

INTELLIGENZA ARTIFICIALE

• INTRODUZIONE

L'Intelligenza Artificiale (IA) è il campo che studia la sintesi e l'analisi di agenti computazionali che agiscono intelligentemente. Un problema che si può verificare in IA è che possono esistere problemi che non ammettono soluzioni in tempo polinomiale, ovvero sono problemi detti NP-completi (cioè problemi non deterministici in tempo polinomiale).

N.B.: la classe P è l'insieme dei problemi che possono essere risolti in tempo polinomiale (ovvero, dato un problema L, allora è possibile definire un algoritmo che risolva il problema L (cioè trovi tutti i risultati) in tempo polinomiale), mentre la classe NP rappresenta l'insieme di tutti i problemi che possono essere verificati in tempo polinomiale (cioè; dato un problema L ed una soluzione, allora possiamo verificare se la soluzione soddisfa L in tempo polinomiale).

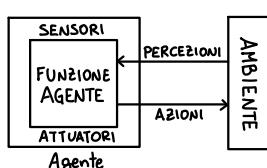
N.B.: un problema si dice risolvibile in tempo polinomiale se il tempo di risoluzione è $t(n) = O(n^k)$.

N.B.: un problema che non ammette soluzione in tempo polinomiale si dice **PROBLEMA INTRATTABILE**.

N.B.: consideriamo la classe P e la classe NP diverse (se dimostrassimo che un qualsiasi problema NP-completo sia risolvibile in tempo polinomiale, otterremmo che le classi P ed NP sono equivalenti, ma ancora non è stata risolta questa questione).

Ad esempio, il problema SAT (il quale, data una stringa di n bit, restituisce un valore vero o falso) è un problema NP-completo. In particolare, un qualsiasi problema NP-completo può essere ricondotto ad un problema SAT.

L'**AGENTE** è un'entità astratta che interagisce con un'ambiente per raccogliere informazioni tramite dei sensori



(canali di input che permettono di percepire le informazioni dell'ambiente) e degli attuatori (canali di output che forniscono delle azioni all'ambiente).

Ad esempio, consideriamo come agente una macchina autonoma. I sensori possono essere il GPS e la camera di prossimità (danno informazioni alla macchina riguardo alla sua posizione ed all'ambiente circostante), mentre gli attuatori possono essere l'acceleratore ed il freno (controllano

la velocità della macchina in base alle informazioni raccolte dai sensori).

Perciò, gli agenti ricevono dei segnali (tramite i sensori) che vengono elaborati per compiere delle azioni sull'ambiente (tramite gli attuatori). Questa attività compiuta dall'agente è detta **FUNZIONE AGENTE**.

Quindi, la funzione agente definisce delle azioni: sulla base delle percezioni catturate ($\xrightarrow{\text{percezioni}}$ $\xrightarrow{\text{Funzione agente}}$ $\xrightarrow{\text{di azioni}}$).

N.B.: una condizione fondamentale è che l'agente sia di tipo computazionale. Perciò, la funzione agente deve essere esprimibile tramite un programma.

Lo **STATO** di un agente permette di definire e memorizzare le possibili sequenze di percezioni che possono essere elaborate dall'agente.

La **TRANSIZIONE DI STATO** è una funzione che aggiorna lo stato dell'agente sulla base degli input rilevati dai sensori.

N.B.: queste informazioni sono contenute nella funzione agente.

Per valutare se un agente è intelligente, dobbiamo analizzare la **PERFORMANCE** della funzione agente.

Un agente si dice **RAZIONALE** quando massimizza (in valore atteso) la misura di performance (ovvero, quando per ogni sequenza di percezioni, l'agente seleziona l'azione da eseguire massimizzando il valore della performance).

Per realizzare la funzione agente, è necessario effettuare delle assunzioni sull'ambiente:

1) ambiente deterministico o stocastico: un ambiente è deterministico quando il suo stato successivo è determinato dal suo stato attuale e dall'azione dell'agente, mentre un ambiente è stocastico quando sono presenti delle variabili inosservabili per cui non possiamo determinare lo stato successivo a partire dallo stato attuale o dall'azione eseguita dall'agente.

N.B.: nella realtà, si possono verificare situazioni non riconoscibili dal sistema per le quali non è possibile definire uno stato futuro (situazione di un ambiente stocastico). Ad esempio, se nella macchina autonoma si oscura la vista, non è più possibile determinare gli ostacoli (in questo caso si dice ambiente parzialmente osservabile).

2) ambiente episodico o sequenziale: un ambiente è episodico quando è necessaria una singola azione, mentre un ambiente è sequenziale quando è necessaria una sequenza di azioni

3) ambiente statico o dinamico: un ambiente è statico se non cambia mentre l'agente calcola la funzione, mentre un ambiente è dinamico se cambia nel momento in cui l'agente calcola la funzione.

4) ambiente continuo o discreto: un ambiente è continuo se puoi effettuare una serie infinita di possibili azioni (quindi non è misurabile), mentre un ambiente è discreto se puoi effettuare un numero finito di azioni (quindi è misurabile).

5) agenti singoli o multipli: possiamo lavorare su un ambiente con un singolo agente oppure con più agenti.

• PROBLEMA DI RICERCA

Si tratta di un problema in cui un agente deve trovare una sequenza finita di azioni (a_1, a_2, \dots, a_T), da passare agli attuatori, che portano da uno stato iniziale s_0 ad uno stato obiettivo (**GOAL**).

N.B.: un goal rappresenta un sottoinsieme dell'insieme degli stati S .

Dobbiamo effettuare delle ipotesi:

- 1) L'ambiente è discreto (perciò, l'insieme degli stati è numerabile);
 - 2) L'ambiente è completamente osservabile (cioè l'agente è sempre in grado di conoscere lo stato attuale dell'ambiente);
 - 3) Le azioni hanno un effetto deterministico (cioè, se un agente compie un'azione, allora questa viene realmente eseguita);
 - 4) L'ambiente è statico;
 - 5) L'agente è guidato dagli obiettivi (**GOAL-DRIVEN-AGENTS**).

ES (sliding tiles puzzle):

stato iniziale			stato goal		
7	2	4		1	2
5		6	3	4	5
8	3	1	6	7	8

Si tratta di un puzzle (3x3) composto da tessere scorrevoli numerate.

Il problema sta nell'eseguire un numero finito di mosse per portare il puzzle da uno stato iniziale S_0 (in cui il puzzle è disordinato) ad uno stato goal (in cui il puzzle risulta ordinato). Un'azione è rappresentata dal movimento della tessera vuota.

Poiche' il puzzle e' composto da 9 tessere, allora il numero possibile di stati in cui puo' trovarsi il puzzle e' 9!

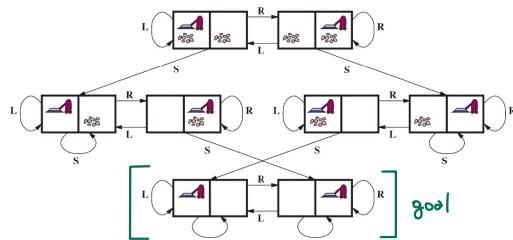
E5 (congettura di Knuth):

Partendo dal numero 4 (stato iniziale), possiamo applicare una sequenza di 3 possibili operazioni (fattoriale, radice quadrata e parte intera inferiore) per ottenere un qualsiasi numero intero.

Ad esempio, $S = \sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}$

In questo caso, le azioni sono rappresentate dalle operazioni che possiamo eseguire.

ES (robot che si muove su 2 celle che possono essere sporche):

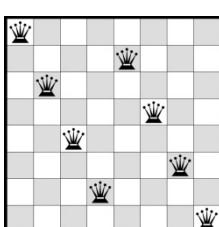


In questo esempio abbiamo un robot che puo' aspirare lo sporco presente nelle 2 celle su cui muoversi.

I possibili stati sono rappresentati da 3 azioni (sporco, spostamento a destra e spostamento a sinistra). Perciò, possiamo avere $2^3 = 8$ possibili stati.

In questo caso, lo stato goal è rappresentato da 2 possibili stati (celle pulite e robot a sinistra oppure celle pulite e robot a destra).

ES (problema delle 8 regine):



Abbiamo una scacchiera su cui possono essere disposte 8 regine.

Lo stato obiettivo è quello in cui nessuna regina si possa trovare sotto attacco.

In questo caso, l'azione è rappresentata dallo spostamento di una regina in un qualsiasi quadrato della scacchiera.

Se consideriamo una scacchiera di dimensione $n \times n$ su cui vogliamo disporre K regine, allora dovremo:

$$\binom{n^2}{K} = \frac{n^2!}{K!(n^2-K)!}$$
 (dove $K = \{0, \dots, n\}$) modi di disporre le regine sulla scacchiera. Perciò, i possibili

stati saranno $\sum_{n=0}^N \binom{N^2}{n}$. Questo numero risulta però troppo elevato (se $n=8$ abbiamo un valore dell'ordine 10^9).

Possiamo semplificare questo risultato effettuando delle assunzioni e degli accorgimenti (ottenendo n! possibili stati).

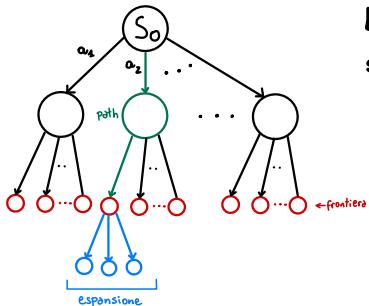
Perciò, per un problema di ricerca dobbiamo definire le seguenti componenti:

- insieme degli stati S ed insieme delle azioni A ;
 - N.B.: lo spazio degli stati puo' essere rappresentato da un grafo in cui i vertici sono gli stati e gli archi sono le azioni.
 - stato iniziale $s_0 \in S$;
 - stato goal $s_T \in S$;
 - funzione Action(s): $S \rightarrow 2^A$ (ritorna il numero di azioni che un agente puo' compiere in uno stato $s \in S$);
 - funzione Result: $S \times A \rightarrow S$ detta modello di transizione (ovvero, $\text{Result}(a, s)$, con $a \in A$ ed $s \in S$, ritorna lo stato raggiunto s' quando si esegue l'azione a nello stato attuale s);
 - funzione goal-test(s): $S \rightarrow \{0, 1\}$ (ovvero, una funzione booleana che ritorna 0 (vero) se e' stato raggiunto il goal, oppure 1 (falso) altrimenti);
 - funzione c detta step-cost (ovvero, $c(a, s)$, con $a \in A$ ed $s \in S$, ritorna il costo che comporta un'azione a eseguita sullo stato s);
 - funzione cost detta path-cost (ovvero, $\text{cost}(s_0, a_1, a_2, \dots, a_T) = \sum_{t=1}^T c(a_t, s_{t-1})$ ritorna il costo totale dovuto da una sequenza di azioni eseguite su un insieme di stati (e' dato dalla somma di tutti gli step-cost)).

