servire apprendere nuove cose da riutilizzare in seguito, i DBMS sanno fare queste due cose. La prima linea di pensiero è stata l’utilizzo della logica matematica, tuttavia presentava problemi dato che non riusciva a gestire operazioni incerte, quindi vengono seguiti i seguenti approcci:

* Ragionamento basato su casi: permette l’apprendimento automatico e naturale;
* Ragionamento probabilistico: permette rappresentazioni grafiche dei modelli.

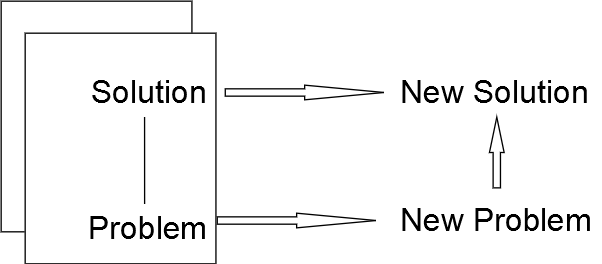
#### Sistemi basati sulla conoscenza (KBS)

KBS è un programma che utilizza la conoscenza per risolvere problemi complessi, esso è formato da:

* Una query, ovvero i percept ricevuti in input;
* Una base di conoscenza KB contenente la conoscenza usata dal sistema per il proprio compito (diagnosi, pianificazione, ecc…); 
* Un motore inferenziale che, sulla base della propria conoscenza e sulle query, compie ragionamenti e fornisce opportune risposte
* Un eventuale modulo di apprendimento (learner) che, se presente, può cambiare la KB modificando la conoscenza del sistema, aggiungendone di nuova o modificando quella obsoleta o inadeguata. Il learner della logica è l’induzione, ovvero la generalizzazione a caratteri comuni.

Esempio: vi sono tre corvi e tutti e tre sono neri, quindi si può dire che tutti i corvi sono neri.

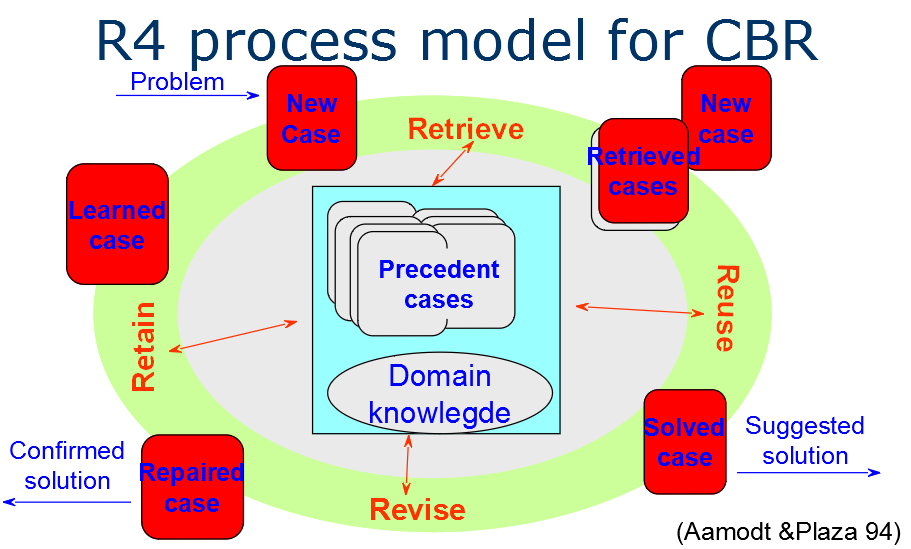
## Case-Based Reasoning (CBR)

La soluzione CBR parte dal presupposto che il sistema può accumulare un certo insieme di esperienza e invece di buttarla via la mantiene così tale e quale, il problema però è codificare tutta questa esperienza in regole e modelli. L’idea utilizzata dal CBR è cercare di sfruttare soluzioni di problemi simili e già risolti (contenute in una libreria apposita) a quello corrente per risolverlo, in questo modo si sfrutta l’esperienza precedente. 

Alcuni esempi si applicazione del CBR sono:

* Classification: dati oggetti con date proprietà, lo si deve inserire nella classe giusta in base a essi; utilizzato solitamente nelle diagnosi mediche e simile;
* Compiling solutions: trovare il modo di spiegare qualcosa con quello che si è già visto;
* Assessing values: vengono valutati oggetti utilizzando riferimenti a oggetti simili;
* Justifying with precedents: i casi attuali vengono valutati prendendo quelli precedenti come riferimento, un esempio è il tribunale anglosassone;
* Evaluating options: si valutano possibili soluzioni e possibili conseguenze con casi passati.

Un sistema CBR è formato dalla libreria di casi risolti e la conoscenza del dominio applicativo, ovvero la conoscenza base senza apprendimento, il suo funzionamento prende in input un nuovo caso da risolvere calcolando la soluzione attraverso 4 regole:

* Retrieve: recuperare nella base di casi l’insieme di casi più simile a quello corrente;
* Reuse: si prende la soluzione del caso più simile e si verifica se è applicabile al caso corrente, se è così, la si propone come soluzione. (questo è però un caso fortunato dato che non sempre è così)
* Revise: se una soluzione non è applicabile per determinati motivi, si prova ad adattarla al problema senza snaturarla, in questo caso si ricorre al dominio; se si riesce ad adattare la soluzione, il caso “riparato” viene mandato in output, altrimenti il sistema fallisce e risponde che non è in grado di risolvere il problema;
* Retain: fase di apprendimento in cui si decide se il nuovo caso risolto deve essere appreso e inserito nella libreria dei casi oppure no; apprendere nuovi casi non è sempre un bene dal momento che può impattare sulle prestazioni.

#### Rappresentazione dei casi

I casi possono essere considerati come oggetti aventi insiemi di attributi (detti anche feature), la soluzione di un dato problema viene infatti calcolata attraverso questi ultimi. Nei casi più specifici di un problema, vi possono essere delle estensioni in base agli stessi attributi, il contesto infatti rimane lo stesso, così come i collegamenti ad altri casi e gli outcome, ovvero i suoi successi e fallimenti, questi ultimi vengono considerati in quanto è possibile utilizzarli per evitare i fallimenti. Rappresentando i problemi come oggetti, si hanno vantaggi sull’ereditarietà, infatti è possibile creare una classe radice contenente gli attributi e, in base al problema, specificare il caso. Gli attributi possono essere discriminanti o meno, essi permettono di indicare la similarità tra due casi, ovvero quanto sono simili.

Esempio: valutazione di auto usate

Le auto sono caratterizzate dall’anno, dal modello, dal codice, dagli optional presenti, dalle condizioni e dal chilometraggio, l’obiettivo è il prezzo. Di questi attributi, quelli discriminanti sono il chilometraggio, il modello, gli optional presenti, le condizioni e l’anno.

## Case Indexing

Una delle prime cose da decidere una volta rappresentati i casi è la loro l’indicizzazione, essa serve per capire come recuperarli e come eseguirne il match, quindi a capire quali features sono importanti e quanto contribuiscono al match. Il modo più semplice per recuperare i casi è inserirli in un database relazionale oppure a oggetti, tuttavia esistono altri modelli di memoria:

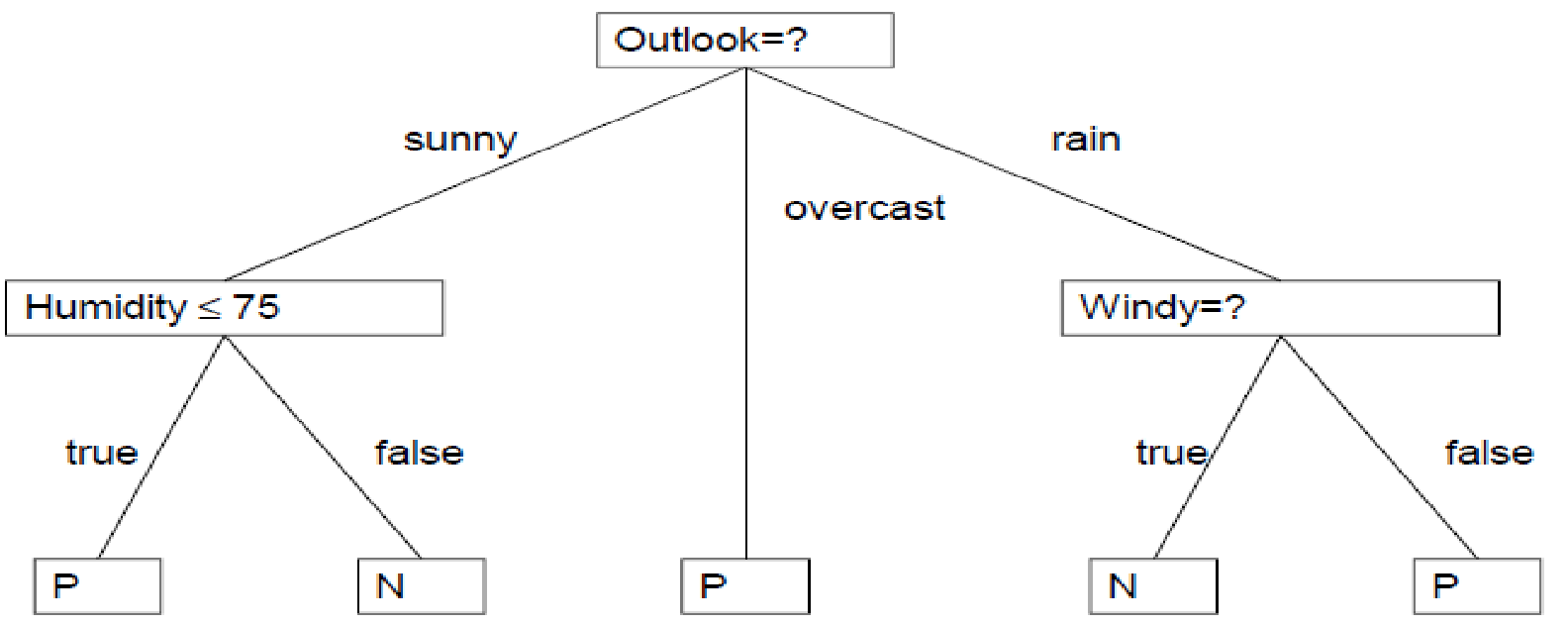
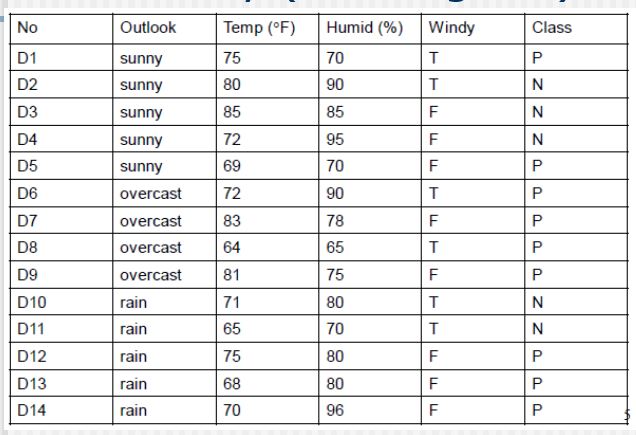
* memorie piatte di tipo associativo: struttura a indice invertito in cui si ha un array di puntatori per ogni coppia feature-valore differente, i casi saranno un insieme di attributi stanziati. Per la ricerca vengono utilizzati dei contatori che indicano quanti casi ci sono per una data corrispondenza feature-valore;
* memorie strutturate (decision trees e k-d tree): Sono utilizzati per problemi di classificazione in cui la classe è la soluzione, esse si trovano nelle foglie.

A seconda del modello che si sceglie, gli algoritmi di ricerca possono essere diversi, dalla ricerca esaustiva a quella binaria con tempi di ricerca inferiori a quella lineare.

#### Alberi di decisione (DT)

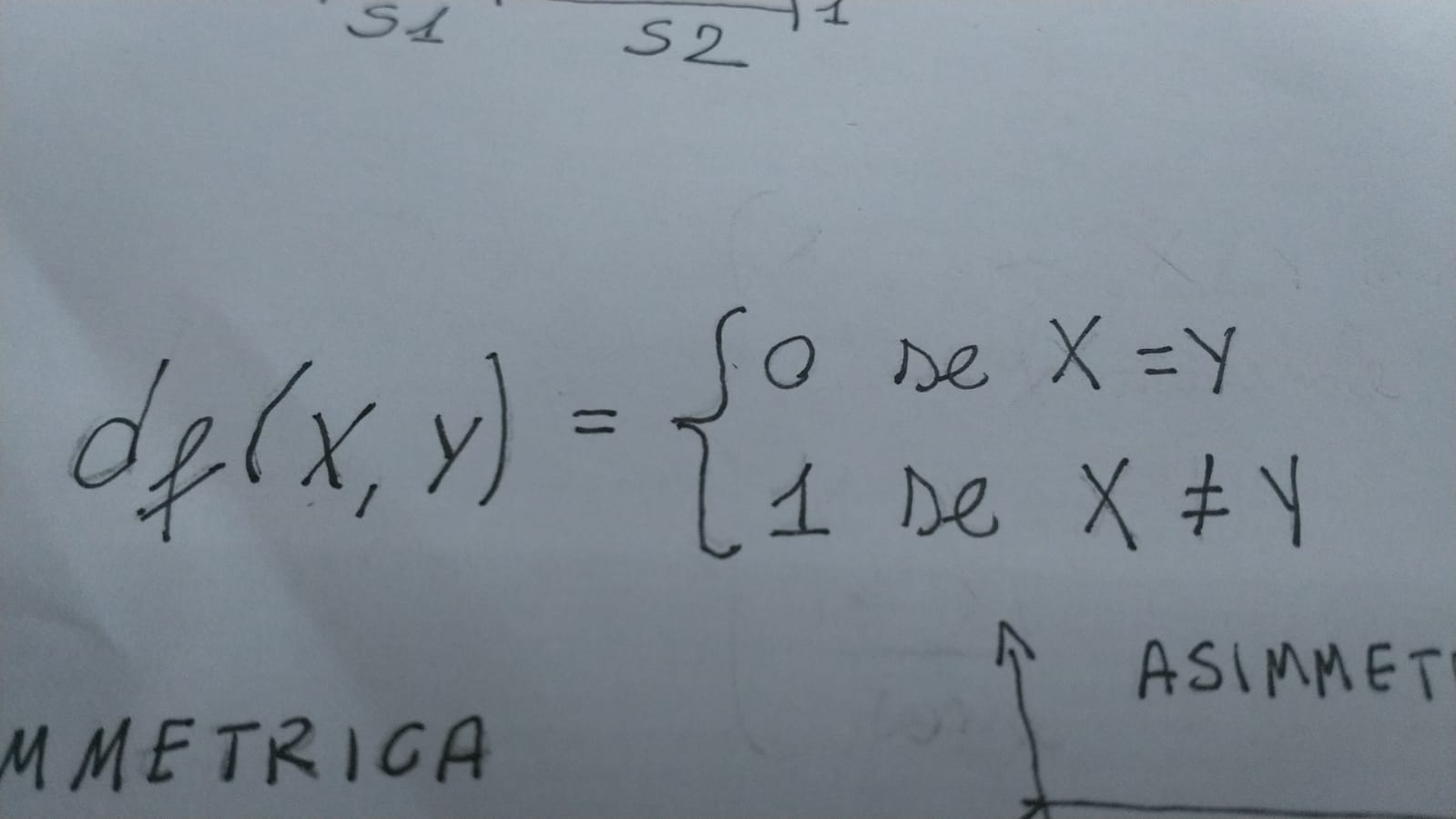
Un DT è una struttura utilizzata nei problemi di classificazione, ogni nodo è rappresentato da un attributo e gli archi uscenti dai possibili valori applicabili su esso, precisamente è formato da:

* Dataset: attributi descrittivi più le classi da assegnare;
* Query: attributi descrittivi;
* Soluzioni: assegnamento di una query a una classe;

Esempio: si considera un albero di decisioni in cui si decide se giocare (P) o meno (N) a tennis in base al meteo, gli outlook utilizzati sono le condizioni meteo, ovvero sole, nuvole o pioggia, temperatura e umidità in percentuale e la presenza di vento attraverso un booleano. La tabella indica i dataset a disposizione, da essi si ricava l’albero di decisione. A seconda dei valori di ogni query, questi ultimi vengono assegnati a ogni classe, questo processo è detto retrieval. Nella pratica tuttavia potrebbero esserci informazioni mancanti, quindi se il caso target è completo, si attraversa l’albero normalmente, altrimenti si prende l’intero sottoalbero per il nodo in cui manca l’informazione, proseguendo normalmente dove invece è completa. Quando in quest’ultimo caso non si raggiunge una soluzione univoca, si ricorre al “voto a maggioranza”, esaminando quanti casi di quelli recuperati danno più supporto ad una soluzione o a un’altra, tuttavia la classificazione può anche essere pesata. Quando il CBR è utilizzato per recuperare casi simili a quello di partenza, si dà una similarità implicita, essa indica quanto sono simili il caso di partenza da quello preso in considerazione, tuttavia è possibile darne anche una esplicita. Per calcolare la similarità esplicita si utilizza la distanza: dati due oggetti X e Y, X è molto simile a Y se e solo se è poco distante da quest’ultimo, quindi si può dire che la similarità è l’inverso della distanza:

S=|1-D|

Ogni caso viene visto come un punto n-dimensionale in uno spazio, dove n è il numero di attributi che lo descrive. Una metrica o distanza su un insieme X è una funzione d:X x X →R che soddisfa le seguenti proprietà:

* Con una distanza uguale a 0, i due casi presi in considerazione sono uguali;
* Bisogna rispettare la disuguaglianza triangolare;
* Non esistono distanze negative.

La distanza discreta o di overlap è data dalla presenza di più funzioni per calcolarla. La distanza è importante perché con essa si esegue il cosiddetto matching tra casi, ovvero il loro confronto.

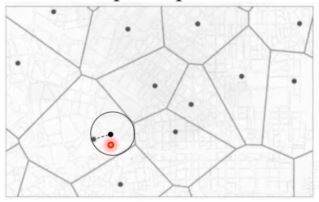
## Matching

Come già detto in precedenza, un caso è paragonabile a un punto avente n dimensioni, una per ogni attributo, quindi è possibile applicare il cosiddetto k-Nearest-Neighbor (k-NN), ovvero i k nodi più simili a quello di partenza, per farlo si seguono i seguenti punti:

* si definisce una funzione di distanza tra i valori di ogni features;
* si calcola la distanza tra il caso in input (target) e quelli nella case memory combinando le distanze sulle features, eventualmente pesandole in base alla loro “importanza”;
* si scelgono (retrieval) i k casi più’ vicini al target e si usano le loro soluzioni come base per quella del target (reuse/revise).

Esempio: con due attributi i casi sono rappresentabili su un piano cartesiano, con k uguale a 1 si restituisce il caso più vicino a quello di partenza, nel caso ci siano distanze eque, invece, è possibile prendere tutti i nodi oppure sceglierne uno arbitrariamente, la similarità è quindi rappresentabile come una circonferenza il cui centro è il caso di partenza.

### Diagramma di Voronoi

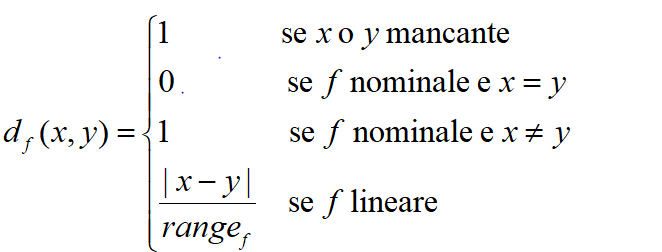
Il diagramma di Voronoi è un diagramma in cui il piano è partizionato in regioni rispetto a un insieme di punti e alla funzione di distanza. Ogni regione è associata a un punto tale che quest’ultimo sia il punto più vicino a qualsiasi altro all’interno della regione. Questo succede dando a ogni attributo la stessa importanza, quindi la circonferenza che viene tracciata è normale e tutti i punti sono equidistanti dalla soluzione, se non è così la circonferenza è più schiacciata in base ai pesi degli attributi, di conseguenza i punti hanno distanze diverse da quello di riferimento. 

Gli attributi di un caso possono essere:

* attributi nominali o categorici: quelli che non hanno un ordinamento (semplici simboli)
* attributi lineari: posseggono un ordinamento quindi sono mappabili con dei numeri discreti o continui, essi sono rispettivamente mappabili su numeri naturali, quindi sono valori numerici e simbolici, e su quelli reali.

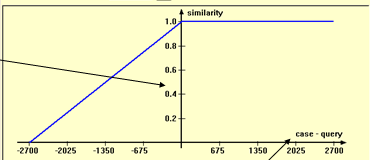
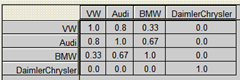
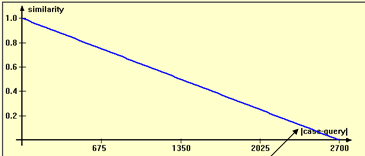
Data la grande differenza tra questi attributi, la distanza tra due casi si calcola attraverso HEOM (Heterogeneous Euclidean Distance Metric), una distanza di overlap in cui si definiscono le varie sottodistanze in base agli attributi.

Esempio:



Poco precisa per features nominali (o d=0 o d=1)

rangef=val max – val min

Per ottenere valori nominali più precisi, è possibile sostituire l’insieme di regole con una matrice di distanza, essa può essere utilizzata anche con le similarità dato che una è complementare all’altra. Nel caso in cui gli attributi siano nominali, si costruisce una matrice di distanza in cui sulla diagonale ho tutti i valori a 1 (come in figura), questo perchè si ha un incrocio tra due attributi identici, mentre sopra e sotto la diagonale sono simmetrici e hanno valori compresi tra 0 e 1. Se non c’è simmetria, significa che degli attributi hanno più importanza di altri. Se gli attributi sono lineari, è possibile costruire un grafico in cui sulle ascisse si mette la distanza tra i due casi che vengono analizzati,mentre sulle ordinate si mette la similarità. Nel caso in cui ci sia simmetria, la distanza tra il caso memorizzato e quello della query viene calcolata in valore assoluto. Serve quindi una funzione che per una distanza uguale a 0, dia similarità uguale a 1, mentre per distanze sempre più grandi, la similarità tende a 0. Nel caso asimmetrico (come nella figura a fianco) invece conta se la distanza è positiva o negativa, nel primo caso significa che nel caso che ho in memoria ho un valore più alto e quindi gli viene assegnata similarità 1, nel caso in cui è minore o uguale a zero significa che in memoria ho un caso con un valore inferiore quindi la funzione gli attribuisce un valore tendente a 0 man mano che la distanze tende a meno infinito. Come già detto in precedenza, la similarità si può descrivere col concetto di distanza, per farlo si calcolano le distanze tra coppie di attributi e da essi si calcola la media pesata, ogni attributo ha infatti un peso che indica la sua importanza. Infine, per ottenere la similarità, si normalizza a 1 la media trovata e si calcola il complementare. Ovviamente prima di effettuare il calcolo bisogna definire gli attributi e il loro metodo di calcolo, per quelli lineari si utilizza la normalizzazione a 1 mentre per quelli nominali si utilizzano tabelle di distanza oppure overlap.

Una generalizzazione della metrica locale HEOM è HVDM (Distanza di valore eterogeneo), essa permette di essere più precisi quando un attributo è nominale ed è utilizzata soprattutto nei metodi di classificazione.

La diversità rispetto a HEOM sta nel fatto che, per calcolare la distanza, si utilizza la probabilità che un caso appartenga a una classe sapendo che un attributo ha un dato valore. La probabilità si può stimare così:

dove, sapendo che l’attributo f ha un valore x Nx,i è il numero casi nella classe Ci e Nx è il numero totale di casi.

Esempio: 2000 maschi su 6000 hanno i capelli chiari, lo stesso colore lo hanno anche 5000 femmine su 10000, si può quindi dire che:

* P(M|f=chiari)=2000/(2000+5000)=2/7
* P(F|f=chiari)=5000/(2000+5000)=5/7
* P(M|f=scuri)=4000/(4000+5000)=4/9
* P(F|f=scuri)=5000/(4000+5000)=4/9

La distanza tra chi ha i capelli chiari e chi scuri sarà D(chiari,scuri)=, La similarità è quindi S=1-0,225=0,775

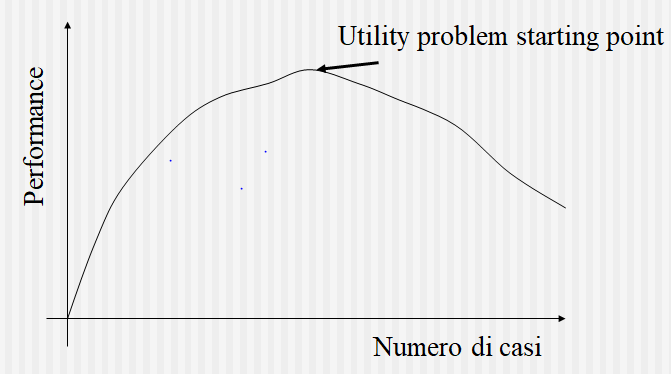
## Implementazione di k-nn

L’implementazione di k-NN può essere fatta attraverso una struttura kd-trees che permette di interrogare in tempo logaritmico rispetto alla dimensione della case memory (numero di casi): O(log N) (per k=1). Per questa struttura necessita di costruzione (O(n log n)) e mantenimento (O(log n)) della struttura kd-tree, tuttavia viene ricostruito ogni volta, il suo compito è recuperare i k nodi più vicini. Il kd-tree è un albero in cui un dato caso viene solamente confrontato con quelli più simili, vi sono infatti condizioni di split su uno degli attributi, in questo modo è possibile dividere l’insieme dei casi in due parti e quindi considerare solo una di esse. L’idea è utilizzare un heap dove ogni punto ha una sottostima della distanza dalla query come chiave, a ogni passo si tiene il miglior nodo trovato, ovvero il più simile. A ogni nodo in cui si verifica la condizione rispetto alla query, si inserisce il figlio nell’heap mettendo 0 come valore, il figlio non verificato viene sempre aggiunto ma con valore pari alla differenza tra la soglia della condizione e quello della query. L’algoritmo termina quando l’heap si svuota o il migliore dell’heap è peggio del best.

### Adattamento

Nel CBR può capitare che non vi sia il Retrieval, in tal caso si effettuano adattamenti a delle soluzioni già presenti, per farlo si utilizza uno dei seguenti metodi:

* Transformational adaptation: si riutilizza una soluzione per adattarla al problema cambiando una serie di parametri oppure modificando alcuni passi;
* Derivational adaptation: si riutilizza il metodo per trovare la soluzione, molto utile nei problemi di pianificazione.

L’ultimo passo delle 4 R è il passo di Retain, esso è il vero passo di apprendimento in cui si deve decidere come e quando memorizzare un caso, anche dimenticandone altri per fare spazio, utilizzando apposite funzioni di valutazione sui casi. In questi sistemi viene fuori il problema dell’utilità o swamping problem in cui la performance del sistema degrada dopo un certo limite di informazione in memoria, esse dipendono da:

* tempo di esecuzione;
* qualità delle soluzioni;
* competenza, ovvero la capacità di fornire soluzioni.

Queste dipendenze certamente non vanno nella stessa direzione quindi si creano questi problemi: Più casi ci sono in memoria, più si ha competenza e una qualità più alta, tuttavia si avranno anche tempi di risposta maggiori, allo stesso modo meno casi si hanno, più i tempi saranno migliori al costo di qualità e competenza na avrò un peggior tempo di risposta. Il learning serve quindi quando non si ha una soluzione oppure quando l’adattamento è molto consistente. Nel CBR vi è la cosiddetta maledizione della dimensionalità, ovvero un problema di rilevazione dei vicini all’aumentare degli attributi di un caso, questo perchè oltre a essi aumenta anche lo spazio occupato.

Il CBR andrebbe usato quando:

* esiste un gran volume di dati storici;
* gli esperti parlano del dominio “per esempi”;
* l’esperienza vale quanto la conoscenza di base;
* i problemi non sono formalizzati in modo preciso;
* esistono molte eccezioni alle regole di base;

CBR tipicamente offre una soluzione a minor costo al cosiddetto “knowledge acquisition bottleneck”. I sistemi CBR possono apprendere con l'esperienza e quindi sono self-maintaining. Rule-based systems sono migliori quando e’ difficile raccogliere i dati sui casi.

## Ragionamento incerto

Il ragionamento incerto è importante perché nella maggior parte dei casi non si ha a disposizione tutta la conoscenza su un dato problema, l’incertezza infatti nasce dalla mancanza di informazione e può compromettere le soluzioni dato che non si sa se siano vere o meno, e per calcolarle si utilizza la probabilità. Il ragionamento solitamente utilizzato è quello per causa-effetto, tante cause infatti possono produrre gli stessi effetti, spetta al sistema scegliere quelle più probabili. Il calcolo della probabilità è un approccio di tipo normativo, esso permette di ricavare soluzioni migliori rispetto ai meccanismi ad-hoc, questo è stato dimostrato da Cox attraverso sette regole:

* Chiarezza: le proposizioni devono essere ben definite e quindi solo vere o solo false;
* Rappresentazione scalare: un singolo numero reale è sufficiente per rappresentare la certezza;
* Completezza: una certezza può essere assegnata a una proposizione ben definita;
* Dipendenza dal contesto: una certezza dipende da altre certezze;

Belief(X|Y)

* Condizioni ipotetiche: la certezza che due variabili siano entrambe soddisfatte è data dalla probabilità condizionata;

Belief(X and Y)=f[Belief(X|Y),Belief(Y)]

* Complementarità: la certezza del complemento di una proposizione è una funzione di certezza nella proposizione stessa;

Belief(-X)=f[Belief(X)]

* Consistenza: c’è la stessa certezza nelle proposizioni logicamente equivalenti;

## Variabili casuali

Nell’intelligenza artificiale vengono utilizzate le variabili casuali, ovvero eventi che assumono un dato valore con probabilità.

Esempio: date due v.a. A e B, lo spazio Ω è dato dal prodotto cartesiano e ogni coppia (ai,bj) ha probabilità, esse sono densità congiunte e quindi è possibile calcolare la probabilità di ogni variabile effettuando la marginalizzazione.

Per ottenere un modello probabilistico coerente, bisogna avere la probabilità congiunta delle variabili, in quanto con essa si può ottenere ogni combinazione di variabili. Assegnare una probabilità a tutte le variabili è proibitivo, infatti si ha bisogno di valori con n equivalente al numero di variabili, tuttavia è possibile diminuire questo numero utilizzando le reti bayesiane. Per il calcolo della probabilità è possibile seguire due approcci:

* frequentista: si utilizza la statistica classica e la probabilità è legata alla frequenza di un evento, esso va bene solo se ripetibile:

dove è il numero di volte in cui E avviene su n ripetizioni;

* soggettivista: si utilizza la statistica bayesiana con probabilità soggettive, essa dà un grado di certezza a un evento basandosi sulla scommessa, ovvero la quantità di utile che si è disposti a scambiare per 1 se avviene e 0 nel caso non sia così, essa è coerente quando il bilancio è 0.

## Teorema di Bayes

Il teorema di Bayes indica che, dalla definizione di probabilità condizionata, dato n eventi h mutuamente esclusivi ed esaustivi e un qualsiasi evento e:

L’agente valuta tutte le evidenze e ne calcola la probabilità quando ne arriva una nuova (in questo caso e. P(hi) è la probabilità di hi prima di vedere e, P(e|hi) è la verosimiglianza, ovvero la probabilità in cui l’ipotesi è vera, P(hi|e) è la probabilità di hi dopo aver visto e (quindi quella a posteriori). La sommatoria di tutte le probabilità permette la normalizzazione a 1.

Esempio: un paziente può avere l’influenza (h1) o la malaria (h2) e non può stare bene. Se il paziente ha la febbre che probabilità ci sono che abbia le altre malattie? Dato che vi è poca informazione, si assegnano equamente le probabilità, quindi:

P(h1)=P(h2)=0,5, P(e|h1)=0,8 e P(e|h2)=0,9.

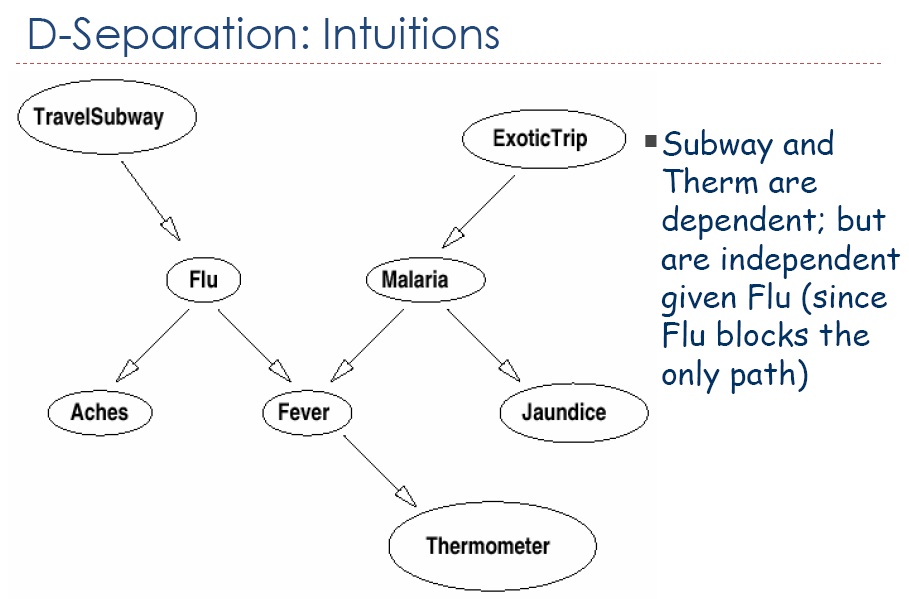
Ora si passa al calcolo delle probabilità a posteriori:

P(h1|e)=0,5\*0,8/(0,5\*0,8+0,5\*0,9)=0,4/0,85=0,47

P(h2|e)=0,5\*0,9/(0,5\*0,8+0,5\*0,9)=0,45/0,85=0,53

Bayes è utilizzato per evitare soluzioni errate e forzate. Quando si verificano più eventi in contemporanea, Bayes normale non funziona più, quindi si utilizza una versione generalizzata:

=

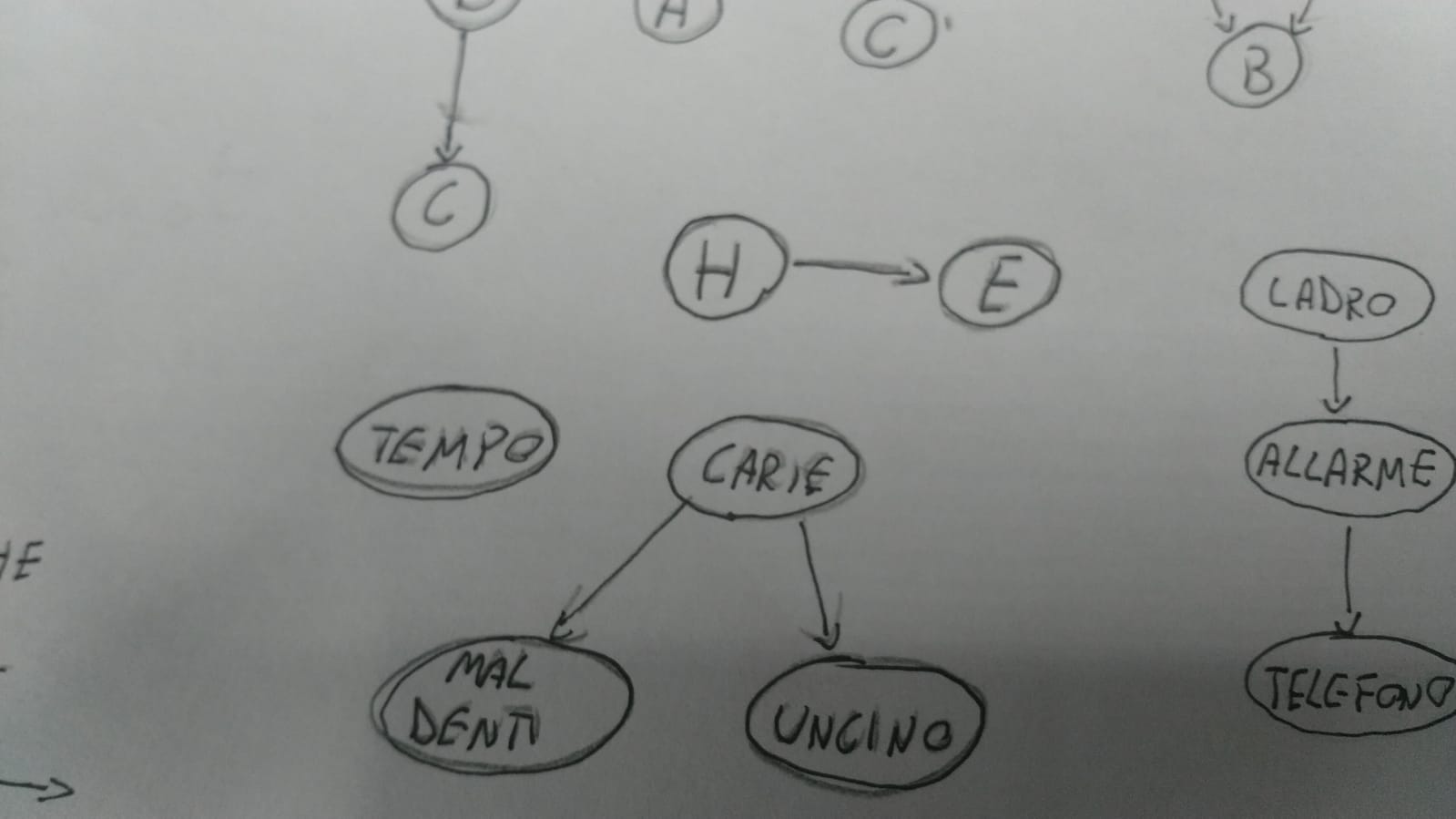
Riprendendo l’esempio precedente, in caso si verifichino più eventi, bisogna calcolare le probabilità di tutte le possibili combinazioni, per ridurle si utilizza la definizione di indipendenza: in questo modo si hanno solo n probabilità al posto di . 