N.B.: una soluzione rappresenta un cammino che va dallo stato iniziale s_0 allo stato goal s_T .

• ALBERO DI RICERCA

L'albero di ricerca rappresenta una struttura dati matematica di dimensione infinita (perciò non può essere memorizzato).



La radice di tale albero rappresenta lo stato iniziale s_0 , mentre ogni nodo dell'albero contiene uno stato $s \in S$ e rappresenta un cammino (ovvero una sequenza di azioni).

Perciò, un nodo dell'albero n è una struttura dati che contiene alcune informazioni come:

- lo stato $s \in S$ (indicato con l'attributo $n.state$);
 - il nodo padre (indicato con l'attributo $n.parent$ e definito da un puntatore al nodo padre);
 - l'azione applicata al nodo padre che ha generato questo nodo (indicata con l'attributo $n.action$);
 - il costo del cammino totale, dallo stato iniziale fino a questo nodo (indicato con l'attributo $n.path_cost$);
- N.B.: il $path_cost$ viene definito con una funzione $g(n)$.

Applicando la funzione **Result** ad un determinato nodo dell'albero (ovvero, eseguendo un'azione $a \in A$ ad uno stato $s \in S$), possiamo passare ad un nodo figlio. Questa operazione può essere definita come **ESPANSIONE** di un nodo.

Perciò, l'operazione di espansione di un nodo permette di generare dei nuovi nodi figli applicando tutte le possibili azioni ad un determinato nodo dell'albero.

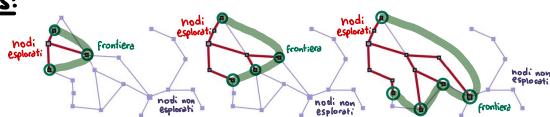
Definiamo la **FRONTIERA** (detta anche open list) come una struttura dati che possiede le seguenti operazioni:

- **empty** (ritorna vero se la frontiera è vuota, altrimenti ritorna falso);
- **pop** (estrae i nodi della frontiera);
- **push** (inserisce i nodi nella frontiera);

N.B.: spesso si rappresenta la frontiera attraverso una coda (ordinata secondo tecniche diverse (FIFO, LIFO o priorità)).

Proprietà (separazione della frontiera): ogni cammino che connette un nodo già esplorato con un nodo non ancora esplorato deve intersecare la frontiera.

ES:



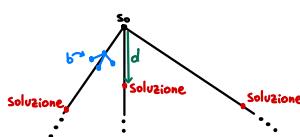
N.B.: la frontiera contiene i nodi candidati all'espansione.

Un **ALGORITMO DI RICERCA** permette di definire una soluzione (oppure un'indicazione di fallimento se non riesce a trovare la soluzione) di un particolare problema di ricerca.

Proprietà (algoritmo di ricerca):

- **Completezza**: se esiste una soluzione, allora l'algoritmo garantisce di trovarla;
- **Ottimalità**: l'algoritmo garantisce di trovare la soluzione con costo minimo;
- **Tempo e spazio**: l'algoritmo richiede un certo tempo di esecuzione ed un certo spazio di memorizzazione.

Definiamo:



- il valore d come la profondità della soluzione meno profonda (depth of solution);
- il valore b come il fattore di ramificazione (branching factor), ovvero il numero di figli generati da un nodo;
- il valore m come la massima lunghezza di un cammino di ricerca.

N.B.: il tempo e lo spazio di un algoritmo di ricerca possono essere espressi come funzioni di b e d .

Inoltre, possiamo definire 2 classi di soluzione:

1) Blind search (ricerca non informata);

2) Heuristic search (ricerca informata);

• BEST-FIRST SEARCH

L'algoritmo di ricerca best-first permette di definire quale nodo n espandere sulla base di una certa funzione di valutazione $f(n)$ calcolata per ogni nodo.

Vogliamo quindi espandere la frontiera con il nodo che possiede il valore di $f(n)$ più piccolo, finché non arriviamo allo stato goal.

N.B.: a seconda di come viene implementata la funzione $f(n)$, possiamo realizzare algoritmi di ricerca diversi.

Ogni nodo viene inserito nella frontiera solo se non era ancora stato raggiunto. Altrimenti viene reinserito solo se è stato raggiunto da un cammino con $path_cost$ minore. Questo permette di non rivisitare gli stati già visti (possiamo implementare questa situazione attraverso una tabella hash le cui chiavi sono gli stati (detto reached)).

Vediamo quindi come possiamo implementare l'algoritmo di ricerca best-first:

```

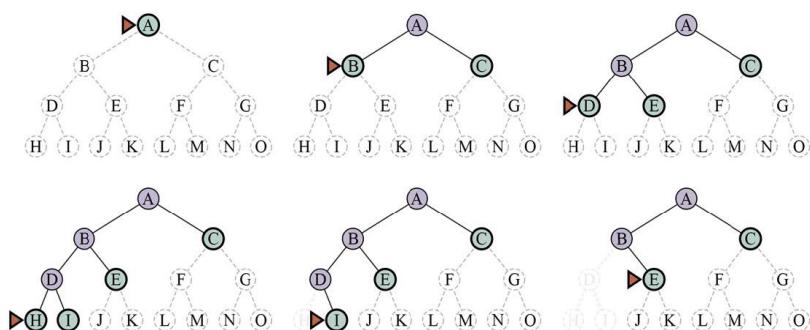
function BEST-FIRST-SEARCH(problem,f) returns a solution node or failure
  node  $\leftarrow$  NODE(STATE=problem.INITIAL)
  frontier  $\leftarrow$  a priority queue ordered by f, with node as an element
  reached  $\leftarrow$  a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node  $\leftarrow$  POP(frontier)
    if problem.Is-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s  $\leftarrow$  child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s]  $\leftarrow$  child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s  $\leftarrow$  node.STATE
  for each action in problem.ACTIONS(s) do
    s'  $\leftarrow$  problem.RESULT(s, action)
    cost  $\leftarrow$  node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)

```

• DEPTH-FIRST SEARCH

L'algoritmo di ricerca depth-first espande sempre il nodo della frontiera più profondo. Tale algoritmo può essere implementato a partire dall'algoritmo best-first con funzione di valutazione $f(n) = -n.\text{depth}$ (ovvero $f(n)$ è l'opposto della profondità del nodo).



Il depth-first ritorna la prima soluzione che trova (anche se non è quella con costo minore). Perciò, in generale, tale algoritmo non è ottimale.

Inoltre, il depth-first non è completo quando abbiamo infiniti stati (infatti, in questo caso si possono verificare dei cicli infiniti per i quali non è possibile trovare una soluzione).

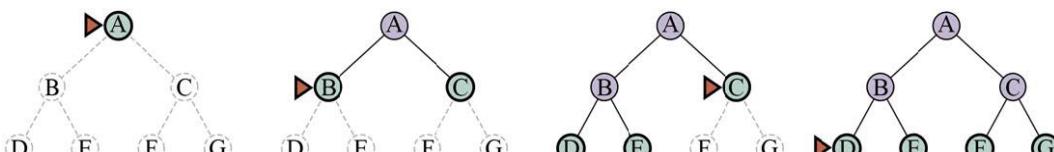
N.B.: in questa situazione (stati infiniti), il tempo di esecuzione dell'algoritmo può essere infinito.

Se, invece, consideriamo un problema che possiede un numero di stati finito, allora il tempo di esecuzione sarà $O(b^m)$, mentre lo spazio richiesto sarà $O(b \cdot m)$ che risulta essere un valore piccolo (vantaggio del depth-first).

Inoltre, tale algoritmo non richiede la tabella reached e la frontiera ha una struttura LIFO che causa la ricerca ^(depth) in profondità.

• BREADTH-FIRST SEARCH

L'algoritmo di ricerca breadth-first viene utilizzato quando tutte le azioni richiedono lo stesso costo. In tale algoritmo, prima si espanderà la radice, poi si espanderanno i successori della radice e così via.



Possiamo implementare il breadth-first a partire dall'algoritmo best-first con funzione di valutazione $f(n) = n.\text{depth}$ (ovvero, $f(n)$ è la profondità del nodo (cioè il numero di azioni necessarie per raggiungere il nodo)).

I vantaggi dell'algoritmo breadth-first sono: la completezza (anche nel caso di stati infiniti) e l'ottimalità (se lo step-cost è costante per tutti i nodi).

Invece, il tempo di esecuzione e lo spazio sono esponenziali ($O(b^d)$).

Inoltre, tale algoritmo richiede la tabella reached e la frontiera ha una struttura FIFO che causa la ricerca ^(breadth) in ampiezza.

• UNIFORM-COST SEARCH

L'algoritmo di ricerca uniform-cost viene utilizzato quando le azioni richiedono costi differenti. Tale algoritmo viene implementato a partire dall'algoritmo best-first con funzione di valutazione $f(n) = g(n)$ (ovvero $f(n)$ è il costo del cammino che va dalla radice al nodo (cioè il path-cost)).

L'algoritmo uniform-cost espnde il nodo meno costoso (ovvero il nodo che comporta path-cost minore).

Tale algoritmo e' completo ed ottimale.

Dim (ottimalit` dell'uniform-cost search):



Consideriamo un nodo di partenza (start) ed una frontiera in cui ogni cammino ha pi` o meno lo stesso costo. Percio', consideriamo un cammino che raggiunga il goal ottimo n^* . Notiamo che, sulla frontiera e' presente un nodo n' che passa per il cammino ottimo. Sicuramente, se ogni azione costa $E > 0$ e la lunghezza del cammino che va da n' ad n^* e' ℓ , allora il path-cost da n' ad n^* sara' $C = E \cdot \ell > 0$ (ovvero, il path-cost da n' ad n^* e' positivo). Supponiamo, per assurdo, che esista un goal subottimo n (ovvero un ottimo migliore di n^*) sulla frontiera.

Allora avremo $g(n) > g(n^*)$ (poiche' n e' un subottimo).

Ma, $g(n^*) = g(n') + C > g(n')$ poiche' $C > 0$. Percio', non e' possibile che $g(n) > g(n')$

CVD

Chiamato C^* il costo della soluzione ottima ed $E > 0$ il costo minimo di ciascuna azione, allora la complessita' del tempo e dello spazio sara' $O(b^{1+\lfloor C/E \rfloor})$ (che risulta essere molto maggiore di $O(b^d)$).

N.B: nel caso in cui tutte le azioni richiedano lo stesso costo, allora la complessita' diventa $O(b^{d+1})$ che assomiglia a quella del breadth-first search (infatti, l'algoritmo uniform-cost e' una ricerca in ampiezza sul path-cost).

La frontiera ha una struttura a priorita'.

• DEPTH-LIMITED SEARCH ED ITERATIVE-DEEPING SEARCH

L'algoritmo di ricerca depth-limited sfrutta l'algoritmo deep-first imponendo un limite ℓ sulla profondita' (ovvero, esclude i successori dei nodi che si trovano ad una profondita' ℓ). Questo permette di avere un algoritmo con una complessita' di spazio $O(b\ell)$, evitando problemi dovuti allo spazio degli stati infinito (ad esempio, cicli infiniti).

N.B: la complessita' del tempo sara' $O(b^\ell)$.

Invece, l'algoritmo di ricerca iterative-deepening applica iterativamente l'algoritmo depth-limited incrementando ad ogni iterazione il valore del limite ℓ , finche' non trova una soluzione (oppure un valore di errore o un valore di interruzione).

Tale algoritmo combina i benefici della depth-first (la complessita' dello spazio sara' $O(bd)$ se trova una soluzione, oppure $O(bm)$ altrimenti) e quelli della breadth-first (l'algoritmo iterative-deepening e' ottimale per problemi in cui tutte le azioni hanno lo stesso costo ed e' completo su spazi di stati finiti).

N.B: la complessita' del tempo sara' $O(b^\ell)$ se trova una soluzione, oppure $O(b^m)$ altrimenti.

Vediamo in dettaglio lo pseudocodice di tale algoritmo:

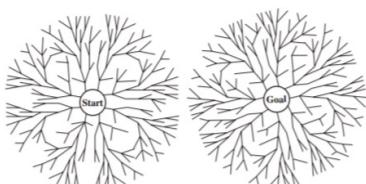
```

function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
  
```

• BIDIRECTIONAL SEARCH

L'algoritmo di ricerca bidirectional si espnde contemporaneamente in avanti, dallo stato iniziale, ed all'indietro dallo stato goal fino a che le due frontiere non si incontrano ottenendo la soluzione.



N.B: questo algoritmo puo' essere eseguito soltanto quando si conosce il goal.

La particolarita' di questo algoritmo e' che il tempo di esecuzione ha una complessita' $O(b^{d/2})$ (poiche' le due ricerche si incontrano a met` strada) che risulta essere molto minore dei tempi di esecuzione degli algoritmi visti fino ad ora (ovvero $O(b^d)$).

• RIASSUMENDO

	Deep-First	Breadth-first	Uniform-cost	Iterative-Deepening	Bidirectional
Completo	X	✓	✓	✓	✓
Ottimale	X	✓	✓	✓	✓
Tempo	$O(b^m)$	$O(b^d)$	$O(b^{1+\lceil C/\epsilon \rceil})$	$O(b^d)$	$O(b^{d/2})$
Spazio	$O(bm)$	$O(b^d)$	$O(b^{1+\lceil C/\epsilon \rceil})$	$O(bd)$	$O(b^{d/2})$

N.B.: gli algoritmi visti fino ad adesso sono blind search (ovvero ricerca non informata) e ciò risulta essere un grosso limite.

• HEURISTIC SEARCH

Gli algoritmi di ricerca informata (heuristic search) permettono di trovare le soluzioni in modo più efficiente degli algoritmi di ricerca non informata (blind search). Questo perché utilizzano una funzione euristica, definita come $h(n)$, la quale stima il costo del cammino che va dallo stato del nodo n allo stato goal (perciò, dato uno stato, allora $h(n)$ stima la distanza minore per arrivare al goal).

• GREEDY BEST-FIRST SEARCH

L'algoritmo di ricerca greedy best-first è un algoritmo di ricerca informata (heuristic search). Tale algoritmo può essere implementato dal best-first con funzione di valutazione $f(n)=h(n)$.

Perciò si espande il nodo con valore $h(n)$ minore (ovvero, il nodo che appare più vicino al goal).

Tale algoritmo, però, non è completo nel caso di stati infiniti ed inoltre non è ottimale (poiché un nodo che apparentemente sembra più vicino al goal non è detto che lo sia effettivamente).

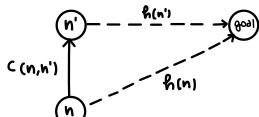
• A* SEARCH

L'algoritmo di ricerca A* (pronunciato A star) è l'algoritmo di ricerca informata più comune. Tale algoritmo può essere implementato dal best-first con funzione di valutazione $f(n)=g(n)+h(n)$.

Perciò, la funzione $f(n)$ rappresenta il costo stimato del percorso migliore che va dal nodo n allo stato goal.

Per garantire il funzionamento di tale algoritmo è però necessario effettuare 2 assunzioni:

- $h(n)$ deve essere ammissibile, ovvero non deve sovrastimare il costo per raggiungere il goal;
- $h(n)$ deve essere consistente, ovvero $h(n) \leq c(n, n') + h(n')$ dove $c(n, n')$ è lo step-cost per andare da n ad n' .

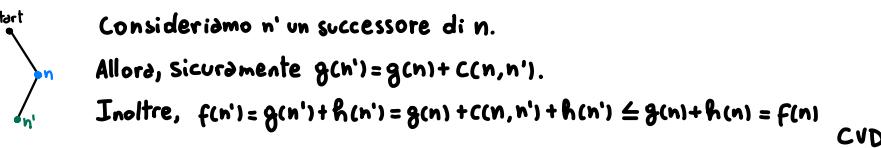


N.B.: la proprietà di consistenza è simile alla disugualanza triangolare (ovvero, l'euristica $h(n)$ è minore o uguale al costo $c(n, n')$ più l'euristica $h(n')$).

N.B.: la consistenza implica l'ammissibilità, ma il viceversa non è vero in generale.

Lemma: se $h(n)$ è consistente, allora il valore di $f(n)$ lungo un qualunque cammino è non decrescente.

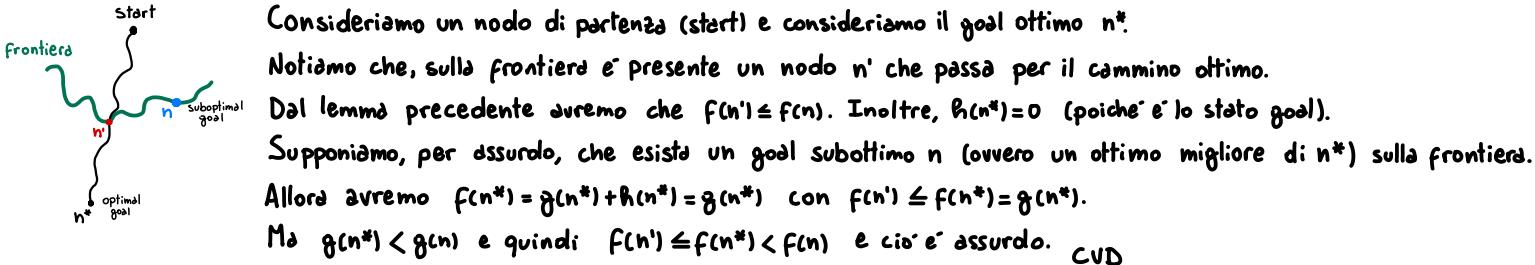
Dim (lemma):



Teorema: se $h(n)$ è ammissibile, allora l'algoritmo di ricerca A* è ottimale.

Teorema: se $h(n)$ è consistente, allora l'algoritmo di ricerca A* è ottimale.

Dim (teorema):



Def: consideriamo due diverse funzioni euristiche $h_1(n)$ ed $h_2(n)$. Se $h_2(n) \geq h_1(n) \forall n$, allora si dice che h_2 è più informata di h_1 (oppure si dice che h_2 domina h_1).

N.B.: date 2 idee euristiche diverse (h_1 ed h_2), allora consideriamo l'euristica dominante (poiché vogliamo il valore di h più grande).

Es:

8	1	2
6		3
7	5	4

	1	2
3	4	5
6	7	8

Consideriamo le seguenti funzioni euristiche:

- $h_1 = \# \text{tessere fuori posto}$
- $h_2 = \sum_{i=1}^n |x_i - x'_i| + |y_i - y'_i|$ detta distanza Manhattan (dove (x_i, y_i) è la posizione di i nello stato corrente ed (x'_i, y'_i) è la posizione di i nello stato goal).

Notiamo che $h_2(n) \geq h_1(n)$, ovvero $h_2(n)$ è più informata (perciò useremo h_2).

Teorema: Sia S_1 un insieme di nodi espansi dall'algoritmo A* con la funzione euristica h_1 e sia S_2 un insieme di nodi espansi dall'algoritmo A* con la funzione euristica h_2 . Se h_2 domina h_1 , allora $S_2 \subseteq S_1$ (perciò il tempo impiegato per S_2 è minore di quello impiegato per S_1).

N.B.: nel caso in cui avessimo due euristiche $h_1(n)$ ed $h_2(n)$ per le quali $h_1(n) \leq h_2(n)$ ed $h_2(n) \leq h_1(n) \forall n$ (ovvero nessuna delle due euristiche domina l'altra), allora sceglieremo punto per punto un'euristica $h_3(n) = \max\{h_1(n), h_2(n)\}$. In particolare, $h_3(n)$ è ancora ammmissible. Tale regola può essere applicata in generale per m funzioni euristiche (si sceglie sempre l'euristica maggiore).

• OSSERVAZIONI SULL'EURISTICA

Se avessimo un'euristica perfetta, allora il tempo di esecuzione diventerebbe $O(d)$, poiché ad ogni passo effettueremmo sempre la scelta migliore (infatti avremmo un branching factor effettivo $B=1$, ovvero come se avessimo una sola azione disponibile ad ogni passo). Questa situazione estrema risulta però irrealizzabile.