## Reti Bayesiane (RB)

Una rete bayesiana è un metodo che permette di interrogare delle variabili probabilistiche data l’evidenza, essa si rappresenta utilizzando i grafi DAG, i nodi infatti rappresentano le variabili mentre gli archi le loro dipendenze:

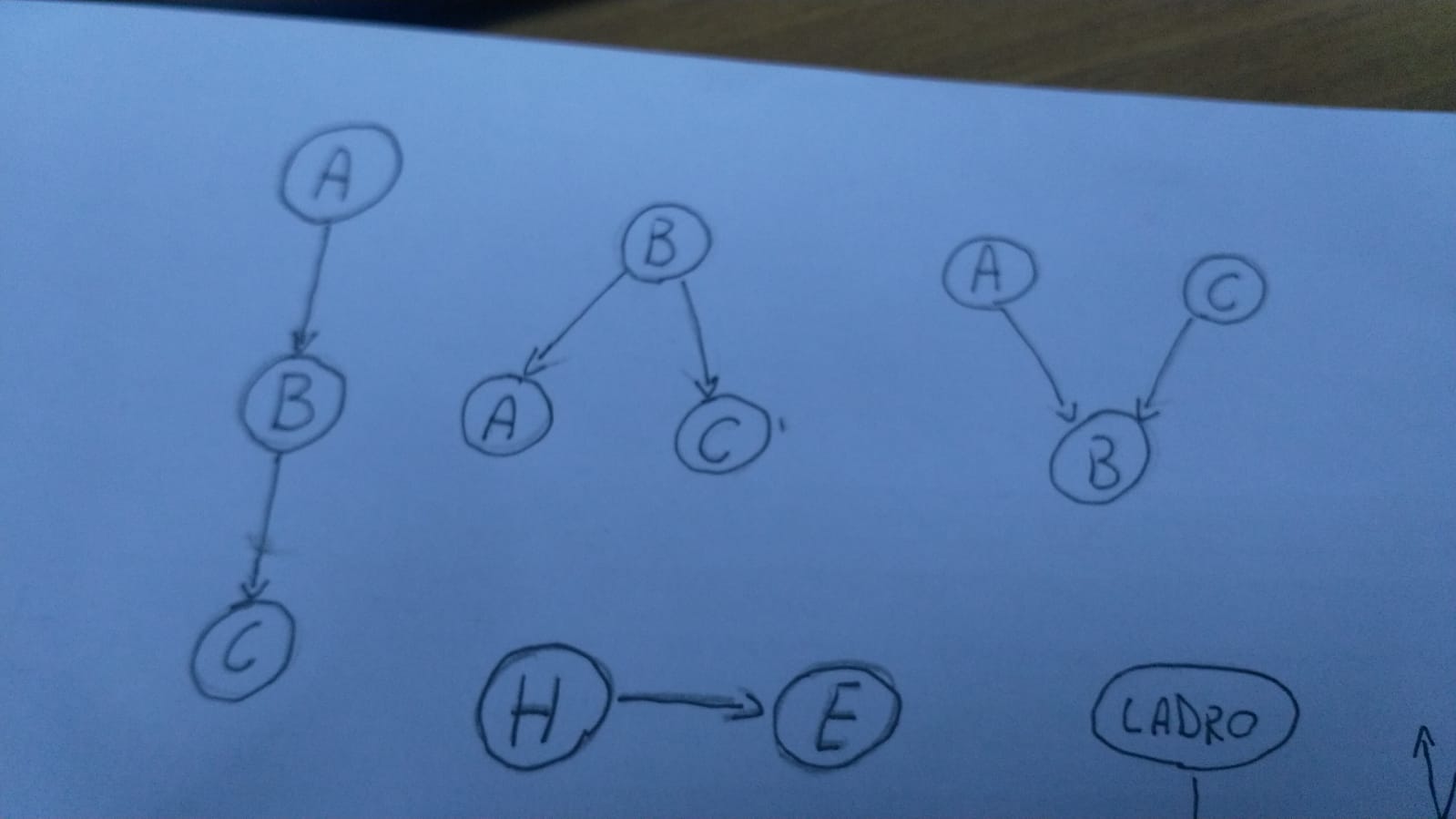
Una rete bayesiana è una coppia <G,P> dove G è un grafo e P è una distribuzione di probabilità sui nodi del grafo stesso, infatti secondo il teorema di fattorizzazione:

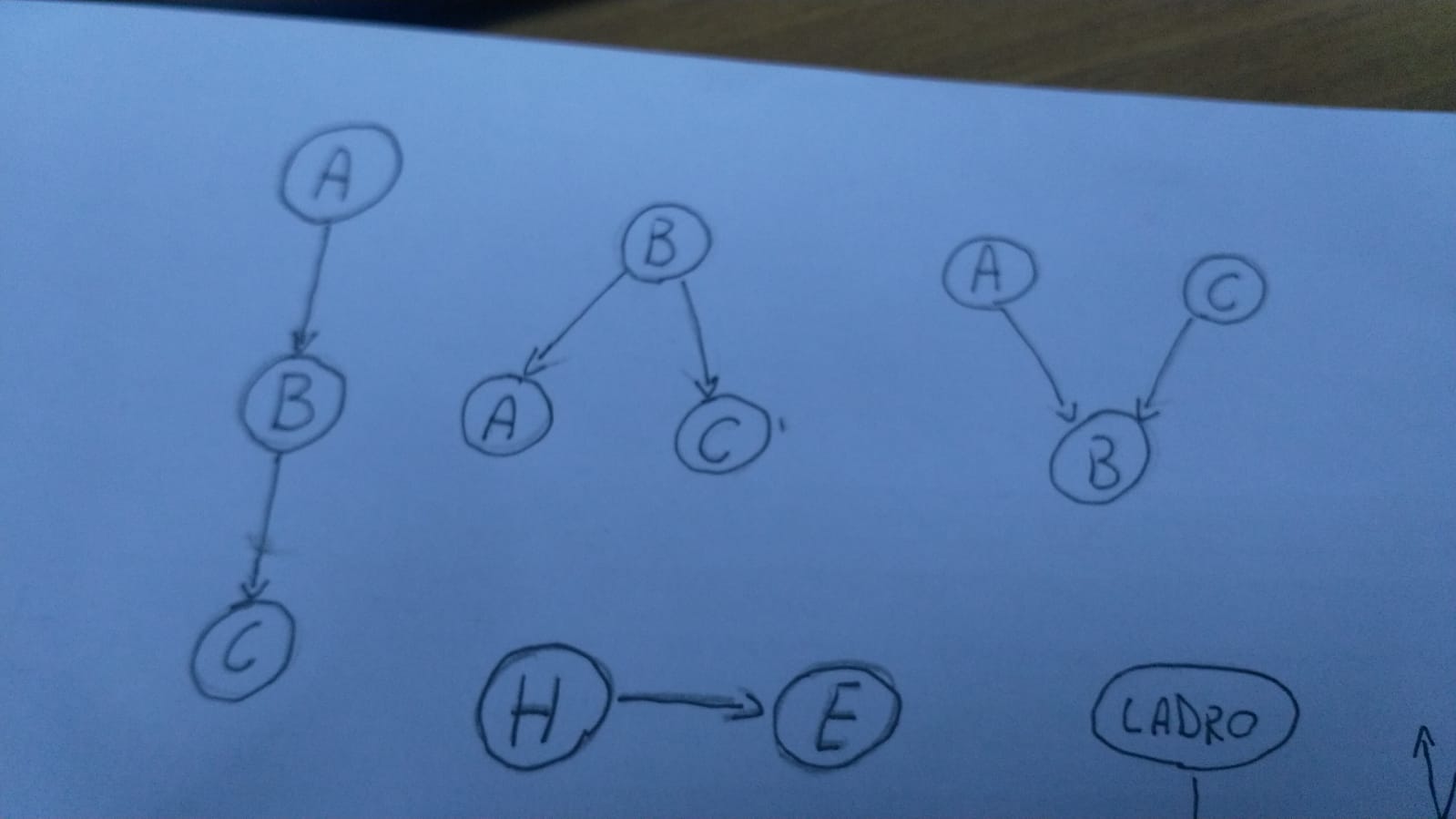
La probabilità congiunta dei nodi è il prodotto di tante condizionate del nodo stesso dato il genitore, questo è importante perché permette di ottenerla lavorando sui singoli nodi e i relativi genitori.

Esempio: un paziente ha una carie quando ha il mal di denti oppure quando l’uncino si blocca, esse non vengono condizionate dal tempo atmosferico. Nel caso venga il mal di denti quando nevica, allora si aggiunge una dipendenza tra tempo e maldidenti.

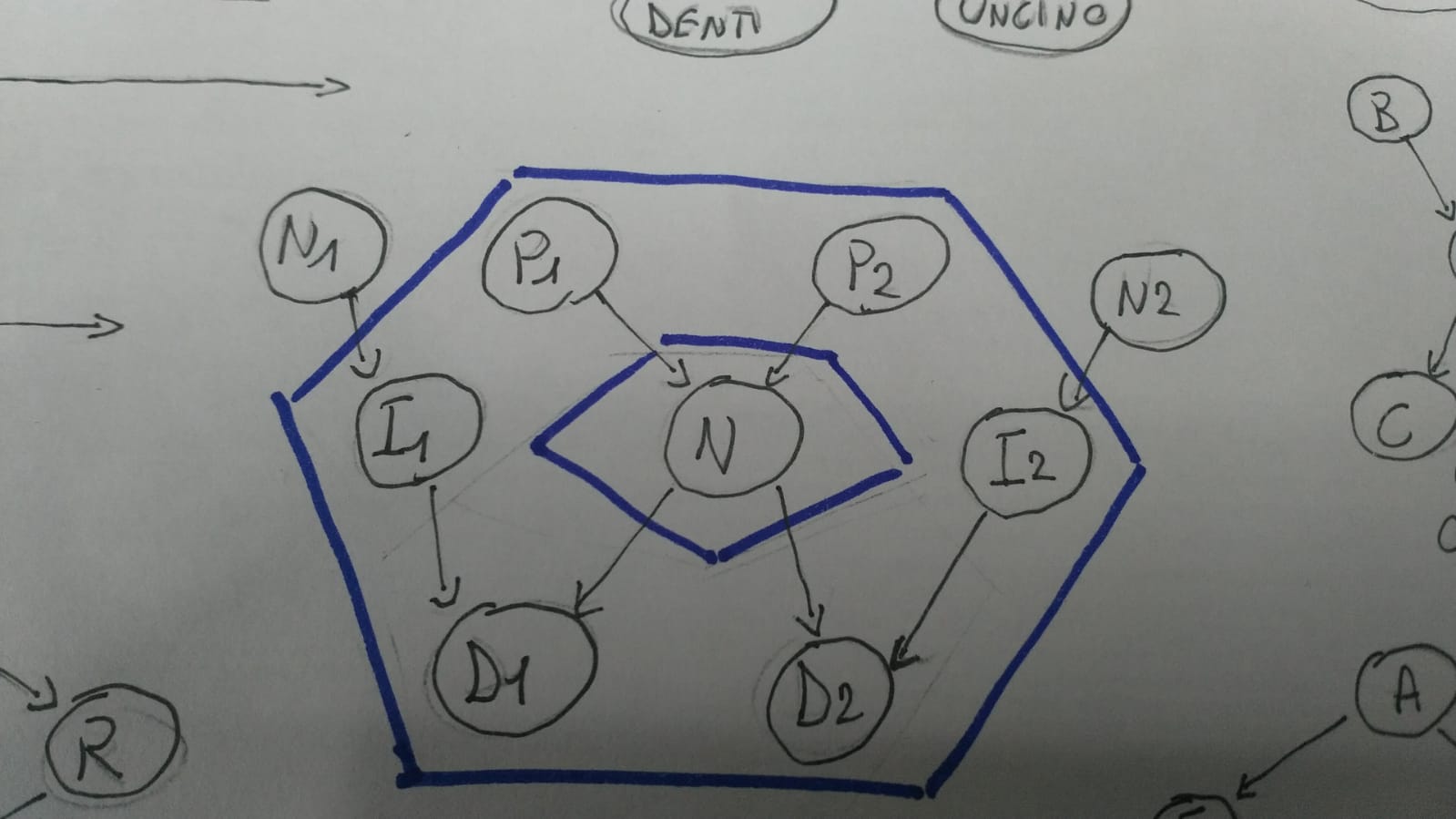
#### Indipendenze condizionali

Nelle RB vi sono casi in cui, date delle dipendenze, vi possono essere informazioni implicite, nel primo caso si ha B dipendente da A e C dipendente da B (quindi dipende anche da A), tuttavia nel caso in cui B sia noto allora A e C diventano indipendenti tra loro dato che la verifica di A non influenza in alcun modo C, questo modello è detto Bayes lineare.

Esempio: se entra un ladro in casa, l’allarme suona e manda un messaggio. Sapendo che l’allarme è suonato, il messaggio viene mandato ma non è detto che la causa sia il ladro, quindi non mi serve altra informazione dato che il risultato non cambia.

Un secondo caso è quello in cui A e C dipendono entrambi da B, in questo caso A e C dipendono l’uno dall’altro: se si ha A, aumenta la probabilità di B e di conseguenza anche di C, se si ha B invece i due eventi sono indipendenti, questo modello è detto Bayes divergente.

L’ultimo caso è quello in cui B dipende sia da A che da C, in tal caso A e C sono indipendenti, tuttavia diventano dipendenti se si sa il valore di B, questo modello è detto bayes convergente.

Esempio: L’allarme può suonare a causa di un ladro o di una mosca, se si sa che in casa è entrato un ladro, la probabilità della mosca non cambia. Se invece l’allarme è suonato e che era una mosca, la probabilità del ladro scende.

Dato un grafo DAG, due vertici u e v ed un sottoinsieme E di nodi, u è d-separato da v dato E se ogni cammino indiretto da u a v viene bloccato da E, ovvero che:

* esiste un nodo z lineare o divergente in E;
* esiste un nodo z tale che nè sè stesso nè i discendenti appartengono a E

Le probabilità delle reti bayesiane vengono date ai nodi rispetto ai genitori, ognuno ha infatti una tabella in cui si segnato tutte le probabilità per ogni possibile combinazione (più genitori ci sono, più celle ci saranno). In una RB, se si conoscono i genitori di un dato nodo, tutti i non discendenti sono irrilevanti, di conseguenza quel nodo è influenzato solamente dai genitori e dai discendenti, essi insieme agli altri genitori dei figli formano la coperta di Markov, una coperta che indica le dipendenze di un dato nodo, permettendo quindi di trovarne la probabilità. Una RB permette diversi tipi di ragionamenti:

* Predittivo: si ragiona seguendo le direzioni delle frecce osservando le cause e prevedendo gli effetti;
* Diagnostico: dati gli effetti si trovano le relative cause;
* Intercausale: si considera solo una causa e si buttano via le altre;
* è possibile effettuare molteplici ragionamenti dal momento che sono un formalismo isotropico.