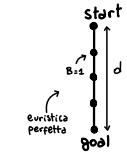
Se, invece, avessimo un albero perfettamente bilanciato (ovvero ogni nodo ha lo stesso numero di azioni), allora il numero di nodi espansi sarebbe $N = B + B^2 + B^3 + \dots + B^d = \frac{B}{B-1}(B^d - 1)$, dove N è un numero empirico (cioè ottenuto tramite esperimenti).

Possiamo quindi ricavare il valore di B dall'equazione $N = \frac{B}{B-1}(B^d - 1)$.

Tale valore non è valido in generale per ogni situazione, ma viene calcolato eseguendo il programma e misurando il valore di N .

Possiamo inoltre definire un parametro di performance, detto penetranza, come il rapporto tra la profondità in cui è situato un nodo ed il valore empirico del numero di nodi espansi N , ovvero $P = \frac{d}{N}$.

N.B.: se l'euristica è perfetta, allora $P=1$.



Una tecnica per migliorare il tempo di esecuzione può essere quella di memorizzare alcune configurazioni del problema.

In questo modo, è facile calcolare il numero di azioni necessarie per raggiungere lo stato goal.

Esc pattern databases (Korf & Felver):

stato goal			
	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

?	?	3
?	?	7
?	?	11
12	13	14

Ipotizziamo di memorizzare la configurazione in cui conosciamo la cornice destra e bassa del puzzle (il resto del puzzle può essere completato in qualsiasi modo dai numeri rimanenti).

Le possibili configurazioni in cui può trovarsi il puzzle saranno quindi $\frac{16!}{(16-8)!} \approx 5 \times 10^8$ (stati che possiamo memorizzare).

In questo modo, quando ci troviamo in una delle possibili configurazioni, possiamo conoscere il numero di mosse necessarie a raggiungere lo stato goal.

• RICERCA LOCALE

Mentre negli algoritmi precedenti volevamo conoscere la sequenza di azioni che permettevano di raggiungere lo stato goal a partire dallo stato iniziale, in questo caso non ci interessa la sequenza di azioni con cui arriviamo allo stato goal (cioè non ci interessa la configurazione) ma ci interessa soltanto trovare uno stato goal.

Ad esempio, nel problema delle 8 regine ci interessa trovare una soluzione per posizionare le regine sulla scacchiera in modo che nessuna regina sia sotto attacco, ma non ci interessa la sequenza con cui posizioniamo le regine.

Le caratteristiche di questi algoritmi di ricerca locale sono:

- assenza di frontiera;
- esplorano il vicinato di uno stato (non guardano in profondità);
- non memorizzano i cammini;
- non sono sistematici;
- si accontentano di soluzioni subottime (poiché la ricerca di una soluzione ottima è troppo costosa);
- richiedono poca memoria;
- associano ad ogni stato uno score.

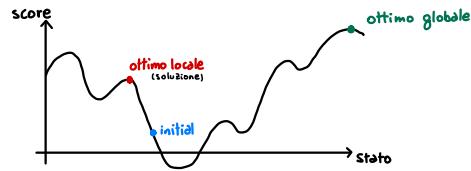
Tali algoritmi assomigliano a problemi di ottimizzazione (problem: di massimo o di minimo).

• HILL CLIMBING

L'algoritmo di ricerca locale hill climbing tiene traccia dello stato corrente ed ad ogni iterazione si muove verso lo stato vicino con score migliore. Tale algoritmo termina quando si raggiunge una situazione in cui nessun vicino ha score migliore. Perciò, ad ogni iterazione, il nodo attuale viene rimpiazzato con il vicino migliore.

Vediamo lo pseudocodice dell'algoritmo hill climbing:

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    current ← problem.INITIAL
    while true do
        neighbor ← a highest-scored successor state of current
        if SCORE(neighbor) ≤ SCORE(current) then return current
        current ← neighbor
```



Tale algoritmo è molto semplice, ma non sempre risulta funzionale. Si chiama hill climbing poiché ad ogni passo si seleziono il vicino con valore più alto (si scala la collina) per problemi di massimizzazione, oppure si seleziona il vicino con valore più basso (si scende la collina) per problemi di minimizzazione.

N.B.: puo' accadere che l'algoritmo determini un ottimo locale che non soddisfa la nostra soluzione (perche', ad esempio, vogliamo l'ottimo globale). Per risolvere questo problema si puo' eseguire un **RANDOM RESTART**, ovvero si esegue nuovamente l'algoritmo in un nuovo stato iniziale scelto in modo casuale, finché non viene trovata una soluzione soddisfacente (si esegue l'algoritmo n volte su diversi stati iniziali e si considera la soluzione con score più alto).

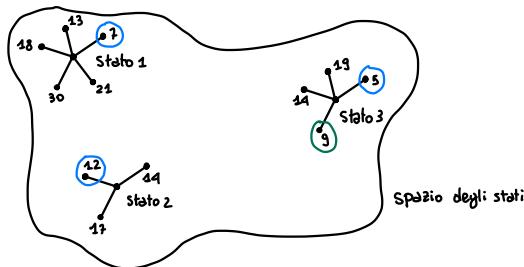
• BEAM SEARCH



L'algoritmo di ricerca locale beam (ovvero torcia) tiene traccia di K diversi stati (invece che di uno solo) definiti in modo casuale (con random restart). Ad ogni iterazione, vengono generati tutti i successori dei K stati.

Se uno di tali successori rappresenta lo stato goal, allora l'algoritmo termina. Altrimenti si selezionano i successori con score migliore e si riesegue l'algoritmo.

ES:



In questo caso applichiamo l'algoritmo con dimensione del beam $K=3$. Ad ogni passo sviluppiamo i successori e consideriamo quelli con score migliore.

Notiamo che possono esserci più valori migliori trovati tra i successori di uno stato rispetto a quelli trovati da un altro stato (ad esempio, lo stato 2 trova il successore con score=12, ma lo stato 3 possiede uno score=9).

Potremmo quindi considerare questo nuovo stato come successore dello stato 2.

• SIMULATED ANNEALING

L'algoritmo di ricerca locale simulated annealing sfrutta un principio fisico, detto annealing (ovvero ricottura), in cui si surriscalda un metallo (per poterlo lavorare più facilmente) e lo si lascia raffreddare lentamente (diminuendo la sua temperatura, fino a farla arrivare a zero). Perciò, si considera la funzione obiettivo ed una seconda funzione molto più rilassata rispetto alla prima. Vogliamo quindi fare collassare la funzione rilassata sulla funzione obiettivo, secondo il criterio $P_i = \frac{1}{Z} e^{-\frac{E_i}{kT}}$ dove P_i rappresenta la probabilità di avere la configurazione E_i , k è la costante di Boltzmann, T è un parametro ed Z è una funzione di normalizzazione (ovvero $Z = \sum_i e^{-\frac{E_i}{kT}}$).

In questo modo è più probabile ottenere il massimo globale.

Una mossa viene effettuata sicuramente solo se $P_i = \frac{P_f}{P_i} = e^{-\frac{E_f - E_i}{kT}} \geq 1$ (dove P_f è lo stato di arrivo e P_i è lo stato di partenza).

Altrimenti, se $P < 1$, allora la mossa viene effettuata con una probabilità P . Perciò, le mosse ottime vengono sempre eseguite mentre le mosse non ottime vengono eseguite solo con una certa probabilità.

Il parametro T deve tendere lentamente a zero con una funzione $T(t) = \frac{T_0}{\log(t+\alpha)}$



• PROGRAMMAZIONE DEI VINCOLI (constraint programming)

• CONSTRAINT SATISFACTION PROBLEM (CSP)

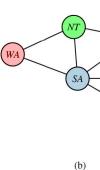
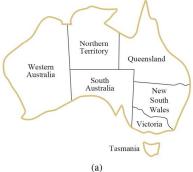
Un problema di soddisfacimento dei vincoli è costituito da una tripla (X, D, C) , dove:

- $X = \{X_1, X_2, \dots, X_n\}$ è un insieme finito di **VARIABILI**;
- $D = \{D_1, D_2, \dots, D_n\}$ è un insieme finito di **DOMINI**, in cui ogni singolo dominio rappresenta un insieme finito;
- $C = \{C_1, C_2, \dots, C_k\}$ è un insieme finito di **VINCOLI**, in cui ogni vincolo C_i rappresenta la coppia $C_i = (\text{scope}, \text{relationship})$;

N.B.: gli insiemi X e D hanno uguale dimensione n (ad ogni variabile X_i è associato un dominio D_i), mentre l'insieme C ha dimensione K .
 Una soluzione è un assegnamento completo (ovvero, a tutte le variabili) che soddisfi tutti i vincoli.
 L'insieme delle soluzioni può essere grande, ma spesso ci interessa trovare una sola soluzione (dobbiamo trovarle tutte solo nei casi in cui stiamo cercando la soluzione ottima).

Se il problema non possiede soluzioni, allora si dice che è insoddisfacibile.

Es(mappa politica dell'Australia):



L'obiettivo è quello di colorare i singoli stati della mappa (a) in modo che due stati adiacenti non abbiano lo stesso colore. Possiamo formalizzare il problema utilizzando il grafo (b) (detto **GRAFO DEI VINCOLI**) in cui i vertici rappresentano gli stati e gli archi indicano la vicinanza tra due stati.

In questo caso, possiamo considerare l'insieme delle variabili gli stati (ovvero $X = \{WA, NT, SA, Q, NSW, V, T\}$)

e possiamo associare ad ogni variabile un dominio $D_i = \{\text{red, blue, green}\}$ dei possibili colori.

I vincoli saranno relazioni binarie tra due coppie di variabili connesse da un arco e possono essere scritti in forma implicita (ad esempio, $WA \neq NT$) oppure in forma esplicita (ovvero, con una tabella in cui si scrivono tutte le possibili coppie di colori diversi).

Tali vincoli si dicono **VINCOLI BINARI** (relazione tra due variabili).

N.B.: tutti i problemi possono essere ridotti a problemi con vincoli binari.

N.B.: lo stato della Tanzania non possiede vincoli con altri stati, perciò può assumere qualsiasi colore. In questo caso si dice **VINCOLO UNARIO**.

Es(criptoaritmetica):

$$\begin{array}{r} S \quad E \quad N \quad D \\ + \\ M \quad O \quad R \quad E \\ \hline M \quad O \quad N \quad E \quad Y \end{array}$$

In questo caso, l'obiettivo è quello di interpretare il significato della somma associando ad ogni lettera il corretto valore numerico per soddisfare l'operazione.

L'insieme delle variabili saranno tutte le lettere da decifrare (ovvero $X = \{S, E, N, D, M, O, R, Y\}$), ognuna delle quali possiede il dominio $D_i = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ (cioè il possibile valore che può assumere).

In questa situazione avremo 3 vincoli:

- corretto svolgimento dell'operazione, ovvero $1000S + 100E + 10N + D + 1000M + 100O + 10R + Y = 10000M + 1000O + 100N + 10E + Y$;
- due lettere diverse non possono assumere lo stesso valore, cioè (in forma implicita) $S \neq E, S \neq N, S \neq D, \dots, R \neq Y$;
- dobbiamo evitare i valori illegali, cioè $S \neq 0$ ed $M \neq 0$.

N.B.: quando un vincolo coinvolge tutte le variabili del problema (come quello del corretto svolgimento della somma) si dice **VINCOLO GLOBALE**.

N.B.: i vincoli che mettono in relazione 3 variabili sono detti **VINCOLI TERNARI**. Tali vincoli comportano degli ipergrafi (quindi, per mantenere il grafo normale è necessario trasformare i vincoli ternari in vincoli binari).

La fase in cui si definiscono gli insiemi X, D e C del problema si dice **FASE DI MODELLAZIONE**.

N.B.: tale fase non è unica, ovvero possiamo modellare il problema in più modi diversi tra loro.

• CONSTRAINT PROPAGATION

L'**inferenza** è un'operazione che permette di modificare i domini delle variabili, eliminando quei valori che non possono più essere scelti a causa di un vincolo.

Quindi, l'inferenza permette la propagazione dei vincoli, ovvero si usano i vincoli per ridurre il numero di valori legali per una variabile, che a sua volta permette di ridurre i valori legali per un'altra variabile, e così via.

In questo modo, quando effettueremo un nuovo assegnamento per una variabile, allora potremo escludere quei valori che non sono più legali (a causa della propagazione dei vincoli) e quindi avremo molte meno scelte da considerare.

Es(sudoku):

1	2	3	4	5	6	7	8	9
2	4			3				6
3	6	3	1	2	7	9		
4								
5	7	3	4	8		9	1	
6								5
7		6	2	4	5	1	8	
8	5		8				6	
9	1	4		9	6	5		

AllDifferent indica tutti i vincoli binari in cui le variabili espresse devono essere tutte diverse tra loro

In questo caso, per risolvere il sudoku, possiamo considerare l'insieme delle variabili $X = \{X_{11}, X_{12}, \dots, X_{19}, X_{21}, \dots, X_{29}, \dots, X_{91}, X_{92}, \dots, X_{99}\}$ (ovvero ogni variabile rappresenta una cella del sudoku (perciò abbiamo 81 variabili)).

Ad ogni variabile viene associato il dominio $D_i = \{1, \dots, 9\}$ dei numeri che può assumere.

I vincoli sono:

- ogni riga deve contenere tutti numeri diversi, cioè $\text{AllDifferent}(X_{11}, \dots, X_{19})$, $\text{AllDifferent}(X_{21}, \dots, X_{29})$, ecc...
- ogni colonna deve contenere tutti numeri diversi, cioè $\text{AllDifferent}(X_{11}, \dots, X_{91})$, $\text{AllDifferent}(X_{21}, \dots, X_{92})$, ecc...
- ogni quadrato deve contenere tutti numeri diversi, cioè $\text{AllDifferent}(X_{11}, \dots, X_{19}, \dots, X_{31}, \dots, X_{39})$, ecc...

N.B.: in totale abbiamo 27 vincoli (9 vincoli sulle righe, 9 sulle colonne e 9 sui quadrati).

In questo caso, possiamo avere delle inferenze sulle variabili, poiché alcune variabili sono già state assegnate e quindi, per soddisfare i vincoli, non possiamo assegnare quegli stessi valori (ad esempio, la variabile X_{11} può assumere solo il valore 1, quindi possiamo restringere il dominio $D_1 = \{1\}$).

Se assegnamo un valore ad una variabile (ad esempio, assegniamo $X_{11}=1$), allora influenzeremo altre variabili che non possono più assumere quel valore. Quindi possiamo nuovamente effettuare delle inferenze (si propagano i vincoli).

Per eliminare correttamente i valori definire le seguenti nozioni:

- **CONSISTENZA DEI NODI** (node consistency): X_i si dice nodo consistente, se tutti i valori di nel dominio D_i soddisfano i vincoli undir.
- **CONSISTENZA DEGLI ARCHI** (arc consistency): X_i si dice arco consistente rispetto ad X_j , se per ogni valore di nel dominio D_i esiste un valore di nel dominio D_j tale che, assegnato ad X_j , permette di soddisfare il vincolo tra X_i ed X_j .

N.B.: l'arc consistency è una nozione orientata, ovvero dire che X_i è AC rispetto ad X_j non implica che X_j sia AC rispetto ad X_i (e viceversa).
ES: Date le variabili X ed Y con $D_x=D_y=\{0, \dots, 9\}$, vogliamo soddisfare il vincolo $Y=X^2$.

In particolare, X non è AC (arc consistent) rispetto ad Y , perché non tutti i valori del dominio D_x possono soddisfare il vincolo (infatti, se ad esempio consideriamo il valore $X=4$, notiamo che non esiste nessun valore nel dominio D_y che possa soddisfare il vincolo, ovvero non esiste il valore $Y=16$).

Perciò, possiamo eliminare tutti quei valori che non soddisfano il vincolo, fino a rendere X AC rispetto ad Y , ovvero $D_x=\{0, 1, 2, 3\}$.

N.B.: tale operazione di rimozione dei valori non legali è detta cross-out.

In questo caso, anche Y non è AC rispetto ad X . Possiamo quindi effettuare il cross-out, ovvero $D_y=\{0, 1, 4, 9\}$.

Possiamo implementare l'algoritmo di arc consistency nel seguente modo:

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
queue ← a queue of arcs, initially all the arcs in csp
while queue is not empty do
     $(X_i, X_j) \leftarrow \text{POP}(queue)$ 
    if REVISE(csp,  $X_i$ ,  $X_j$ ) then
        if size of  $D_i = 0$  then return false
        for each  $X_k$  in  $X_i.\text{NEIGHBORS} - \{X_j\}$  do
            add  $(X_k, X_i)$  to queue
return true

function REVISE(csp,  $X_i$ ,  $X_j$ ) returns true iff we revise the domain of  $X_i$ 
revised ← false
for each  $x$  in  $D_i$  do
    if no value  $y$  in  $D_j$  allows  $(x, y)$  to satisfy the constraint between  $X_i$  and  $X_j$  then
        delete  $x$  from  $D_i$ 
        revised ← true
return revised
```