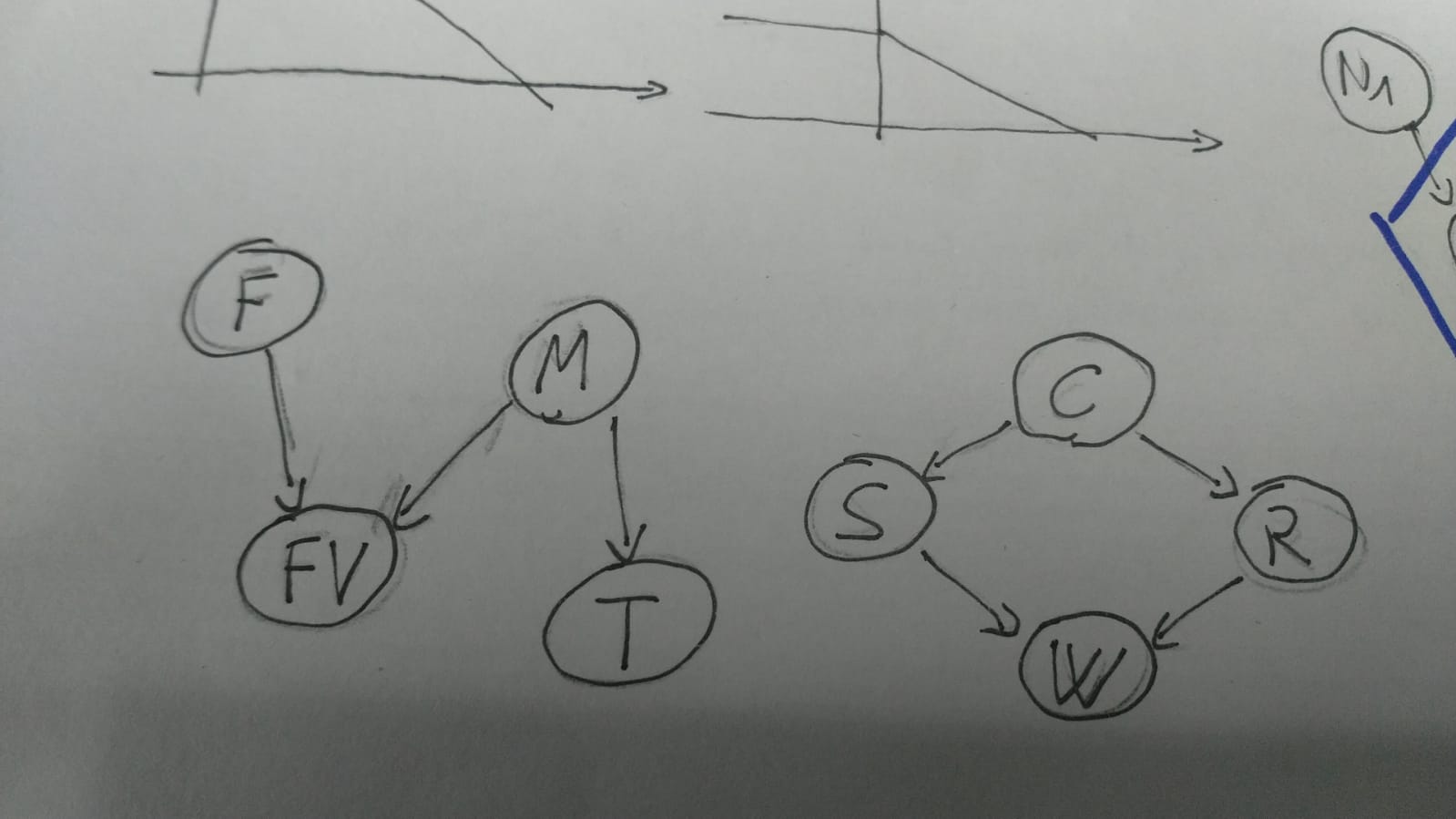
#### Noisy OR e Noisy Max

Esempio: data una RB fatta come in figura, vi sono 3 possibili cause per FV e quindi 8 valori, per evitare ciò si suppone che le cause siano indipendenti e se ne prende una, M ad esempio: per causare FV potrei anche non avere C e F, la prima può essere però inibita: P(-FV|...) è la probabilità di inibizione, ovvero che, data una causa, essa non permette l’effetto desiderato:

P(-FV|M,C,F)=1-P(M)\*P(C)\*P(F)

La probabilità di inibizione è quindi il prodotto delle probabilità contrassegnate come vere, se nessuna di esse lo è, allora non c’è effetto.

Col noisy OR è quindi possibile rendere lineare il numero di probabilità, il noisy max è leggermente differente: di tutte le probabilità si trova quella più alta e si utilizza per tutti i calcoli del caso, essa è utile quando un nodo non è binario, ovvero che ha molteplici valori invece di vero e falso.

Esempio: si considera la seguente RB

FV in questo caso può avere tre valori: forte, leggera o assente, dal momento che il nodo non è più binario, lo si deve gestire con dovuti accertamenti: prima di tutto bisogna ordinare i valori in modo decrescente e si trova quello con probabilità massima.

Oltre alle probabilità presente, ce n‘è una in più, ovvero il LEAK: la probabilità di tutte le cause non presenti nel modello.

## Algoritmi di inferenza

Gli algoritmi di inferenza servono a rispondere a una query probabilistica in una rete bayesiana, effettuando il calcolo delle probabilità a posteriori date le evidenze: dato l’insieme delle variabili query Q e le evidenze E, si deve calcolare P(Q=q|E=e), essa è calcolabile come il rapporto tra la probabilità di Q ed E diviso quella di quest’ultimo:

P(q|e)=P(q,e)/P(e)

Per calcolare P(q,e) basta marginalizzare tutte le v.a che non fanno parte di Q ed E:

Dato l’insieme Xp,...,Xn corrispondente a tutte le v.a. della rete tranne quelle di Q ed E,

gli algoritmi possono essere:

* Esatti: forniscono un valore preciso;
* Approssimati: forniscono un valore approssimato dato che sono molto efficienti anche se convergono al risultato esatto.

La complessità di questi algoritmi dipende dalla struttura della RB:

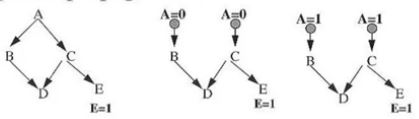
* con una struttura singly connectly infatti non sono presenti loop indiretti, quindi per ogni nodo corrisponde a un cammino che permette di raggiungerlo, permettendo una risoluzione in tempo polinomiale;
* con una struttura multiply connectly vi sono cicli indiretti, questo potrebbe causare la presenza degli algoritmi nell’insieme NP-Hard.

L’inferenza nelle RB di questo tipo è data dall'algoritmo di Kim-Pearl che risolve le query date in tempo polinomiale, esso funziona come uno scambio di messaggi tra i nodi. L’algoritmo è applicabile anche nelle rete multiply-connected, per farlo la si converte in singly-connected (attraverso il clustering o il conditioning) e si applica su esso, tuttavia il passo inverso può essere esponenziale.

#### Clustering

L’idea alla base del clustering è raggruppare i nodi in macronodi in modo da rendere la rete singly-connected, in tal caso però bisogna cambiare la quantificazione nel macronodo creato: per ogni possibile combinazione dei nodi uniti, si effettuano le moltiplicazioni tra le probabilità, per calcolare le probabilità a posteriori però bisogna controllare la RB originale.

#### Conditioning

Il conditioning converte una RB multiply in singly istanziando una variabile: istanziando un nodo assegnandogli un valore è infatti possibile rompere il ciclo, come risultato si ottengono due RB singly-connected (una per ogni valore della variabile). Tutte le probabilità vengono calcolate utilizzando entrambe le RB e, una volta calcolate, si effettua la media delle stesse.

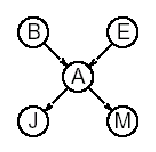
#### Eliminazione di variabili

L’eliminazione di variabili è un algoritmo che permette di togliere le variabili diverse dalla query e dall’evidenza, esse si autoeliminano perchè si marginalizzano con delle somme.

Esempio:

per eliminare s, si sommano le probabilità in cui è presente.

Ordinamenti diversi danni differenti risultati, quello ottimale è NP-Hard, tuttavia ci sono buone euristiche. Data la query e l’ordinamento delle distribuzioni, si mettono delle sommatorie nel seguente modo:

* A partire dal fondo, si prosegue verso sinistra;
* Una variabile viene eliminata quando non compare più a sinistra nell’ordinamento;
* Una variabile osservata è eliminata in automatico.

Esempio: data una RB, la query B|j,m e l’ordine “BEAJM”, si osservano le variabili J e M per togliere A ed E, le prime due sono già eliminate perchè osservate, per queste ultime invece:

f è una funzione che restituisce un vettore grande in cui sono presenti tutti i valori di probabilità di ogni caso considerato. Questi vettori devono essere moltiplicati tra loro in base ai valori booleani dati: nel caso di -a,b ed e si prendono A-J[-a], A-M[-a] e ABE[-abe] e si moltiplicano,quindi:

=

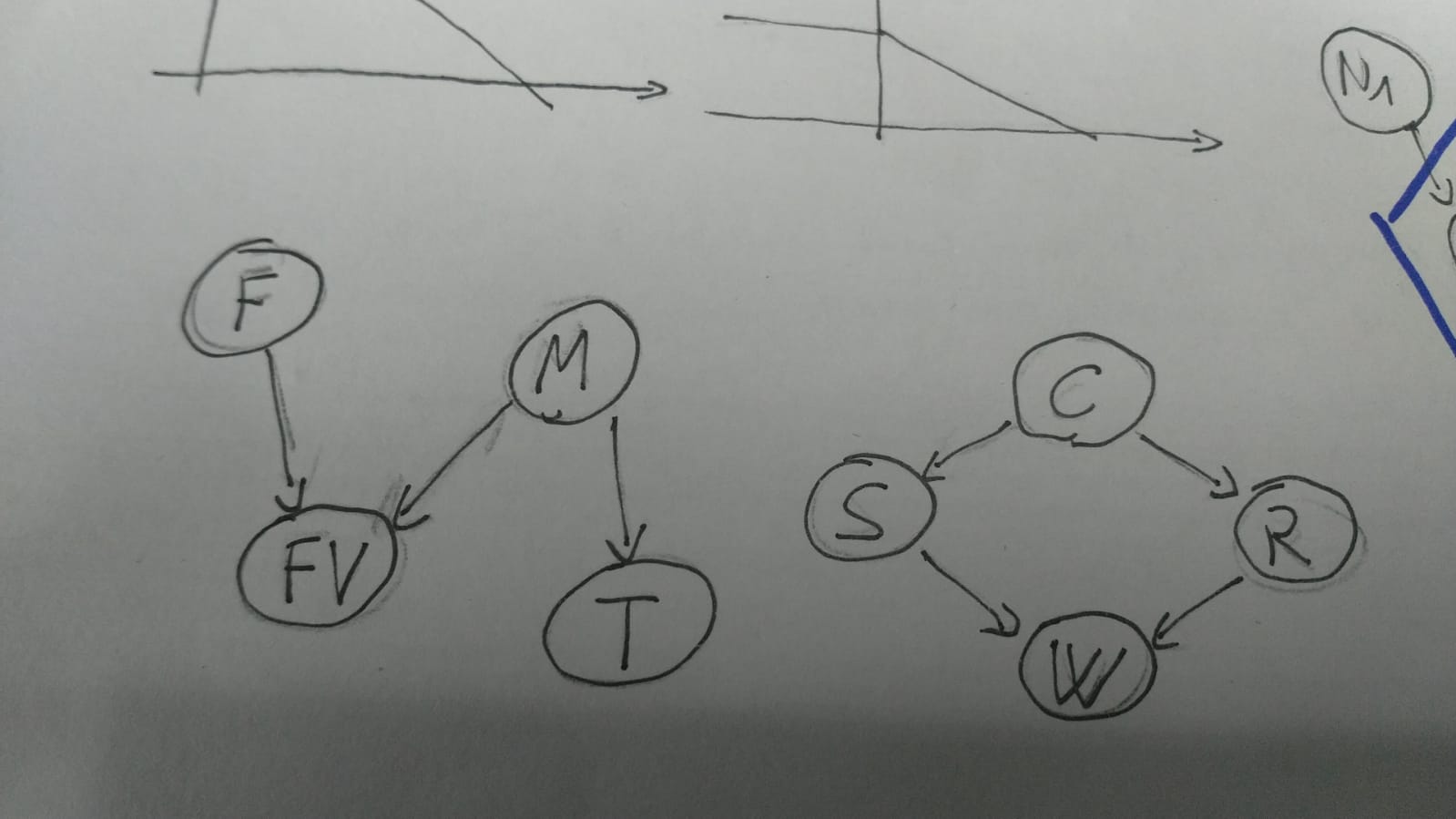
Le variabili irrilevanti sono quelle che non sono evidenze o antenati della query.

#### Inferenza stocastica

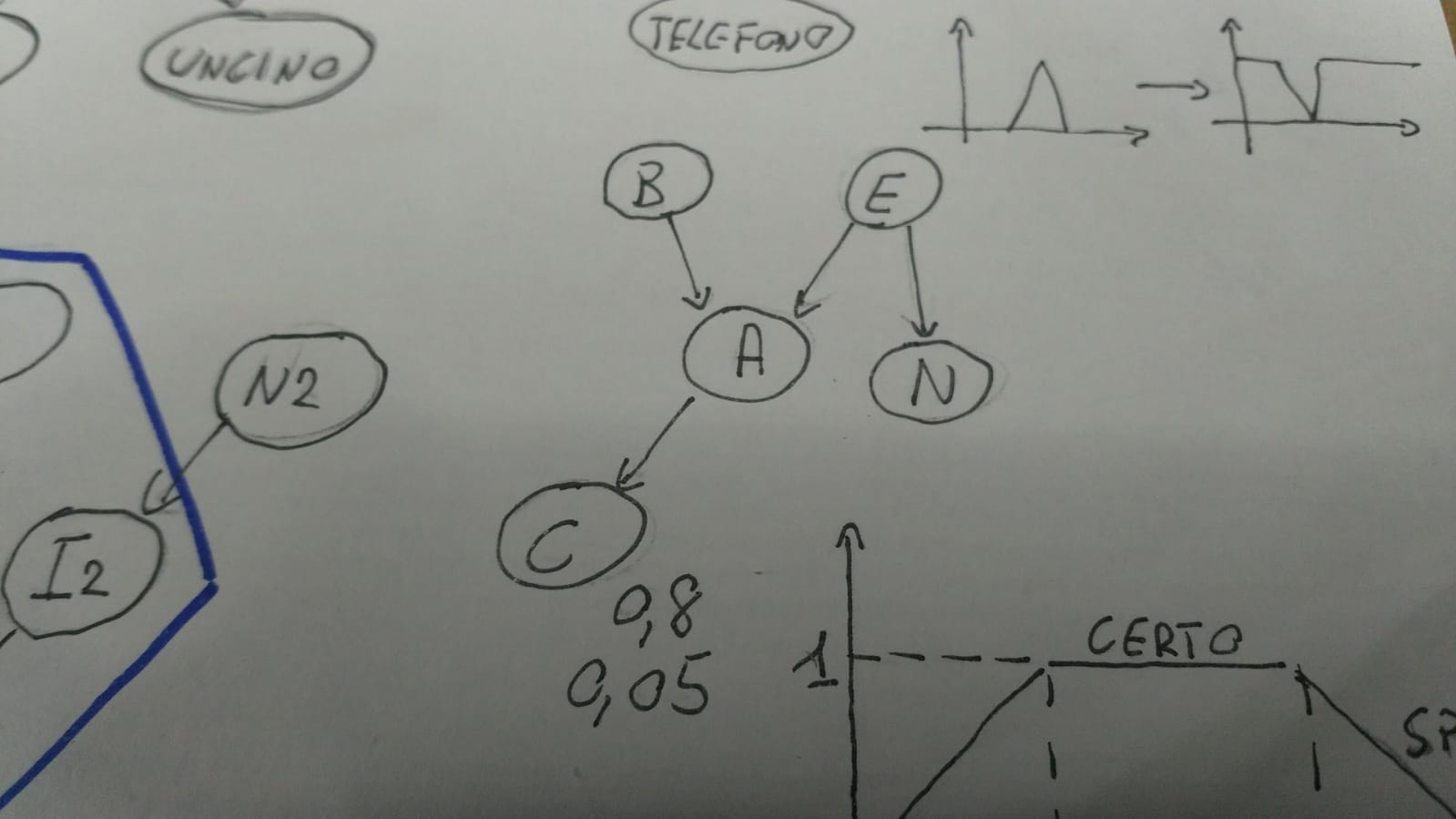
L’utilizzo di algoritmi stocastici potrebbe risultare utile per il calcolo delle probabilità, in essi più tempo si dà alla risposta e più è precisa, inoltre si basano sulla simulazione stocastica: i valori vengono campionati ogni tanto data la loro distribuzione, essa deve essere rispettata in media con n ripetizioni del campionamento. Dopo che tutte le variabili sono state campionate, vengono eseguiti n run di simulazione, più n è alto, più il risultato tende al valore corretto. Per campionare una variabile, l’idea è, data una variabile aleatoria X con k stati, dividere la distribuzione in k parti di ampiezza pi, si genera α casualmente e, se cade nell’i-esima parte, allora X=xi. Gli algoritmi si differenziano nel modo in cui campionano.

#### Rejection sampling

Il rejection sampling campiona le variabili partendo dalle radice e seguendo le frecce, dal momento che tutte dipendono dai genitori. Una volta campionate, se alcune v.a. sono nell’evidenza e il campione preso è diverso da essa, si ha un run inconsistente e quindi si butta via, la probabilità è data dal rapporto tra le run con successo diviso il numero totale:

Esempio: data una RB, si campiona C e risulta vero, quindi si campionano S e R utilizzando una distribuzione di probabilità vera. Si suppone che S sia falso e R vero, per campionare C bisogna quindi considerare quella data situazione. Se si osserva S come vero, la run è inconsistente, se invece si osserva R come vero, non lo è. Con 100 run di cui 20 che rispettano le condizioni date, la probabilità è di 20/100=⅕. Questo algoritmo non va bene quando si devono campionare valori di v.a. poco probabili, quindi con evidenze molto rare.

#### Likelihood weighting

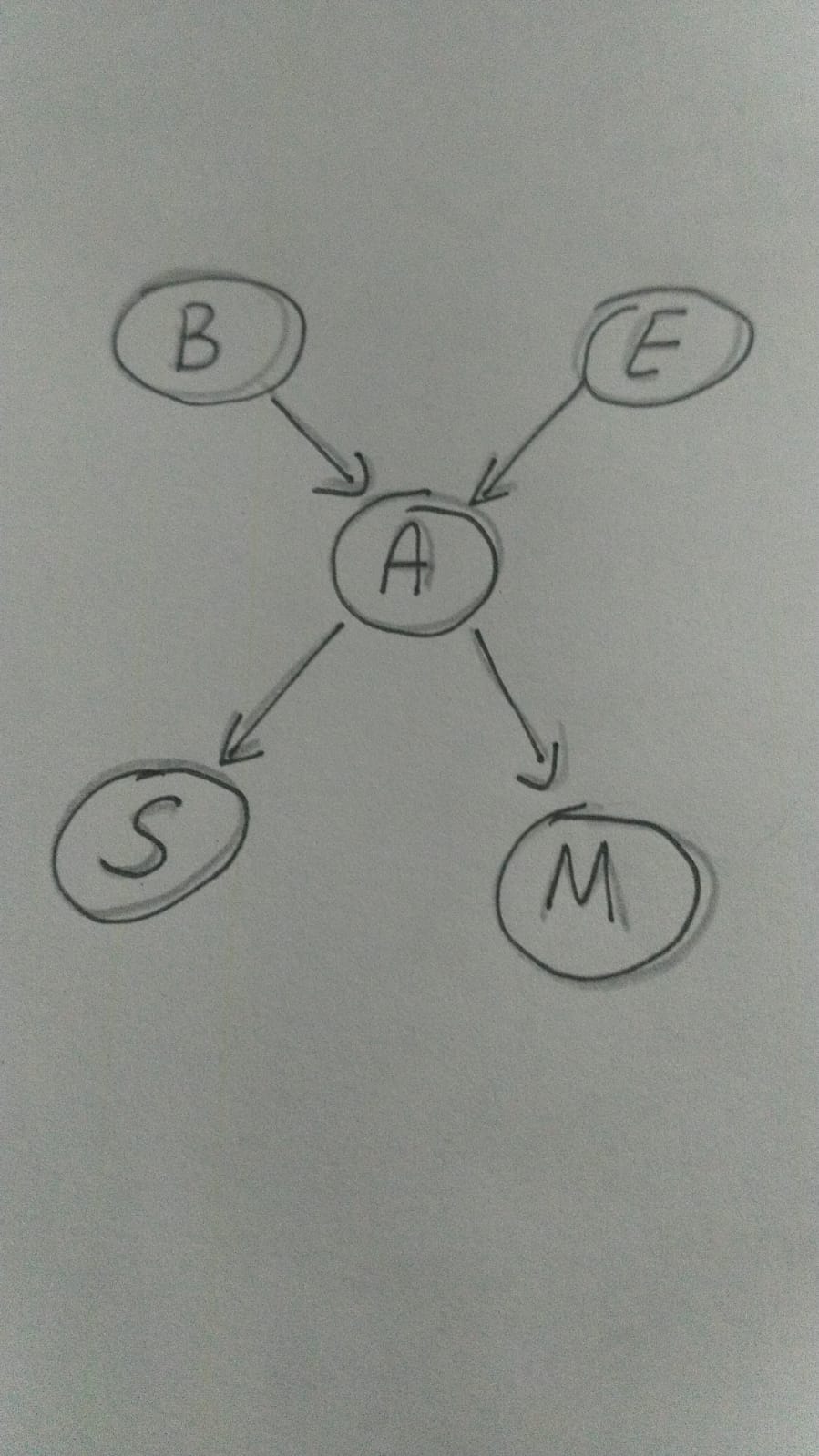
Il likelihood weighting permette di pesare i vari campioni con la differenza che non vengono campionati i valori osservati, per non imbrogliare infatti si può utilizzare la probabilità dell’evidenza come peso della run.

Esempio: Data una RB, si campionano B, E, A e C e tutte e quattro risultano vere, N invece risulta falsa dopo averla osservata. Osservando C come vero, la probabilità di B dato C è il rapporto tra un peso i e la somma di tutti i pesi, quindi il peso della run è la probabilità di C, esso dipende da come viene osservato A. La stima della probabilità è quindi una media ponderata tra tutti i pesi.

#### Markov sampling

Markov sampling utilizza la coperta di Markov per il calcolo delle probabilità.

Esempio: Data una RB, la coperta di B è formata da tutti i nodi tranne E, quella di C invece da tutti. Una v.a. coperta è campionabile e quindi si può calcolarne la probabilità:

L’inizializzazione delle v.a. viene effettuata casualmente, quelle osservate invece hanno la propria probabilità, in seguito si indica il numero di run. Se la v.a. è un evidenza, il campione è il valore osservato, altrimenti è la probabilità del valore condizionato dalla coperta.

L’algoritmo ha una fase di riscaldamento in cui vi sono un certo numero di run escluse dal conteggio. Per calcolare la probabilità della query data l’evidenza si fa:



Esempio:Data una RB, se la v.a. presa in considerazione è l’evidenza, allora il campione è il valore osservato, altrimenti è la probabilità del valore condizionato dalla coperta, quindi:

Si inizializzano tutte le variabili tranne E e si campiona B, la sua coperta è AE

P(B|AE)=P(B)\*P(A|BE)

P(b|-a-e)=P(b)\*P(-a|b-e)=0,01\*0,06=0,006

P(-b|-a-e)=P(-b)\*P(-b|-a-e)=0,99\*0,99=0,998

Campionando E, si ha che:

P(E|AB)=P(B)\*P(A|BE)

P(e|-a-b)=P(e)\*P(-a|-be)=0,142

P(-e|-a-b)=P(-e)\*P(-e|-b-e)=0,997

Dopo aver campionato tutte le variabili, si trova il numero di run e si calcola la probabilità:

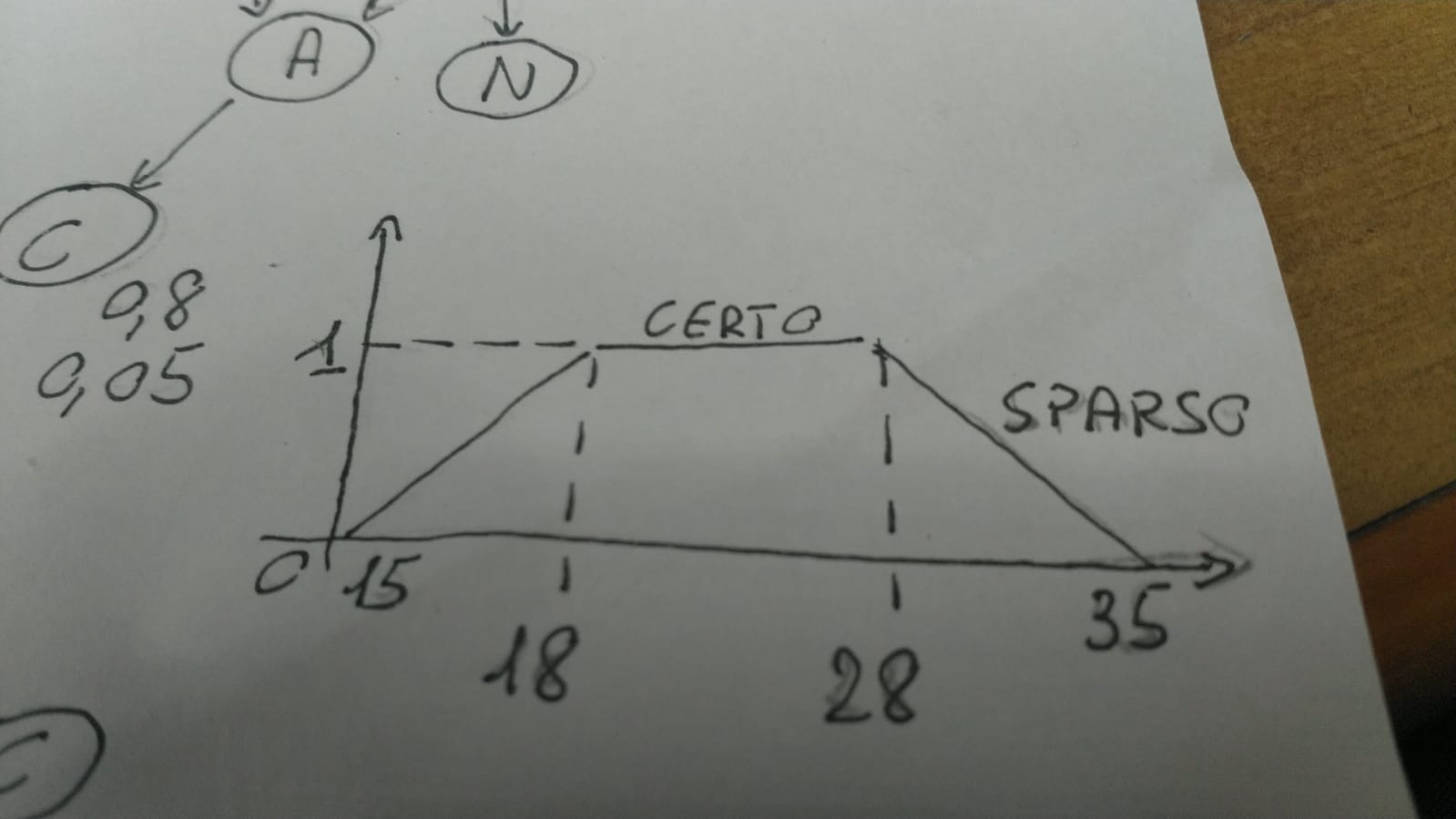
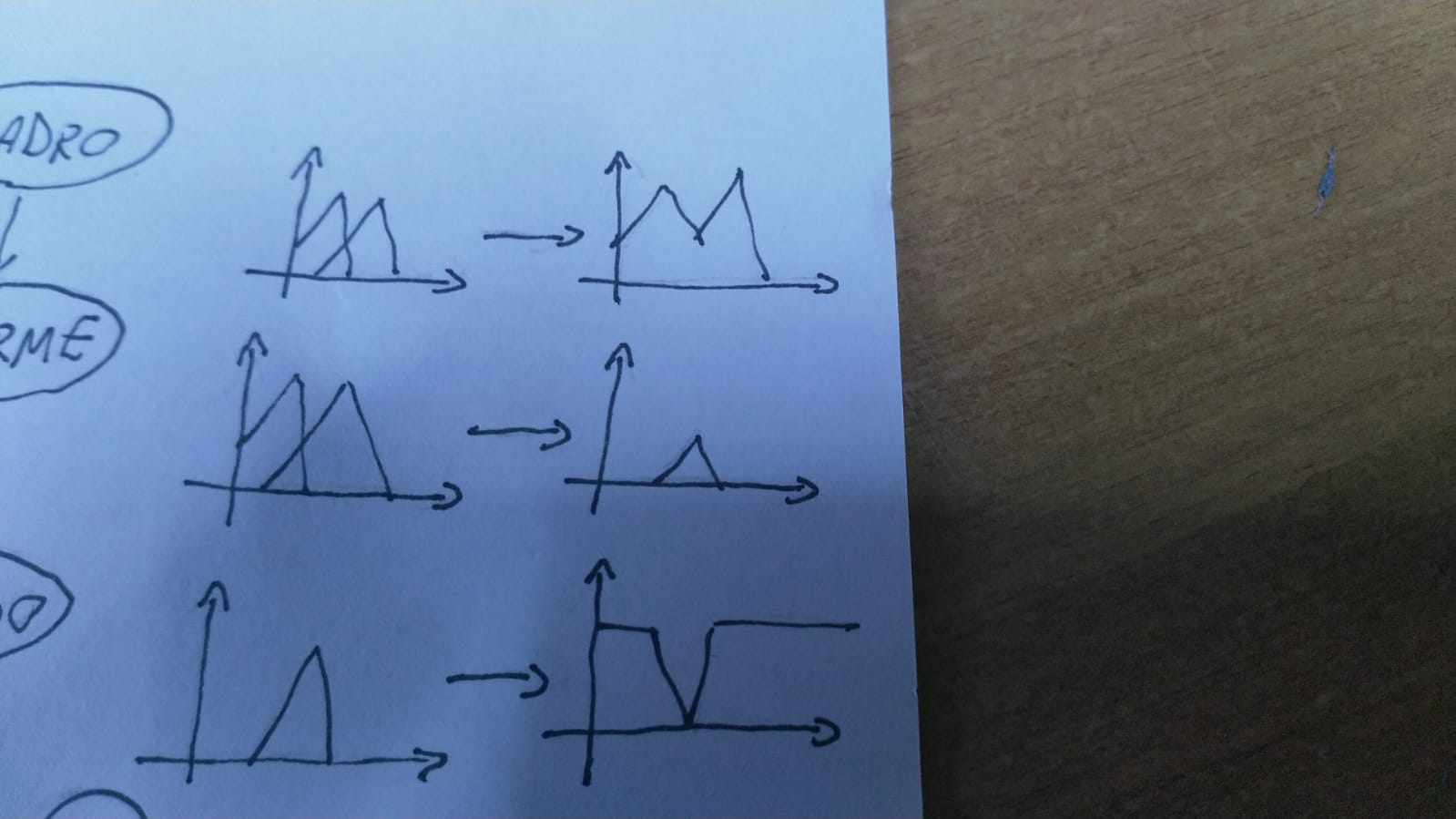
P(a)=¾ P(a-b)=¼

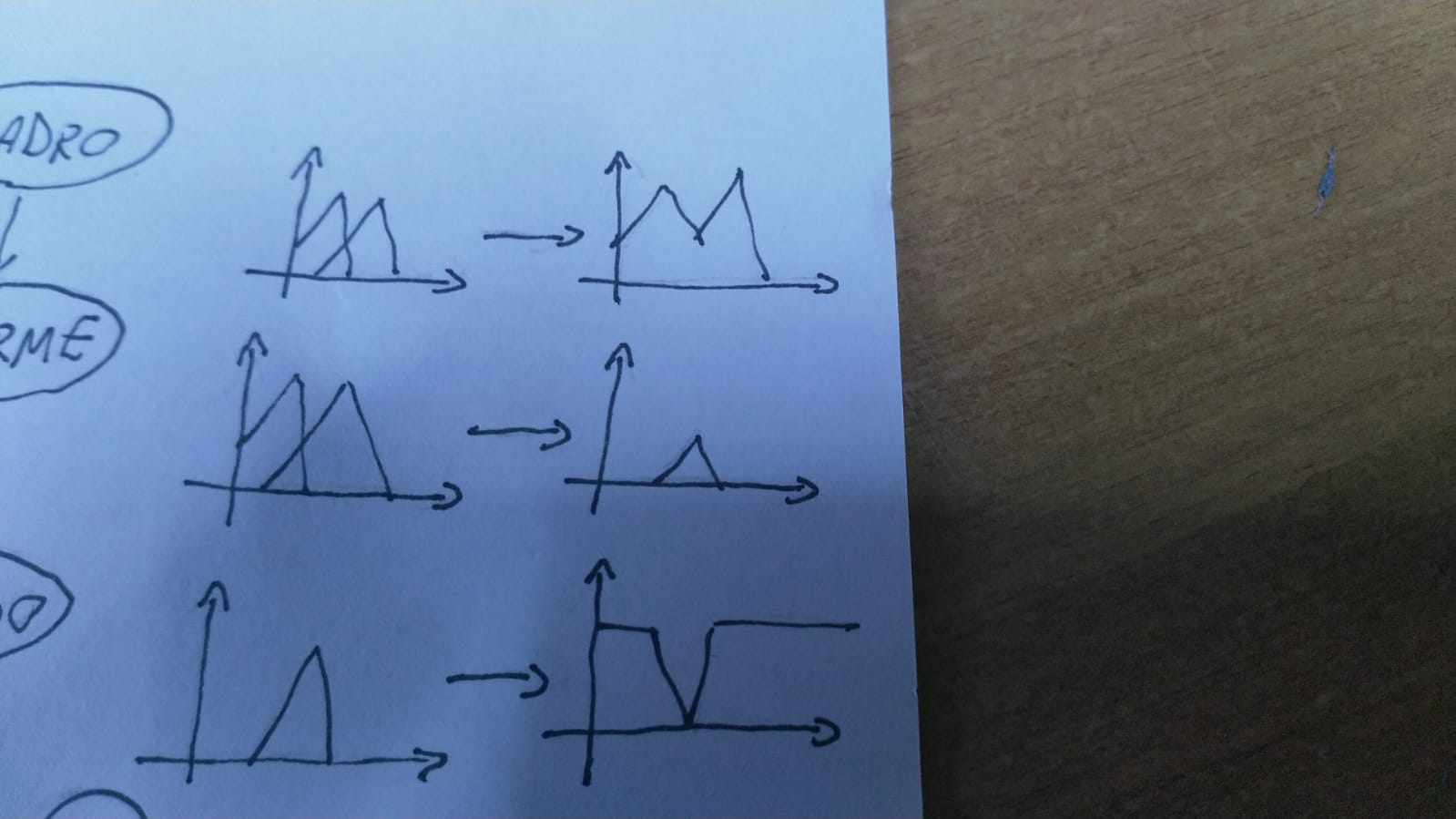
## Logica Fuzzy

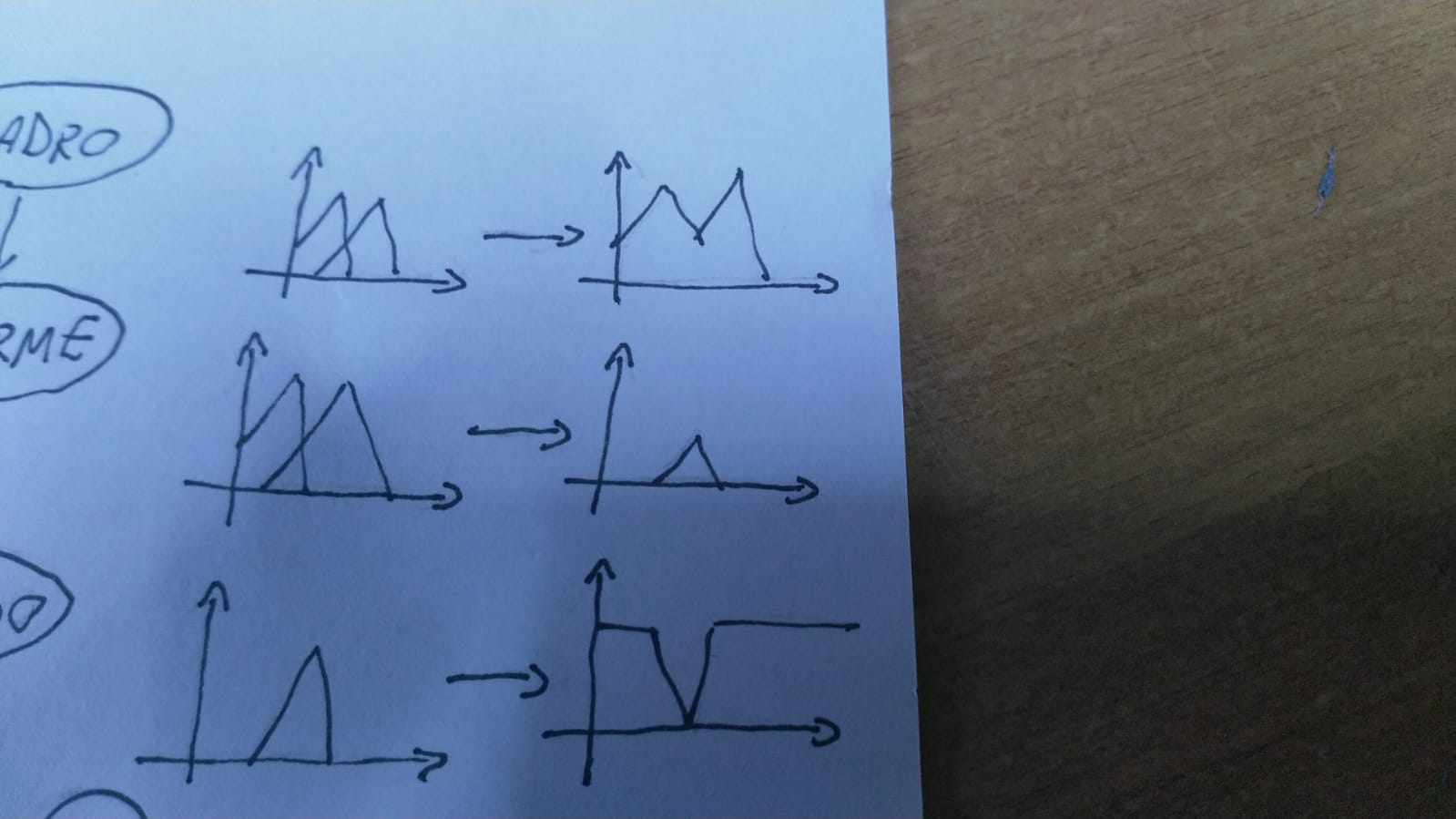
La logica fuzzy è un tipo di logica in cui l’incertezza è data da un’informazione approssimata, quindi i concetti non sono ben definiti e le verità sono parziali, infatti questi non possono essere rappresentati con valori booleani. Vaghezza e incertezza possono coesistere tra loro, l’informazione infatti può avvenire ma non si sa come.