Si considera il problema CSP con la tripla (X, D, C) e si inserisce in una coda tutti gli archi del problema.

Finché la coda non è vuota, si estrae un arco dalla coda su cui applichiamo la funzione REVISE che controlla se l'arco è AC, altrimenti lo rende AC (ritornando true).

N.B.: se l'arco era già AC, allora la funzione REVISE ritorna false (cioè non ha portato modifiche al dominio).

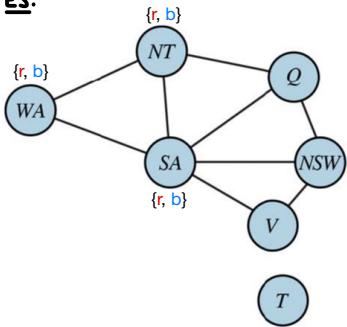
Se, durante la revisione, il dominio D_i è diventato vuoto, allora non possiamo avere soluzioni (problema insoddisfacibile). Altrimenti si considerano i nodi vicini ad X_i e si inserisce i nuovi archi in coda (constraint propagation).

Chiamiamo c il numero di vincoli (ovvero $c=\#\mathcal{C}$) ed d la cardinalità del dominio più grande (ovvero $d=\max|D_i|$). Allora, la funzione REVISE avrà un costo $O(d^3)$, mentre l'algoritmo AC3 avrà un costo $O(cd^3)$.

N.B.: tale algoritmo è finito poiché i domini hanno cardinalità finita, ma non garantisce una soluzione.

- **CONSISTENZA DEI CAMMINI** (path consistency): la coppia (X_i, X_j) si dice cammino consistente rispetto ad X_m , se per ogni assegnamento $X_i=a$ ed $X_j=b$, con $a \in D_i$ ed $b \in D_j$, esiste un assegnamento $X_m=c$, con $c \in D_m$, tale che soddisfi contemporaneamente i vincoli tra X_i ed X_m e i vincoli tra X_m ed X_j .

ES:



Consideriamo il grafo dell'Australia. Vogliamo vedere se l'arco (WA, SA) è path consistent rispetto ad NT (ipotizziamo che i domini siano $D_{WA}=D_{SA}=D_{NT}=\{\text{red, blue}\}$).

Possiamo effettuare due diversi assegnamenti per WA ed SA , ovvero:

- $\{WA=\text{red}, SA=\text{blue}\}$
- $\{WA=\text{blue}, SA=\text{red}\}$

In entrambi i casi non possiamo fare l'assegnamento $NT=\text{red}$ oppure $NT=\text{blue}$ (ovvero (WA, SA) non è PC rispetto ad NT). Perciò, possiamo fare il cross-out dei valori dai domini, ottenendo dei domini vuoti.

Possiamo concludere che il problema non ha soluzioni (non possiamo colorare gli stati con solo due colori).

N.B.: la nozione di path consistency è più forte dell'arc consistency (può trovare soluzioni laddove l'AC è insoddisfacibile).

- **K-CONSISTENCY**: un CSP si dice K-consistency se, per ogni insieme di $K-1$ variabili e per ogni assegnamento consistente per tali variabili, allora possiamo assegnare un valore consistente alla variabile X_K .

N.B.: la node consistency rappresenta la 1-consistency, l'arc consistency rappresenta la 2-consistency e la path consistency rappresenta la 3-consistency.

N.B.: se $K=n$, allora possiamo sicuramente trovare una soluzione (infatti, potremmo associare un valore consistente ad ogni variabile del problema), ma la risoluzione richiederebbe un costo troppo elevato.