Esempio: una persona che soffre di epatite presenta nel 60% dei casi una forte febbre, nel 45% con un colore giallastro e nel 30% con severa nausea.

In questo caso si ha sia incertezza che vaghezza in quanto i sintomi potrebbero non presentarsi e, in più bisogna definire delle regole. Nel caso della febbre, è possibile mettere un intervallo di temperature: da 40 a 42 la febbre si può considerare forte, altrimenti no, questo permette di convertire informazioni vaghe in booleani, tuttavia questi ultimi non sono un buon metodo per rappresentare la vaghezza, quindi si utilizzano numeri interi.

Esempio: la temperatura confortevole deve essere tra i 18 e i 28 gradi, da questa informazione si può dire che, finchè la temperatura è nel range, allora l’appartenenza è certa e quindi ha valore 1, altrimenti 0. Ragionando in questo modo, però, temperature come 29 o 17 risulterebbero non confortevoli anche se molto vicine all’intervallo, per risolvere ciò, agli estremi del range si scende lentamente, in questo modo la temperatura in questa discesa si può considerare confortevole ma non del tutto, la sua appartenenza è infatti detta sfumata e ha un valore compreso tra 0 e 1. Questi numeri non sono considerabili in alcun modo come probabilità. La distribuzione delle logica fuzzy si può rappresentare come una curva a campana. La logica fuzzy utilizza le cosiddette variabili linguistiche dal momento che si basano su come sono scritte, esse sono formate da variabili base e da un certo numero di termini linguistici (qui ci possono essere ambiguità), per ognuno di essi vi è una funzione di appartenenza (la cosiddetta membership). In caso di situazioni ambigue, gli intervalli sono incrociati, quindi vi sono diversi grafi in base al termine linguistico. Sulle membership si possono effettuare le seguenti operazioni: 

* Unione: è data dal massimo delle membership;
* Intersezione: è data dal minimo delle due membership;
* Complemento: complemento a 1 della membership data.

Una proposizione fuzzy è una proposizione in cui il valore di verità è compreso tra 0 e 1, su essa si definiscono le membership per ogni valore linguistico, con queste ultime è possibile utilizzare modificatori linguistici. Una regola fuzzy è una regola in cui, se vale una condizione fuzzy, allora vale la relativa proposizione.

Esempio: sistema per dare la mancia basandosi sulla qualità del servizio e del cibo seguendo tre regole:

* Servizio povero e cibo rancido: poca mancia;
* Buon servizio: mancia media;
* Servizio eccellente o cibo delizioso: molta mancia.

Prima di tutto si definiscono le membership in base ai termini linguistici, per rappresentarli si utilizzano valori interi nel seguente modo:

* Si calcolano le membership dati i numeri;
* Si osservano i booleani: AND è l’intersezione, OR è l’unione e, da essi, si uniscono.

La defuzzificazione è il valore medio della membership, essa equivale a una somma per le funzioni discrete e un integrale per quelle continue. Ogni funzione non lineare e continua è approssimabile a fuzzy con una precisione a piacere.

## Teoria delle decisioni

La teoria delle decisioni è un metodo che permette di catturare l’incertezza e di utilizzarla al fine di trovare la decisione migliore, essa è normativa dal momento che incertezza e utilità vengono pesate attraverso il calcolo delle probabilità.

Questo modo di prendere decisioni non è tuttavia descrittivo dal momento che gli esseri umani non seguono le regole della probabilità in modo preciso.

La teoria delle decisioni fornisce quindi una serie di regole che un agente artificiale deve seguire al fine di ottimizzare un percorso verso un dato obiettivo:

* Credenze e conoscenza incerta: l’agente opera in un ambiente incerto e ha delle credenze supportate da prove certe;
* Decisioni: In base a un dato obiettivo bisogna scegliere la migliore tra tante decisioni in base alla loro utilità, bisogna quindi valutare le possibili situazioni.

Dopo aver saputo cosa fa una certa probabilità, esse si utilizzano per ottenere la miglior soluzione massimizzando l’utilità e minimizzando i costi, se si effettua una scommessa, ad esempio, si preferisce la soluzione che permette più vincite. In questa teoria vi è il concetto di preferenza: date due soluzione A e B, A≻B indica che si preferisce la prima alla seconda, con A~B vanno bene entrambe e lo stesso vale per A≿B, in quest’ultima però si ha una preferenza verso A. La scelta delle decisioni avviene attraverso le lotterie, ovvero delle scelte probabilistiche a n opzioni, ognuna con una probabilità, nel caso di una lotteria avente due possibili scelte, essa si può rappresentare come L=[p,A;(1-p),B], cioè che la scelta A ha una probabilità p mentre B di 1-p.

Esempio: vi sono 2 due lotterie: la prima permette con un probabilità del 20% di vincere 40k euro, la seconda 30k col 25%, per sapere quale delle due conviene vi sono diversi criteri, essi devono rispettare dei vincoli:

* ordinabilità: con due soluzioni vi sono solamente tre possibili risultati;

A≻B V A~B V A≿B

* transitività: se A≻B e B≻C, allora A≻C
* continuità: se A è preferito a B e quest’ultimo a C, allora esiste una probabilità p tale per cui è possibile rimanere con la soluzione B oppure prendersi A o C con una lotteria [p,A;(1-p),C], in poche parole si può preferire una soluzione intermedia così come giocarsela con quella massima e minima;
* Sostituibilità: se A~B, allora una lotteria tra A e C è indifferente a una con B e C, l’importante è che A e B abbiano la stessa probabilità;
* Monotonia: se A≻B, esiste una probabilità p>=q se e solo se una lotteria con A e B con probabilità p è preferibile o indifferente a una con probabilità q;

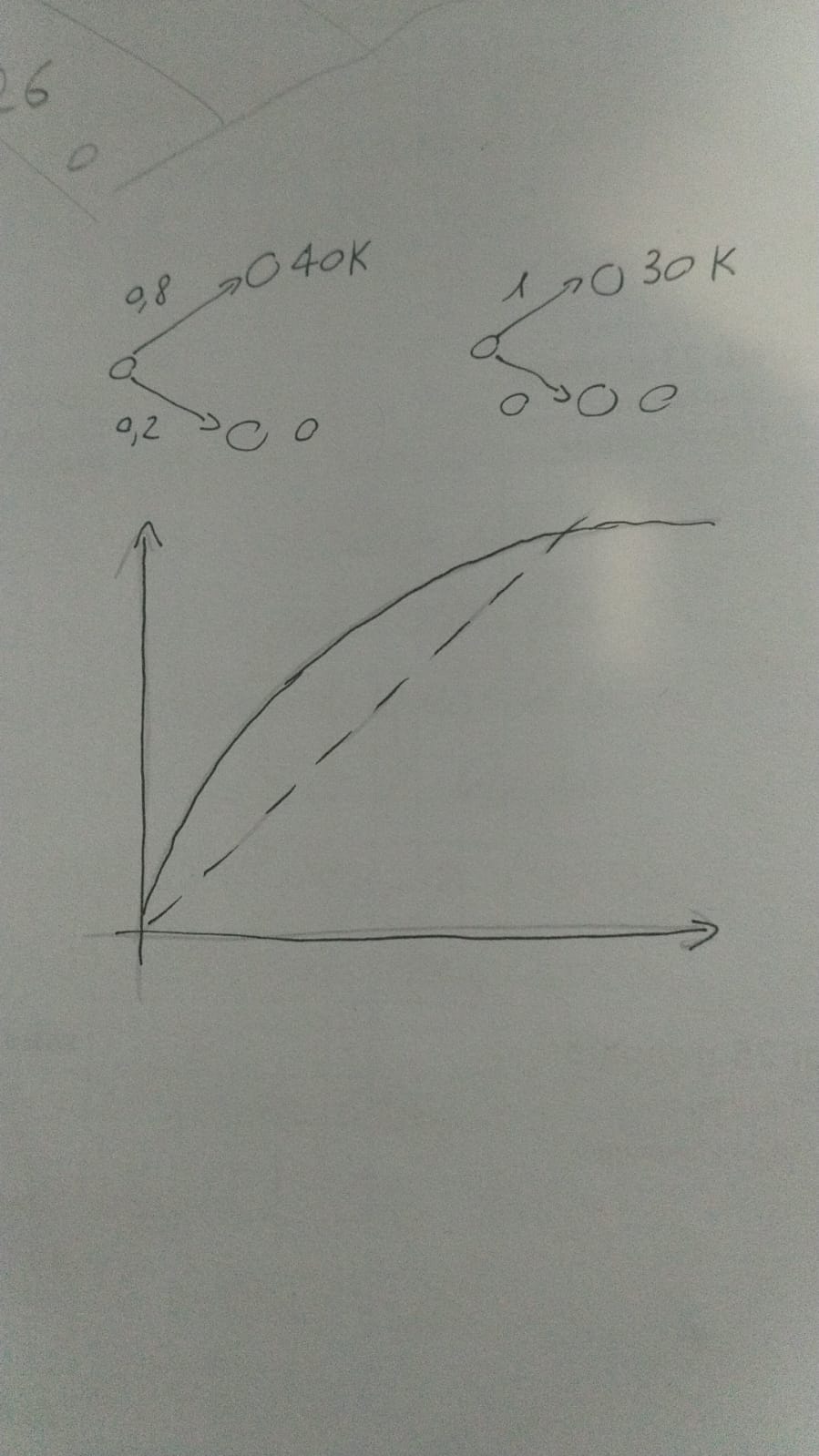
Se questi vincoli vengono tutti soddisfatti, l’insieme di preferenze fornisce comportamenti razionali. Nel caso in cui venga violata la transitività, si ha il cosiddetto money pump, con esso è possibile prendere tutto l’utile dato che tutte le soluzioni sono sia preferite, sia il contrario.

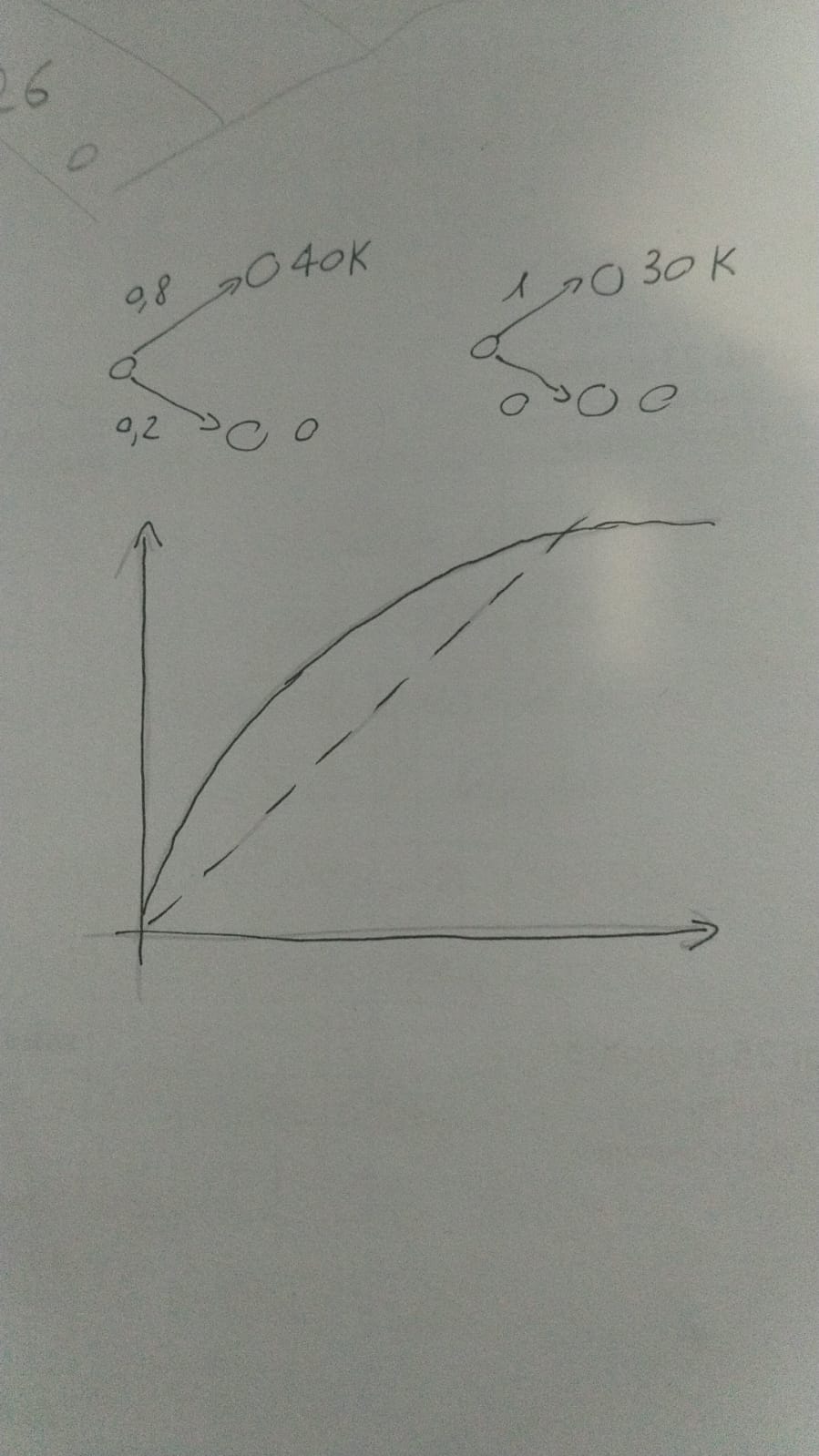
Per indicare il criterio da utilizzare, consideriamo il seguente esempio: Bisogna andare in montagna ma c’è una probabilità che quel giorno piova, se si va c’è la probabilità che quel giorno non piova del 70% con un’utilità di 1, il restante 30% ha probabilità 0, se non si va invece si ha 0,1 di utilità se non piove, altrimenti è 0,6. L’utilità attesa EU è la media delle utilità pesate con le rispettive probabilità, in questo esempio:

EU(V)=0,3\*0+0,7\*1=0,7 EU(NV)=0,3\*0,6+0,7\*0,1=0,25

Conviene andare in montagna.

Un altro metodo consiste di calcolare tutte le utilità attese e prendere quella massima, tuttavia bisogna valutare bene le utilità onde evitare errori, in questi casi si applica la razionalità: è meglio la scelta con utilità massima o quella più probabile?

Esempio: Quale delle due lotterie è meglio?

Nella seconda vi è una vincita sicura, tuttavia la prima ha utilità maggiore e quindi più conveniente, l’utilità infatti non è la vincita monetaria ma quella dello stato in cui ci si trova se si prende una data decisione: se si hanno 0 soldi, vincerne 30k subito è molto più conveniente che tentare la fortuna di vincerne 10k in più, in tal caso la seconda lotteria avrà un’utilità più alta secondo questo punto di vista, questo tuttavia dipende dalla situazione di partenza. Se l’utilità equivale alla vincita, si ha una linea retta, tuttavia non è sempre così e quindi si tende a dare un’utilità più alta alle cifre più basse per evitare i rischi, questo avviene in caso di vincite positive, in caso di perdita, invece, la curva è risk seeking, ovvero si rischia per provare ad annullare la perdita.

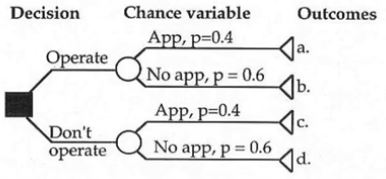
Esempio: un’assicurazione rimborsa completamente la casa in caso d'incendio, avente un 50% di probabilità, con una crescita lineare si può rischiare di ottenere tutto il prezzo, tuttavia si preferisce pagare un premio di costo minore per ottenere solo quello che ho pagato.