• BACKTRACKING SEARCH

L'algoritmo di ricerca **backtracking** è un algoritmo di risoluzione dei problemi di soddisfacimento dei vincoli (CSP) in cui si effettuano delle assegnazioni parziali. In tale algoritmo si sceglie ripetutamente una variabile non assegnata e si provano, a turno, tutti i valori nel dominio di tale variabile, cercando di definire una soluzione tramite una chiamata ricorsiva. Se tale chiamata ha esito positivo, allora viene restituita la soluzione, altrimenti, se fallisce, si ripristina l'assegnamento allo stato precedente e si tenta il valore successivo nel dominio.

N.B: se nessun valore è in grado di definire una soluzione, allora si restituisce un errore.

Oss: la logica di questo algoritmo è come la logica del Sudoku, ovvero si assegna un valore ad una variabile e si guarda se andando avanti si ha una corretta soluzione. Altrimenti si torna indietro e si assegna alla variabile un altro valore.

Possiamo implementare l'algoritmo di ricerca backtracking nel seguente modo:

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, {})

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ← INFERENCE(csp, var, assignment)
            if inferences ≠ failure then
                add inferences to csp
                result ← BACKTRACK(csp, assignment)
                if result ≠ failure then return result
                remove inferences from csp
                remove {var = value} from assignment
    return failure
```

L'efficienza di tale algoritmo si basa su due criteri (euristici):

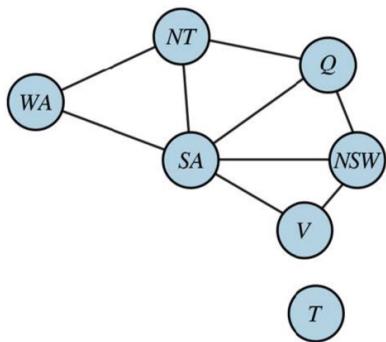
• **ORDERING VARIABLES** (ordinamento delle variabili): dobbiamo definire un modello per decidere l'ordine delle variabili da considerare per effettuare gli assegnamenti nel modo più efficiente possibile (implementato dalla funzione **SELECT_UNASSIGNED_VARIABLE**).

L'idea è quella di eliminare per prima tutte le variabili che possono portare ad un fallimento (in questo modo si evita di espandere sottosalberi che non portano a delle soluzioni).

Una tecnica euristica impiegata per realizzare tale modello è detta **MINIMUM REMAINING VALUES** (MRV), con la quale si sceglie la variabile che contiene il minor numero di valori nel proprio dominio (ovvero, si sceglie la variabile con minore cardinalità del proprio dominio ($\arg\min_i |D_i|$)). In questo modo, se tale variabile dovesse fallire, allora l'algoritmo se ne accorge presto

N.B: nel caso in cui esistono più variabili che possiedono contemporaneamente il minor numero di valori nel proprio dominio (ovvero, nel caso in cui esistono più domini con cardinalità minima ($|\arg\min_i |D_i|| > 1$)), allora si sceglie la variabile che riduce il branching factor sulle scelte future (ovvero, si sceglie la variabile che è coinvolta nel maggior numero di vincoli su altre variabili non assegnate, così da ridurre il numero di possibili scelte future sulle successive variabili). In particolare, si sceglie la variabile con grado maggiore.

ES:



Inizialmente, il dominio di tutte le variabili sarà $D := \{\text{red, blue, green}\}$.

Perciò, ogni variabile avrà cardinalità del dominio minima uguale a 3.

Dobbiamo quindi scegliere la variabile che permette di ridurre il branching factor.

L'idea è quella di scegliere la variabile che possiede il grado massimo del grafo (ovvero, si sceglie il nodo del grafo che possiede il maggior numero di archi incidenti su tale nodo).

In questo caso, la scelta migliore sarà la variabile SA che possiede il grado massimo 5.

In questo modo, se assegnamo un colore ad SA, tutte le altre variabili connesse con SA non potranno scegliere quel colore (perciò abbiamo diminuito il numero di possibili scelte future).

N.B: se più variabili possiedono il grado massimo, allora si sceglie a caso tra queste.

• **ORDERING VALUES** (ordinamento dei valori): una volta che è stata scelta una variabile, dobbiamo definire un modello per decidere come ordinare i valori nel dominio della variabile, così che questi possano essere assegnati nel modo più efficiente per trovare una possibile soluzione (implementato dalla funzione **ORDER_DOMAIN_VALUES**).

L'idea è quella di assegnare per prima i valori che comportano un numero maggiore di soluzioni (in questo modo si ha una maggiore possibilità di trovare una soluzione).

Perciò, non è possibile definire in breve tempo il numero esatto di soluzioni per ogni valore. Quindi dobbiamo effettuare delle approssimazioni. Possiamo utilizzare due tecniche euristiche per effettuare tali approssimazioni:

• **TREE RELAXATION** (rilassamento ad albero): si effettua un'approssimazione del numero di soluzioni per ogni sottoproblema utilizzando i rilassamenti. Il rilassamento è un'operazione che permette di eliminare alcuni vincoli.

Un'applicazione efficace è il rilassamento ad albero in cui si eliminano alcuni archi dell'albero per rimuoverne i cicli.

• **DOMAINS DIMENSION** (dimensione dei domini): si effettua un'approssimazione del numero di soluzioni per ogni sottoproblema mantenendo

grandi le dimensioni dei domini per le variabili non assegnate.

Percio' si definisce una quantita' $\text{Rem}(Y|P)$ (detta remaining) che rappresenta il numero di valori presenti nel dominio D_Y (ovvero la cardinalita' di $D_Y (|D_Y|)$) dopo aver effettuato degli assegnamenti in P e dopo aver effettuato la propagazione dei vincoli.

Quindi, il valore $\hat{a} \in D_X$ da assegnare alla variabile X e' quello che massimizza il prodotto di tutti i remaining di ogni variabile Y che non sono state ancora assegnate, ovvero $\hat{a} = \underset{a \in D_X}{\operatorname{argmax}} \prod_{Y \text{ non assegnata}} \text{Rem}(Y|P \cup \{X=a\})$.

N.B.: un'altra tecnica considera la somma invece del prodotto.

Un ulteriore criterio che permette di aumentare l'efficienza dell'algoritmo e' la scelta della tecnica che implementa l'inferenza, ovvero definire la tecnica per eliminare quei valori che non sono validi e che, se venissero controllati, comporterebbero un grande aumento del tempo di esecuzione. Tale tecnica viene implementata nella funzione INFERENZA.

N.B.: l'algoritmo AC-3 non risulta efficace in questa situazione.

In particolare, possiamo definire due tecniche efficienti per implementare l'inferenza:

• **FORWARD CHECKING (FC):** ogni volta che viene effettuato un assegnamento su una variabile X , allora si cancellano tutti i valori coerenti con il valore assegnato ad X dal dominio di ogni variabile Y non assegnata che e' collegata con un vincolo da X .

In questo modo si garantisce l'arc consistency tra X e tutte le variabili Y non assegnate e collegate ad X .

ES (disposizione di 6 regine):

1	x_1	x_2	x_3	x_4	x_5	x_6
2	Q	x_2	x_3	x_4	x_5	x_6
3	x_1	Q	x_3	x_4	x_5	x_6
4	x_1	x_2	Q	x_4	x_5	x_6
5	Q	x_2	x_3	x_4	x_5	x_6
6	x_1	x_2	x_3	Q	x_5	x_6
	x_1	x_2	x_3	x_4	x_5	x_6

Ipotizziamo di effettuare l'assegnamento $x_1=2$. Allora, possiamo eliminare tale valore dal dominio di tutte le variabili connesse con x_1 (se lo possiedono).

Ipotizziamo ora di effettuare l'assegnamento $x_2=5$ ed eliminiamo nuovamente i valori dalle variabili in conflitto con x_2 .

Assegnamo adesso $x_3=3$ ed eliminiamo nuovamente tutti i valori in conflitto con tale assegnamento.

N.B.: per alcuni problemi, la ricerca sara' piu' efficace quando vengono combinate le tecniche euristiche MRV ed FC. Infatti, il FC puo' essere un modo efficiente per calcolare le informazioni necessarie al MRV per svolgere il suo lavoro.

N.B.: il FC non sempre rileva tutte le inconsistenze, perciò risulta una tecnica meno potente del MAC

• **MAINTAINING ARC CONSISTENCY (MAC):** ogni volta che viene effettuato un assegnamento su una variabile X , allora si cancellano tutti i valori coerenti con il valore assegnato ad X dal dominio di ogni variabile Y non assegnata che e' collegata con un vincolo da X e si controllano ricorsivamente tutte le variabili Y , finche' non e' piu' possibile apportare modifiche ai domini.

ES (disposizione di 6 regine):

1	x_1	x_2	x_3	x_4	x_5	x_6
2	Q	x_2	x_3	x_4	x_5	x_6
3	x_1	Q	x_3	x_4	x_5	x_6
4	x_1	x_2	Q	x_4	x_5	x_6
5	Q	x_2	x_3	x_4	x_5	x_6
6	x_1	x_2	x_3	Q	x_5	x_6
	x_1	x_2	x_3	x_4	x_5	x_6

Ipotizziamo di effettuare l'assegnamento $x_1=2$. Allora, possiamo eliminare tale valore dal dominio di tutte le variabili connesse con x_1 (se lo possiedono). Inoltre, per ognuna di queste variabili controlliamo se esistono delle variabili che possiedono valori inconsistenti con il valore assegnato. In caso positivo si elimina tale valore dal loro dominio.

Analogamente, vediamo cosa accade per l'assegnamento $x_2=5$.

N.B.: il MAC, a differenza di FC, controlla ricorsivamente i domini delle variabili modificate. Perciò risulta essere una tecnica più potente.

TREE-CSP

I CSP sono problemi NP-completi. Infatti, possiamo ridurre un qualunque CSP in un problema SAT.

In particolare, dato un CSP possiamo trasformarlo in un problema SAT (operazione di encoding). Risolvendo tale problema SAT otteniamo una soluzione che deve essere trasformata in una soluzione del CSP (operazione di decoding).



Le operazioni di encoding e decoding possono essere eseguite in tempi polinomiali. Questo dimostra che il CSP considerato è NP-completo.

Esiste però una sottoclassificazione dei CSP, detti **TREE-CSP**, che stanno in P (ovvero possono essere risolti in tempo polinomiale).

Un tree-CSP è un problema senza cicli (aciclico), ovvero è un problema per il quale possiamo definire un albero.

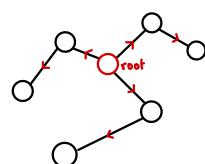
Per tali problemi possiamo realizzare il seguente algoritmo:

```

function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components X, D, C
  n ← number of variables in X
  assignment ← an empty assignment
  root ← any variable in X
  X ← TOPOLOGICALSORT(X, root)
  for j = n down to 2 do
    MAKE-ARC-CONSISTENT(PARENT(Xj), Xj)
    if it cannot be made consistent then return failure
  for i = 1 to n do
    assignment[Xi] ← any consistent value from Di
    if there is no consistent value then return failure
  return assignment
  
```

Date n variabili, si può definire una qualsiasi variabile come radice dell'albero.

Questo comporta un orientamento all'interno dell'albero (TOPOLOGICAL_SORT).



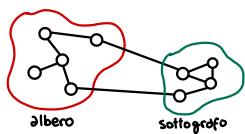
Definito tale ordine topologico sui nodi, si prosegue all'indietro (ci fermiamo prima della radice) per rendere consistenti tutti gli archi padre-figlio.

Una volta resi consistenti tutti gli archi, allora avremo tutti i valori legali in ogni dominio. Perciò l'assegnamento può

essere fatto con qualsiasi valore (poiché ognuno di questi porta ad una soluzione).

N.B.: i TREE-CSP sono difficili da realizzare nella pratica. Ma possiamo utilizzare tale algoritmo come subroutine per implementare alcune tecniche in modo efficiente.

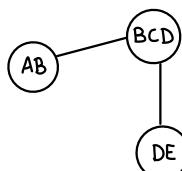
Un'applicazione dei tree-CSP per la risoluzione dei CSP e' detta **CUTSET CONDITIONING** (condizionamento dei tagli).



L'idea e' quella di trasformare il grafo del CSP in un albero, in modo da poter applicare l'algoritmo TREE-CSP-SOLVER su tale albero. Per implementare questa idea possiamo effettuare dei tagli nel grafo (ovvero, si partiziona l'insieme dei nodi), facendo in modo di ottenere un albero ed un sottografo che possieda pochi nodi: (così che si possano trovare tutte le soluzioni del sottografo in un tempo breve).

Perciò, per ogni soluzione del sottografo si effettua una propagazione (per tornare all'albero) e si applica il TREE-CSP-SOLVER.

Un'altra tecnica di risoluzione dei CSP che puo' essere implementata con l'applicazione dei tree-CSP e' detta **VARIABLE CLUSTERING** (raggruppamento di variabili), in cui si trasforma il CSP in un albero i cui nodi sono insiemi (o clusters) di variabili (detti megavariabili).



I domini di tali megavariabili saranno i prodotti cartesiani dei domini di ogni variabile dell'insieme.

Inoltre, dobbiamo aggiornare i vincoli per rendere i compatibili le tuple da assegnare alle megavariabili (ad esempio, se viene assegnato $AB=(1,1)$ allora la variabile B contenuta nella megavariabile BCD deve avere il valore 1).

Tali vincoli sono detti vincoli di consistenza delle variabili.

Una volta definito l'albero si risolvono tutti i sottoproblemi contenuti nelle megavariabili a cui si applica l'algoritmo TREE-CSP-SOLVER. La soluzione ottenuta viene rimappata nelle variabili originali.

Per definire le megavariabili dobbiamo considerare le cricche massimali (maximal cliques).

Def(crica): dato un grafo $G=(V,E)$, allora una crica C e' un sottoinsieme dei nodi di un grafo ($C \subseteq V$) tale che $\forall u, v \in C$ si ha che $(u,v) \in E$ (ovvero per ogni nodo nella crica, esiste un arco nel grafo).

Def(crica massimale): una crica C si dice massimale se \nexists una crica C' tale che $C' \subseteq C$ (cioè, se non esiste un superset di C che e' una crica).

N.B.: definite le megavariabili, allora si collegano con un arco le megavariabili che sono accomunate da una variabile.

A questo punto possiamo ottenere l'albero effettuando l'operazione ^(albero di copertura massimo) **max spanning tree**, ovvero si cerca all'interno del grafo l'albero di costo massimo (dove il costo dell'albero e' dato dalla somma dei costi degli archi che compongono l'albero, ed il costo di un arco e' dato dal numero di nodi in comune tra le due megavariabili). L'operazione di max spanning tree puo' essere implementata con Kruskal o con Counting sort.

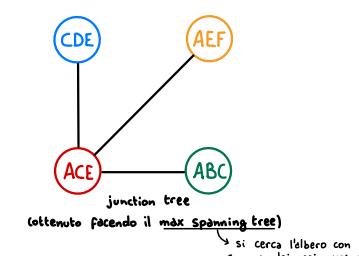
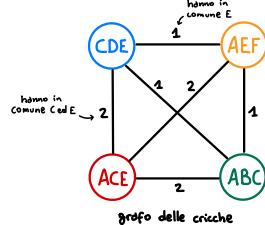
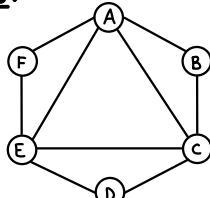
Inoltre, tale albero deve soddisfare una proprietà detta **RUNNING INTERSECTION**.

Def(running intersection): dati due nodi X_i, X_j (clusters) con $Z_{ij} = X_i \cap X_j \neq \emptyset$, allora tutti i nodi nel cammino $X_i \sim X_j$ devono contenere Z_{ij} (ovvero, $Z_{ij} \subseteq Y$ per ogni nodo Y presente nel cammino tra X_i ed X_j).

Gli alberi realizzati in questo modo sono detti **JUNCTION TREE**

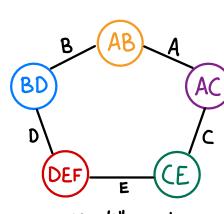
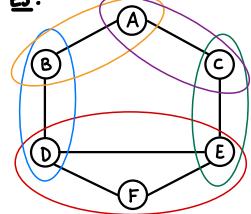
Vediamo come possiamo realizzare un junction tree a partire dal grafo del CSP:

Esempio:

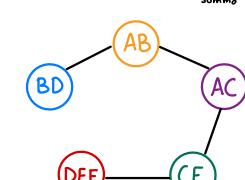


N.B.: il grafo delle cricche si ottiene connettendo i nodi che hanno intersezione non vuota. Possiamo inserire dei pesi sugli archi del grafo delle cricche, dati dalla cardinalità dell'intersezione (ovvero il numero di nodi in comune tra le due cricche).

Esempio:



in questo caso tutti gli archi hanno costo 1 quindi tutti gli alberi interni al grafo hanno lo stesso costo (possiamo eliminare un qualsiasi arco)



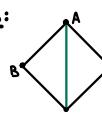
N.B.: in questo caso, però, si verifica un problema nel soddisfacimento dei vincoli, poiché manca la proprietà di running intersection. Infatti, i nodi BD e DEF hanno $Z=0$, ma D non e' sempre presente nel cammino che li unisce (ipotizziamo di fare i seguenti assegnamenti validi: $BD=00$, $AB=10$, $AC=11$, $CE=10$ e $DEF=111$ e notiamo che il vincolo di consistenza su D non e' soddisfatto).

Notiamo che e' possibile ottenere un junction tree, solo quando il grafo e' **TRIANGOLATO** (o anche detto **CORDALE**).

Def(grafo triangolato): un grafo $G=(V,E)$ si dice triangolato se ogni ciclo di lunghezza maggiore o uguale a 4 (≥ 4) ha almeno una corda.

Def(corda in un ciclo): una corda in un ciclo e' un arco non appartenente al ciclo che connette due vertici del ciclo.

Esempio:



Tale grafo e' triangolato. Infatti, gli archi neri definiscono il ciclo ABDC di lunghezza 4, mentre l'arco verde rappresenta la corda (ovvero un arco che non appartiene al ciclo ma connette i vertici A e D del ciclo).

In particolare, se un grafo non è triangolato, possiamo sempre renderlo tale applicando il seguente algoritmo.

Algoritmo di eliminazione delle variabili: Si ordinano i vertici del grafo ($\text{ORDER}(V)$) e per ogni vertice $i = 1, \dots, |V|$ si guardano i suoi vicini.

Se questi non sono connessi, allora vengono connessi eliminando i .

N.B.: tale algoritmo ha una complessità $O(|V|!)$, ma se volessimo ottenere un ordine ottimale dei vertici (in modo da ottenere delle cricche massimali), allora tale algoritmo risulta NP-completo.

In questo modo possiamo ottenere sempre un grafo triangolato, da cui possiamo ricavare il junction tree scegliendo le megavariabili tra le cricche candidate (candidates). Quest'ultime sono ottenute cercando nel grafo i vertici simpliciali.

Def (vertice simpliciale): un vertice v si dice **SIMPPLICIALE** se tutti i suoi vicini inducono un sottografo completo (ovvero tutti i vicini di v sono connessi).

Teorema: un nodo v è simpliciale se e solo se il vertice v ed i suoi vicini formano una cricca.

Teorema: se un grafo G è triangolato, allora contiene almeno 2 vertici simpliciali.

Quindi, assumendo di avere un grafo triangolato (altrimenti si rende tale), possiamo ottenere il junction tree con il seguente algoritmo:

* Si considera l'insieme delle cricche candidate del grafo triangolato vuoto (cioè, $\text{candidates} = \emptyset$);

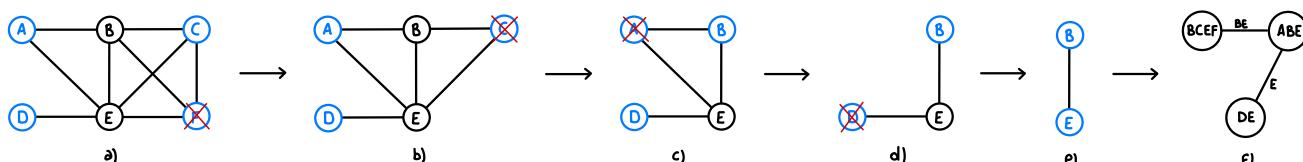
* Si effettua un ciclo per trovare le cricche candidate:

- Si cerca tra tutti i vertici del grafo un vertice simpliciale v e lo si elimina;
- Si aggiunge la famiglia di v (ovvero, v ed i suoi vicini ($\text{family}(v) = \{v\} \cup \text{neighbors}(v)$)) alle cricche candidate;
- Si termina il ciclo quando la famiglia di v è composta da tutti i vertici rimasti nel grafo;

* Si potranno i candidati per cercare la cricca massimale. Ovvero, dati due candidati C ed C' (cioè $C, C' \subseteq \text{candidates}$), allora si rimuove C se $C \subset C'$.

E.S.:

Consideriamo il seguente grafo triangolato e cerchiamo i vertici simpliciali (disegnati in blu nel grafo). Inizialmente abbiamo $\text{candidates} = \emptyset$



a) Il vertice A è simpliciale, perché i vicini B ed E sono connessi. Il vertice B non è simpliciale (infatti, i suoi vicini sono A, C ed E ma i vertici A e C non sono connessi). I vertici C ed F sono simpliciali (poiché i suoi vicini sono connessi). Il vertice E non è simpliciale (poiché A e F , B e D , A e D ed D e F non sono connessi). Il vertice D è simpliciale.