#### Paradosso di San Pietroburgo

Si lancia una moneta, se dopo n lanci esce testa, si vincono soldi, tuttavia per partecipare bisogna pagare. Un prezzo giusto si può calcolare attraverso le probabilità, quest’ultima però è infinita dato che n può tendere all’infinito, quindi si può pagare qualsiasi cifra per giocare. Il problema di questo paradosso è che si utilizza il prezzo come utilità, quindi equivale a una linea retta, se invece si utilizza una curva convessa è meglio.

#### Implementazione

In caso di problemi semplici, l’implementazione avviene utilizzando gli alberi di decisione (DT), ovvero alberi in cui i nodi rappresentano le decisioni, le foglie i risultati e gli archi sono le transizioni tra i nodi. Ogni foglia ha una sua probabilità e un’utilità, bisogna quindi stabilire queste ultime per una scelta razionale.Le situazioni migliori e peggiori sono quelle che si trovano più semplicemente, essi infatti hanno rispettivamente utilità 1 e 0, tutti gli altri sono valori intermedi. Calcolate tutte le utilità, si calcola quella attesa e si confrontano i risultati.

Esempio: per verificare se un paziente abbia l’appendicite o meno, il chirurgo deve decidere se operare o meno. Il caso migliore è quello in cui il paziente non viene operato e non ha l’appendicite, quindi ha utilità 1, quello peggiore invece è non operarlo quando è presente la malattia. 

Il problema dell’albero di decisione è che le sottostrutture sono ripetute, per ovviare a ciò si utilizza un modello simile alla rete bayesiana, il diagramma di influenza, l’utilità in tal caso tiene conto sia della decisione, sia della variabile, permettendo così un modello più compatto.

#### Diagrammi di influenza (DI)

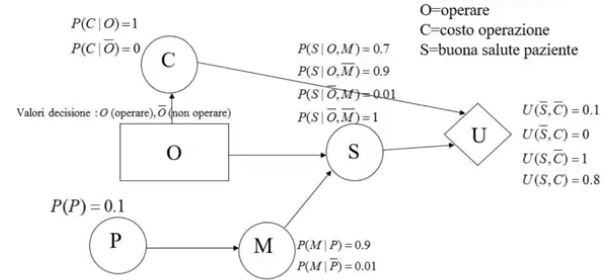
I diagrammi di influenza sono una variante delle RB in cui vi sono i seguenti tipi di nodi:

* Chance (Cerchi): variabili aleatorie;
* Decision (Rettangoli): decisioni da prendere;
* Value (Rombi): risultati di v.a. e decisioni, essi rappresentano l’utilità-

I DI sono DAG che permettono di raggruppare le decisioni in un nodo, gli archi da chance a decision sono detti informazionali ( --->), essi rappresentano il fatto che una v.a. deve essere nota per poter prendere una decisione. I nodi decisione hanno un elenco di possibili azioni, in cui non vi sono quantificazioni, quelli chance hanno invece tabelle di probabilità, la tabella è anche presente nei nodi value, essi hanno tanti valori di utilità (uno per combinazione) quanti sono gli archi entranti.

Esempio: dato un DI, se il paziente ha una probabilità del 10% di essere predisposto a una malattia, per decidere quale decisione prendere, si fa il seguente ragionamento: dato che c’è il 90% dei casi che il paziente sia malato e un 1% di avere la malattia ma non essere predisposti:

* se si opera il il paziente malato, esso guarisce nel 70% dei casi;
* se non lo è e si opera vi è una probabilità del 10% di causare danni;
* se invece non viene operato ed è malato, c’è il 99% di probabilità che stia male.
* se non si opera e non ha la malattia, allora sta bene;

Il costo dell’operazione è massimo se si decide di operare, altrimenti è nullo. L’utilità dipende dalla salute del paziente e dai costi ed è nell’intervallo 0-1, il caso migliore è un paziente sano e un costo nullo, quello peggiore invece è un paziente malato e un costo massimo, quelle intermedie si va a preferenza. Calcolata l’utilità attesa, si scopre in questo caso che è meglio non operare il paziente, se però si è certi di essere predisposti, conviene operare dato che cambia l’utilità. Il DI diventa una RB dato che si eliminano tutte le colonne delle tabelle di probabilità diverse dal valore deciso dal nodo decisione. La valutazione si ottiene valutando ogni possibile decisione,questo però non funziona con gli archi informazionali, in tal caso bisogna conoscere la v.a. da cui dipende. 

Esempio: dato un diagramma di influenza con un nodo decisione senza archi informazionali:

1. Si impostano le variabili osservate all’evidenza, ovvero si associa a ogni variabile il suo valore osservato;
2. Per ogni possibile decisione:
   1. si suppone di prendere tale decisione;
   2. in base a essa si calcolano le probabilità a posteriori dei genitori del nodo utilità, il diagramma diventa quindi una rete bayesiana dal momento che vengono considerate solamente le probabilità riferite alla decisione presa (Nell’esempio precedente, se O è false, allora si considerano le probabilità di C e S in cui O è false).
   3. si calcola l’utilità attesa;
3. Si restituisce la decisione avente utilità attesa massima;

#### Utilità multiattributo

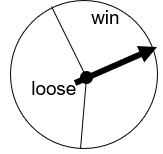
Nel caso reale, i problemi risolvibili con i diagrammi di influenza non sempre dipendono da un solo fattore, di conseguenza diventa importante l’utilità multiattributo.

Esempio: La scelta di una vacanza non dipende solo dal costo ma anche dal tempo di volo, dalla qualità del cibo e delle spiagge, eccetera.

Tutti questi fattori devono essere pesati per scegliere in modo razionale e quindi calcolare l’utilità.

Una cosa che può essere utile è ragionare nell’intervallo [0,1], in questo modo è infatti possibile convertire le utilità in probabilità e viceversa.

Per derivare la funzione di utilità, si utilizza la ruota dell’utilità: una ruota utilizzata per valutare la zona di vincita, più è grande quest’area e più è probabile vincere.

Esempio: si gira la freccia nella ruota dell’utilità, se si ferma su WIN, si vincono 100 euro, altrimenti si perde la stessa cifra. Dato S come lo stato monetario attuale e p come la grandezza dell’area di vincita (e quindi la probabilità), si può calcolare l’utilità attesa:

EU= P\*U(S+100) + (1-p)\*U(S-100)

Come risultato si ha che è preferibile giocare alla ruota se U(S) è minore di EU dato che vi è un potenziale guadagno, in caso contrario conviene non giocare dato che si rischia di perdere denaro, ultimo ma non meno importante è il caso in cui U(S) ed EU si equivalgono, in tal caso si è indifferenti.

L’idea è quella di modificare le due aree della ruota in modo da rimanere indifferenti, se ciò avviene la probabilità p diventa anche l’utilità:

U(X)=p\*U(Max) + (1-p)\*U(Min)=p

dove Max e Min sono rispettivamente il caso migliore e peggiore aventi utilità 1 e 0.

Altri metodi sono associare un costo a ogni outcome con utilità negativa e rinormalizzare il tutto oppure utilizzare le micromorti.

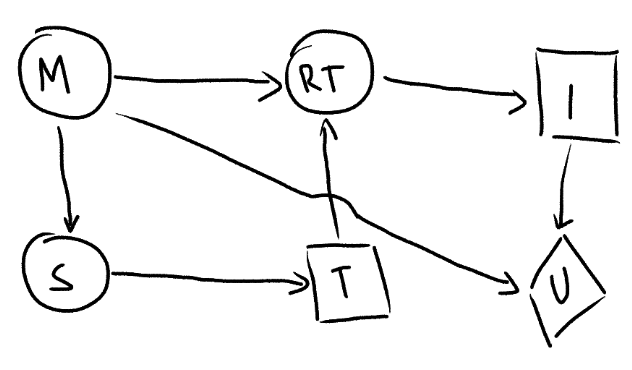
#### Micromorti

Una micromorte indica una probabilità su un milione di un evento negativo, utilizzarle si effettua la seguente procedura:

* si convertono le probabilita in micromorti:

se p=0.5, p\*10^6 = #micromorti = 500,000

* si determina il costo unitario di una micromorte;
* si calcola il costo totale:

totale=costo\*#micromorti

* si normalizzano i risultati.

Esempio:

Il diagramma di influenza a destra indica che:

* Se si ha un sintomo (S), si èuò fare un test diagnostico (T);
* Il risultato del test (RT) ha 3 possibili risultati:
  + 2 di questi indicano il risultato effettivo del test;
  + il terzo è utile quando non si fa il test, in questo modo si gestiscono tutti i casi.
* In base a RT si decide si fare l’intervento o meno (I);
* Prendere la decisione I costa 20 mentre il costo dell’intervento è 5000;
* Se non si effettua I e si è malati (M), la probabilità di morire è del 35%;
* Se invece il paziente non è malato, la probabilità di morire è di 1/50K

Si procede nel seguente modo:

* Paziente Malato e I=no:

#micromorti=P(morte)\*10^6=0.35\*10^6=350K;

totale=20\*350K=7M

* Paziente non malato e I=no:

#micromorti=10^6\* 1/50K=20

totale=20\*20=400;

* Paziente malato e I=sì:

#micromorti=10^6\* 1/50K=20

totale=20\*20+5000=5400;

Eccetera

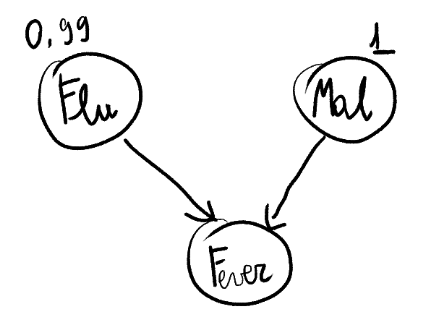
Si ha quindi che:

U(-M,-I)=-400→ 0 U(M,-I)=-7,000,000→ 1 U(-M,I)=-5400→ 0.993 U(M,I)=-9,000→ 0.999

#### Utility independence

Con l’utilità multiattributo, la funzione ha n fattori da considerare, ciò significa che se ognuno degli n attributi ha m possibili valori che può assumere, allora vi saranno n\*m possibili casi.

Questo è un problema dal momento che tutto diverge in modo esponenziale e la funzione di utilità diventa una produttoria, per risolvere ciò si fanno le assunzioni di indipendenza, considerando quindi solo le cose necessario (un modo molto simile al Noisy OR).

Esempio: Per evitare di inserire 4 valori, si usa il noisy OR:

si assume che Fever sia indipendente rispetto alle altre, la differenza è che la sua tabella in questo caso avrà solo i valori necessari per ogni genitore.

Se si ha la febbre, c’è il 99% di possibilità di avere l’influenza e non la malaria oppure il 100% di possibilità che avvenga il contrario.

Il procedimento effettuato è quindi il seguente: Dato un nodo da valutare, si assume che tutti gli altri siano falsi, la stessa cosa verrà effettuata quando si andrà a valutare gli altri nodi.

Nell’utilità, dati due insiemi X e Y che soddisfano questa proprietà, si ha che l’utilità di X non cambia al variare di quella di Y.

Esempio:

X=Colore auto={rosso,verde,nero}

Y=Stile={sport,suv}

Se X è indipendente rispetto a Y, non è necessariamente vero il contrario, l’indipendenza deve essere mutua e ciò permette di diminuire il numero di utilità da n\*m a n, tuttavia bisogna considerare tutte le possibili combinazioni e quindi si ha un numero di prodotti esponenziale.

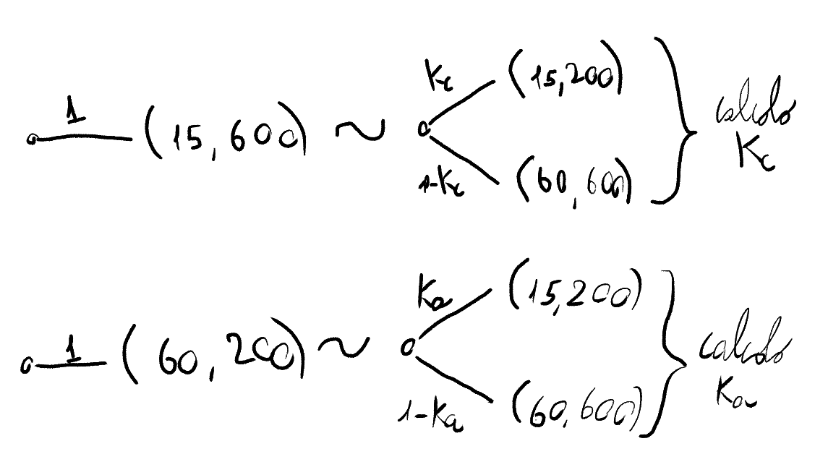
Per far fronte a questo problema si assume il concetto di additività: 2 lotterie tra loro indifferenti sono mutuamente indipendenti quando determinano in modo equiprobabile ogni combinazione di valori degli attributi considerati.

L’utilità diventa quindi una somma pesata delle utilità dei singoli attributi:

U(X1,X2,X3)=k1\*U(X1) + k2\*U(X2) + k3\*U(X3)

#### Determinare i pesi

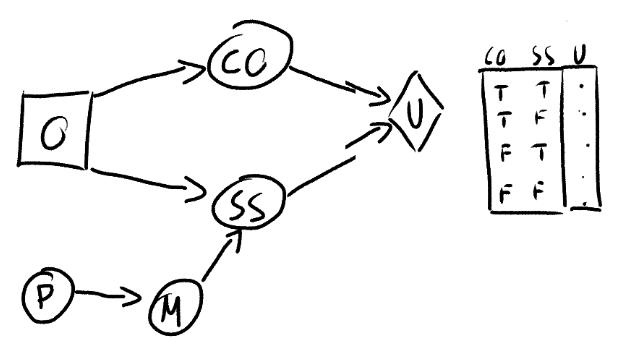
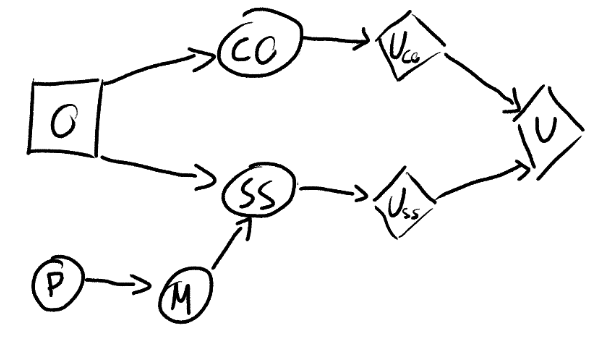
Per determinare i pesi k citati prima, si utilizza una tecnica basata sulle lotterie: date due lotterie X e Y aventi più di due attributi, l’idea è quella di calcolare i pesi nel seguente modo:

* Si considerano le coppie (x0,y0) e (x+,y+) rispettivamente come il risultato peggiore e migliore;
* il peso w(X) viene calcolato prendendo due lotterie, la prima di queste produce (x+,y0) con p uguale a 1, nella seconda invece si ottiene (x+,y+) con probabilità p e (x0,y0) con (1-p);
* Quello che bisogna fare è rendere le due lotterie indifferenti andando a modificare il valore di p, quindi deve essere ricavata in modo che:

p\*U(x+,y+) + (1-p)\*U(x0,y0)=U(x+,y0)

Esempio: Bisogna spargere del pesticida in un dato numero di acri (A) di terreno, dato che il pesticida ha un costo ©, determinare i pesi kA e kC sapendo che si intende minimizzare su C e su A.

E’ possibile inserire queste informazioni nei diagrammi di influenza, per farlo si mettono più input al nodo valore specificando la tabella completa oppure si mettono più nodi valore che vengono poi combinati in uno solo attraverso una formula:



#### 

#### 

In questo modo si hanno più utilità definite su un attributo che vengono tutte combinate nel nodo U, ciò permette di semplificare il tutto concentrandosi su un solo attributo.

#### Strategie (Policies)

In presenza di archi informazionali, viene restituita una policy, ovvero una decisione condizionata da una v.a., essa è ottimale e si prende quella con utilità massima tra tutte.

Esempio: dato il DI, se si prende l’influenza (I) , si ha la febbre (F), quest’ultima è verificabile con un termometro (T) e permane anche dopo (FT). Se si decide di prendere l’aspirina, è possibile diminuire FT ma si rischia una reazione allergica (R), per minimizzarla, la soluzione migliore è non avere FT e R, quella peggiore invece è averle entrambe. Aggiungendo un arco che va da T ad A, le carte in tavola cambiano: a seconda del valore del termometro, si sceglie una decisione anziché l’altra. La policy ottimale è quindi quella di non prendere l’aspirina se non ho la febbre, altrimenti la prendo, l’utilità attesa in questo caso è la media pesata tra tutte quelle ottimali.

#### Valore dell’informazione

Il valore dell’informazione è la differenza tra la miglior decisione con l’informazione e la migliore senza essa, entrambe le decisioni sono utilità attese, quindi:

Val=EU(con inf)-EU(senza inf)

Se l’informazione è gratuita, conviene prenderla, altrimenti bisogna verificare se da essa si trae beneficio o meno, il valore è quindi proporzionale al prezzo equo da pagare per ottenerla.

Esempio: A e B sono giacimenti di petrolio ma solo uno è pieno per un valore di k euro. Ogni giacimento costa k euro e una società geologica offre un sondaggio che dice che in A c’è petrolio al 100%.

Senza informazione: EU(compro A)=EU(compro B)=0.5\*k+0.5\*0-k/2=0

Con informazione:

Policy: se SI compro A, se NO, compro B

EU=P(SI )\*EU(compro A|SI)+P(NO)\*EU(compro B|NO)=

0.5\*(k-k/2)+0.5(k-k/2)=k/2

VOI=k/2-0=k/2

L’informazione non deve costare più di K/2 per essere conveniente, in questo esempio si può dire che è bene sapere l’informazione dato che, con un’informazione di k/2, si ha un guadagno di k/2.

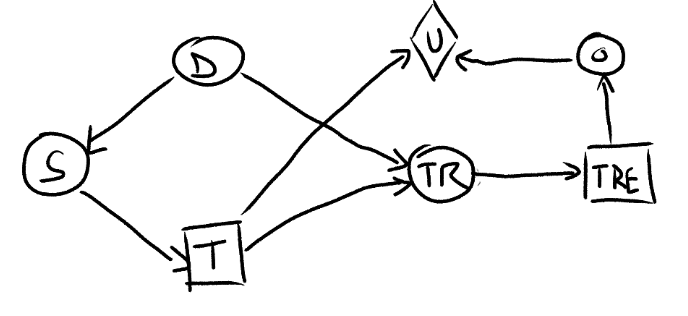
# Generalizzazione di diagrammi di influenza

In presenza di più nodi decisione è opportuno stabilire un ordine in cui effettuarle, per farlo si fanno le seguenti considerazioni:

* Regolarità: si assume che le decisioni non abbiano un ordinamento temporale, quindi deve esistere un cammino diretto che comprende tutti i nodi decisione. Le decisioni vengono quindi presente in sequenza e non “parallelamente”, aggiungendo archi tra i nodi decisioni nel caso in cui non esista un cammino;
* No Forgetting: l’agente non deve dimenticare le decisioni prese in passato, quindi ogni volta che ne prende una deve considerare anche le vecchie scelte oltre che l’evidenza.

Tutto questo può essere un problema dal momento che in caso di lunghe sequenze di decisioni, l’ultima di questa avrebbe troppe informazioni da considerare.

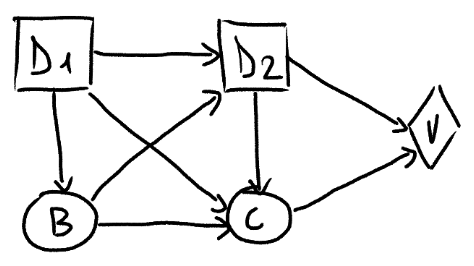
Per ovviare a ciò, esiste una variante dei diagrammi di influenza detta LIMID, dei diagrammi in cui non vi sono nè regolarità, nè no forgetting, tutte le informazioni utili diventano quindi i soli archi entranti.

Esempio:

Nel diagramma di influenza a destra ci sono due decisioni che si trovano sullo stesso cammino, quindi sono ordinate e vengono prese in sequenza.

In mancanza dell’arco TR-TRE, non vi è più un cammino e quindi bisogna stabilire un ordine.

Per prendere la decisione T, basta conoscere solamente la sua evidenza, cioè S, al contrario l’evidenza di TRE non basta per prendere questa decisioni, infatti serve anche quella di T.

Esempio:

Data Ei come l’evidenza al tempo i, per prendere la decisione i-esima Di bisogna conoscere tutta quella precedente (quindi a tempo i-1), nel DI a destra si ha che:

* Per decidere D1,non si ha bisogna di evidenza dal momento che non dipende da niente, quindi E0 è un insieme vuoto;
* Per prendere D2 invece bastano i soli archi entranti, quindi E1={B,C,D1}

Si può quindi dire che:

* Data una sequenza di decisioni D1,...,Dn, l’evidenza Ei è l’unione di tutte le precedenti più l’evidenza per decidere Di-1:

Ei=U(k=0,i-1) Ek U e(Di-1) dove e(Di) indica l’evidenza delle decisione i.

E0=e(Di-1)

* Dato un diagramma di influenza e la sequenza di decisioni D1,...,Dn, per ogni decisione Di, la policy δDi è una funzione che specifica un’istanza di Di per ogni configurazione delle evidenze necessarie.

δDi : ΩEi-1 → ΩDi, nel caso in cui E0 è un insieme vuoto, δD1 appartiene a δD1.

Una policy è quindi una funzione che un particolare stato dell’evidenza necessaria con un altro della decisione.

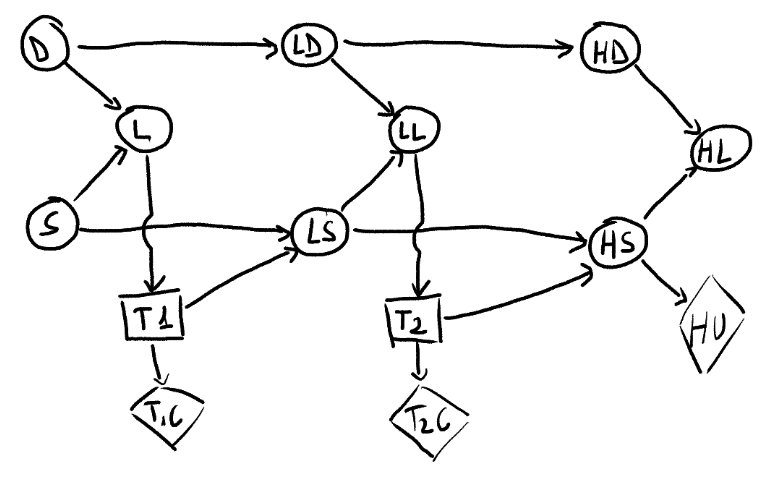
## Limid

Come già detto prima, i limid sono una variante dei diagrammi di influenza in cui non vi è regolarità e no forgetting, la principale differenza rispetto a prima riguarda quindi la policy, infatti in questo caso la funzione mappa uno stato dei genitori con uno della decione:

δD : Ωparents(D) → ΩD se ΩD è vuoto, allora δD è in ΩD

Il vantaggio è che non si pensa più in sequenza dal momento che le informazioni riguardano solo i genitori e ogni decisione può essere presa in modo indipendente dalle altre.

L’insieme di tutte le policy forma la cosiddetta strategia.

Esempio:

Il diagramma di influenza a destra indica come bisogna trattare un meleto, precisamente:

* gli alberi possono essere secchi(D) oppure malati (S) e hanno come sintomo la perdita delle foglie (L);
* Le varianti di D, S e L indicano la stessa cosa ma in periodo di tempo successivo;
* T1 e T2 sono delle decisioni da prendere per fare il trattamento, esse hanno un costo (utilità negativa) descritto dai nodi T1C e T2C;
* HU indica lo stato del meleto.

In base allo stato di salute degli alberi, è opportuno decidere se fare o meno un trattamento e, in base a come si evolve la situazione, effettuarne un secondo o meno.

In questo esempio è meglio utilizzare Limid dal momento che una risoluzione normale porterebbe troppa complessità sulla decisione T2.

L’utilizzo di questo modello comunque non sta nel decidere tutto subito, quindi si può decidere T1 in un primo momento e T2 in un altro.

Limid permette quindi di semplificare tutto rendendo le scelte indipendenti, ciò però non lo rende meglio nè tantomeno sostituibile ai normali diagrammi di influenza, tutto dipende dalle asserzioni del problema.

E’ comunque possibile emulare i diagrammi di influenza con Limid aggiungendo degli archi tra i nodi decisioni, si può quindi dire che Limid è un caso particolare di DN.

# Modelli temporali

I modelli temporali sono particolari modelli in cui le decisioni vanno prese in dati istanti temporali, essi possono essere:

* reti bayesiane dinamiche: estensione delle reti bayesiane in cui si tiene conto del tempo;
* Markov decision processes: modelli in cui l’evoluzione temporali dipende anche da decisioni esterne.

I modelli temporali sono formati da:

* un set di variabili di stato non osservabili a tempo t, esse vengono anche dette nascoste o hidden;
* un seto di variabili di stato osservabili a tempo t, esse rappresentano la cosiddetta evidenza;
* eventuali dipendenze tra variabili;
* un processo campionario il quale permette i cambiamenti;
* L’ipotesi di Markov: lo stato del sistema a tempo t dipende solo da quello precedente, ovvero lo stato a tempo t-1.

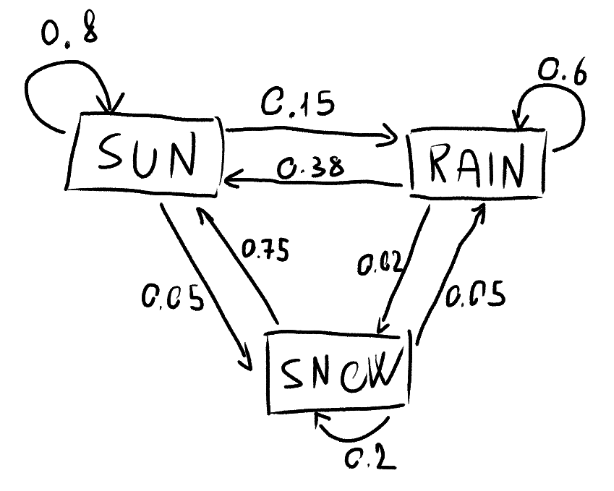
Un processo markoviano è quindi la probabilità di uno stato a tempo t+1 sapende che si è passati in un dato stato a tempo 0, 1 e così via, ciò è uguale alla probabilità di una stato a tempo t+1 sapendo che a tempo t si è passati in un certo stato:

P{St+1 | S0, S1,...,St} = P{St+1 | St}

Quindi è possibile dimenticarsi di tutti gli stati passati tranne l’ultimo.

## Markov Chains

Una Markov chains è un processo markoviano in cui la probabilità di transizione da St a St+1 non dipende da t (omogeneità), ciò permette di modellarla come un automa probabilistico in cui i nodi sono gli stati e gli sono le loro transizioni. Ogni arco ha associata una probabilità di transizione, la somma delle probabilità di tutti gli archi uscenti da uno stato deve essere uguale a 1.

Esempio:

Data la markov chain a destra, si può dire che se oggi c’è il sole:

* c’è la probabilità del 80% che rimanga sereno;
* il 15% che inizia a piovere;
* il 5% che inizi a nevicare.

Una markov chain è formata da:

* un insieme di n stati, ognuno di questi può transire verso altri;
* Ogni transizione da i a j viene effettuate come la probabilità in cui, all’istante Xt+1, si passa a Sj sapendo che a tempo Xt si è in Si:

a(i,j)=P{ Xt+1 = Sj | Xt = Si}

* La distribuzione iniziale i, ovvero la probabilità di uno stato di trovarsi allo stato iniziale (ovvero a tempo 1):

πi=P{X1 = Si}

Se si considera il precedente esempio con π={0.7, 0.25, 0.05}, supponendo di avere sole il primo giorno, seguito da 3 di pioggia e infine 2 di neve, si ha che:

P{Sun} \* P{Rain | Sun} \* P{Rain | Rain}^2 \* P{Snow | Rain} \* P{Snow | Snow} =

=0.7 \* 0.15 \* (0.6)^2 \* 0.02 \* 0.2= 0.0001512

Per trasportare tutto questo nei modelli grafico probabilistici, si modella la markov chain come una rete bayesiana, essa è detta di primo ordine quando dipende dal solo istante precedente, è invece detta di ordine n quando dipende dagli n istanti precedenti, quest’ultima però non si fa perchè matematicamente intrattabile.

I modelli probabilistici sono di ordine finito dato che si analizza il procedimento ponendo un inizio e una fine, il vantaggio è che hanno più variabili Xt nascoste, movito per cui vi sono variabili nascoste ed evidenza connesse tra loro.

## Hidden Markov Models

Gli hidden markov models sono particolari markov chain aventi stati nascosti e che utilizzano i seguenti modelli:

* Un modello di transizione che permette il passaggio da uno stato all’altro:

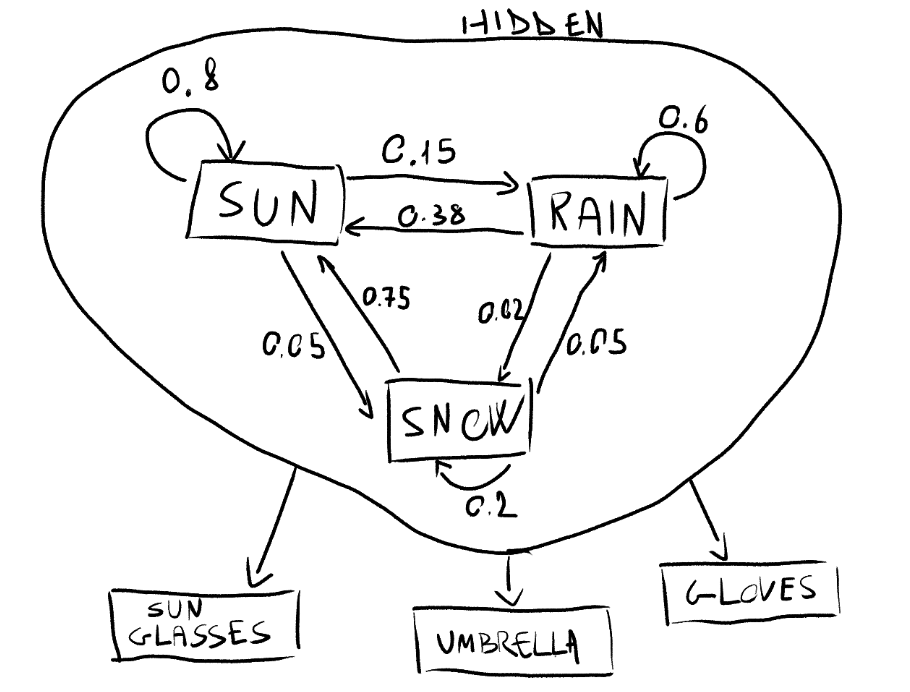
P{Xt+1 | X0,...,Xt} = P{Xt+1 | Xt}

* Un modello sensoriale che indica come le variabile osservabili sono connesse all’evidenza:

P{Et | X0,...,Xt-1,E0,...,Et-1} = P{Et | Xt}

Si può notare che quest’ultimo è stazionario, quindi l’evidenza a tempo t dipende solo dalle variabili nascoste sempre a tempo t.

L’idea degli hidden markov models è che si assume che il modello di transizione sia una markov chain, inoltre per ogni stato nascosto c’è una probabilità di osservare le evidenze, quello che si intende fare è appunto capire gli stati hidden senza osservarli direttamente.

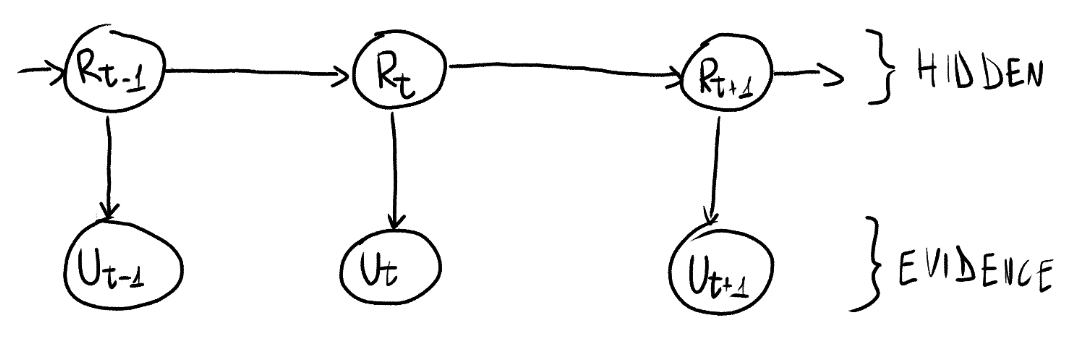
Esempio:

Supponendo di trovarsi in una stanza senza finestre, bisogna capire il tempo atmosferico in base a come è vestita una persona che entra nella stanza.

Arrivano in 7 giorni 7 persone vestite come segue:

Gloves, Gloves, Umbrella, Umbrella, Sunglasses, Umbrella, Umbrella

In base a ciò bisogna trovare la sequenza di hidden più probabile e per farlo si utilizza l’algoritmo di Viterbi.

Anche gli hidden markov model possono essere rappresentati con i modelli grafico probabilistici, come si vede nell’immagine a destra. Non è comunque necessario sbrodolare l’intera sequenza, in questo caso basta utilizzare due nodi in cui si specificano modello di transizione e sensoriale, assumendo che valga per tutto gli istanti temporali.

Le inferenze tra variabili possono essere:

* Filtering: calcolo della probabilità degli stati hidden in base ai segnali emessi;

P{Xt | e1,...,et}

* Prediction: Probabilità di prevedere una situazione futura in base all’evidenza attuale;

P{ Xt+k | e1,...,et} con k>0

* Smoothing: Probabilità di “ricordare” uno stato passato in base all’evidenza attuale;

P{ Xk | e1,...,et} con 0<=k<t

* Most likely explanation: sequenza di nodi hidden più probabile in base all’evidenza.

argmax(x1:t) P{ x1,...,zt | e1,...,et}

## Reti Bayesiane dinamiche

Unendo le inferenze citate prima con gli hidden markov models si formano le reti bayesiane dinamiche, un modello con assunzione markoviana di transizione e una stazionaria sensoriale.

Esempio: Considerando la rete bayesiana dinamica a destra, bisogna capire la durata della batteria e la velocità di un robot aspirapolvere.

B indica la batteria e dipende dall’istanza precedente, viene misurata da BM. X e X’ sono rispettivamente velocità (misurata da Z) e posizione, entrambe dipendono dall’istanza precedente e si influenzano a vicenda, inoltre la posizione dipende anche dalla batteria.

In base agli istanti temporali che si intende osservare, si effettua il cosiddetto unrolling, ovvero si replica la BN tante volte quanti sono gli istanti temporali da analizzare.

## Markov Decision Processes

I markov decision processes sono generalizzazioni di markov chains in cui la transizione avviene tramite l’ausilio di azioni esterne, essi sono formati da quattro elementi:

* un insieme di stati S;
* un insieme di azioni A;
* una funzione di reward, ovvero l’utilità;
* il modello di transizione.

Rispetto alle markov chains, si hanno delle azioni aventi associate delle utilità.

La funzione di transizione non va più da stato a stato ma dalla coppia azione-stato a stato, ciò vuol dire che è la probabilità di transire da uno stato all’altro facendo una determinata azione.

Esempio: un robot su una griglia si può muovere sulle sole caselle bianche, il suo obiettivo è raggiungere +1 evitando -1 e partendo da START.

Dal momento che non si sa se l’azione ha successo o meno (80%) il robot consuma 0.04 ogni volta che si muove, inoltre non può seguire il percorso minimo dal momento che vi è incertezza e quindi può sbagliare. Utilizzando gli MDP, occorre trovare una policy che garantisca l’utilità attesa più alta possibile, infatti per ogni stato bisogna conoscere l’azione più adatta al suo scopo.

Una policy è stazionaria quando non dipende dal tempo e sono utili quando l’orizzonte del problema è finito (vi è un numero di passi preciso), ciò viene meno quando l’orizzonte del problema è infinito.