Ipotizzando di eliminare il nodo F , allora si inserisce la cricca $BCEF$ tra i candidati.

b) si trovano gli stessi vertici simpliciali. Ipotizziamo di rimuovere C , allora si inserisce la cricca BCE tra i candidati.

c) il vertice B diventa simpliciale, poiché i suoi vicini adesso sono tutti connessi. Perciò, si rimuove ad esempio A e si inserisce la cricca ABE tra i candidati.

d) si rimuove il vertice simpliciale D e si inserisce la cricca DE tra i candidati.

e) ci fermiamo perché tutti i nodi rimasti nel grafo fanno parte della stessa famiglia.

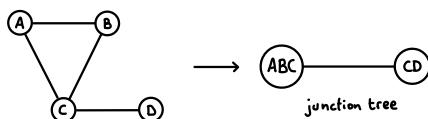
N.B.: al termine del ciclo otteniamo $\text{candidates} = BCEF, BCE, ABE, DE$. Notiamo che la cricca BCE è contenuta in $BCEF$, perciò si effettua la potatura (si rimuove la cricca BCE tra le candidate), ottenendo $\text{candidates} = BCEF, ABE, DE$.

f) per ottenere il junction tree, si connettono le cricche candidate che hanno nodi in comune. Se si ottiene un grafo con dei cicli, allora si estrae il junction tree cercando l'albero di copertura massima (max spanning tree).

N.B.: una volta determinate le megavariabili del junction tree, possiamo definire il dominio di ogni megavariabile come tutte le soluzioni del sottoproblema CSP indotto da quelle variabili (ovvero, il prodotto cartesiano dei domini di ogni variabile della cricca).

Inoltre, dobbiamo aggiungere i vincoli di consistenza delle variabili, ovvero dobbiamo far coincidere le variabili originali del problema.

E.S. ($D_A = D_B = D_C = D_D = \{r, g, b\}$):



Vogliamo colorare i vertici vicini con colori diversi. Vediamo i domini delle cricche:

$$D(ABC) = \{(r, g, b), (r, b, g), (b, g, r), (b, r, g), (g, r, b), (g, b, r)\}$$

$$D(CD) = \{(r, b), (r, g), (g, r), (g, b), (b, r), (b, g)\}$$

Inoltre, l'assegnamento alla variabile C della cricca ABC deve essere lo stesso per la variabile C di CE (vincolo di consistenza della variabile C).

JOBSHOP SCHEDULING (esempio)

Dobbiamo decidere il modo per eseguire dei compiti (jobs) su due macchine diverse. Ad esempio, ogni compito rappresenta un oggetto da produrre che deve essere tagliato e successivamente colorato. Perciò, la macchina1 si occupa di tagliare l'oggetto e la macchina2 si occupa di colorare il pezzo.

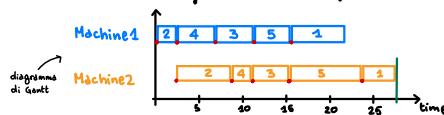
Ipotizziamo che il primo oggetto sia lento da tagliare e veloce da verniciare. Al contrario, il secondo oggetto

Machin1	Machin2
1	1
2	2
3	3
4	4
5	5

richiede poco tempo per essere tagliato, ma impiega molto tempo per essere verniciato (e così via).

Percio; nel disegno possiamo vedere il tempo richiesto da ogni job per essere completato sulla rispettiva macchina.

L'obiettivo e' quello di completare tutti gli oggetti (rispettando i vincoli (ovvero prima dobbiamo tagliare e poi verniciare)) nel piu breve



tempo possibile. Possiamo rappresentare questo schema attraverso il diagramma di Gantt, nel quale si definiscono i tempi in cui si eseguono i job nelle macchine. Percio; per ogni job viene definito un tempo iniziale (in cui entra nella macchina) ed un tempo finale (in cui esce dalla macchina).

Vogliamo formalizzare questo problema. Innanzitutto, definiamo come variabili i tempi di inizio di ogni job su una macchina (in rosso) ed il tempo totale per terminare l'esecuzione di tutti i job su tutte le macchine (in verde).

Successivamente si definiscono i vincoli, cioe' che le macchine operino nella sequenza corretta e che completi ogni job entro il tempo richiesto (per evitare la situazione in cui si devono eseguire contemporaneamente piu job sulla stessa macchina).

Minimizzando questo problema possiamo ottenere la migliore soluzione.

• LOGICA COMPUTAZIONALE

L'obiettivo e' quello di formalizzare e descrivere la conoscenza attraverso delle formule.

Una formula e' una stringa composta da caratteri e definita da una sintassi (permette di definire una formula ben formata) e da una semantica (permette di interpretare i caratteri della formula).

Nella logica possiamo definire il mondo come un'entita' astratta che contiene elementi come oggetti, funzioni, relazioni e simboli (esistenti per quella logica). Tali elementi costituiscono l'ontologia del mondo.

Percio; la semantica descrive una funzione (detta interpretazione) che mappa i caratteri di una formula in elementi del mondo.

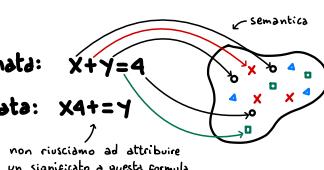
In generale, una particolare logica possiede un impegno ontologico, che rappresenta il codominio della funzione di interpretazione (ovvero gli elementi del mondo), ed un impegno epistemologico, che rappresenta le condizioni ed i metodi per avere una conoscenza (ad esempio, definire se una formula e' vera o falsa, oppure definire un grado di verita' ad una formula (detta fuzzy logic), oppure definire una probabilita' che si verifichi una formula (detta believe logic)).

ES (logica aritmetica):

Vediamo una formula ben formata: $x+y=4$

Vediamo una formula mal formata: $x4+y$

sintassi



mondo aritmetico, contiene: - oggetti: (variabili, numeri, ...)

- **funzioni** ($+, -, \cdot, \dots$)
- **relazioni** ($\leq, >, \leq, \geq, =, \dots$)
- **simboli** ($(,)$, \forall, \exists, \dots)

} ontologica

↓ possiamo aggiungere
e rimuovere formule

In generale gli agenti utilizzano i concetti della logica per agire e definire le azioni da eseguire. In particolare, gli agenti sfruttano un sistema detto KNOWLEDGE BASE (abbreviato con KB) che rappresenta una collezione dinamica di formule ben formate (ovvero la cui sintassi e' corretta) ed implicitamente congiunte (ovvero $KB = \alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n$).

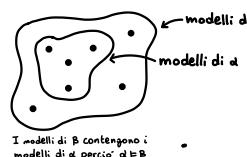
Tale KB possiede 2 operazioni per lavorare sulla collezione di formule:

- Tell(KB, F): aggiunge la formula F alla KB.
- Ask(KB, F): interroga la KB sulla formula F (ovvero verifica se la formula F e' conseguenza logica della KB).

N.B.: la conseguenza logica e' una relazione binaria tra le formule detta ENTAILMENT (si indica con \models). Ovvero $\alpha \models \beta$ significa che la formula α ^(entails) comporta la formula β (cioe' β e' conseguenza logica di α e quindi β risulta vera per ogni interpretazione che rende α vera). Ad esempio, la formula matematica $xy=0$ e' conseguenza logica della formula matematica $x=0$ (perciò $x=0 \models xy=0$).

N.B.: l'implicazione (\Rightarrow) e' diversa da entailment (\models), poiche' stanno su piani logici differenti.

Definiamo il MODELLO di una formula α come l'interpretazione che rende tale formula α vera. Ad esempio, se consideriamo la formula



aritmetica $x+y=4$, allora l'interpretazione che assegna il valore 5 ad x ed il valore 1 ad y non e' un modello perche' non rende tale formula vera, mentre l'interpretazione che assegna il valore 3 ad x ed il valore 1 ad y e' un modello (poiche' $3+1=4$ e' vero).

Quindi, possiamo dire che $\alpha \models \beta$ se β e' vera per tutti i modelli di α (ovvero β e' vera se anche α e' vera).

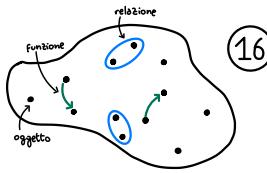
Quindi, un algoritmo di logica computazionale ha il compito di verificare la relazione di entailment tra due formule.

Le logiche che studieremo sono dette.

• Logica Proposizionale (Propositional Logic): il mondo della logica proposizionale contiene solamente fatti. Vediamo la sintassi:

- definiamo le formule atomiche utilizzando simboli proposizionali (ad esempio A, B, C, P, Q, ...).
- definiamo le formule complesse utilizzando formule atomiche e connettivi logici ($\neg, \wedge, \vee, \Rightarrow, \Leftarrow, (), \dots$) (ad esempio $(P \wedge R) \Rightarrow \neg Q$).

• Logica dei Predicati o Logica di Primo Ordine (Predicate Logic o First Order Logic): il mondo della logica di primo ordine contiene oggetti, funzioni e relazioni. In particolare:



- gli oggetti sono gli elementi del mondo (sono infiniti). Un oggetto si rappresenta con un punto.

- le funzioni sono le proprietà di un oggetto. Una funzione si rappresenta con una freccia tra due oggetti.

- le relazioni (o predicati) sono legami binari tra oggetti. Una relazione si rappresenta con una tupla di oggetti.

La logica del primo ordine è semidecidibile poiché non sempre possiamo determinare la relazione di entailment (questo perché gli oggetti sono infiniti).

• LOGICA PROPOZIONALE

Abbiamo visto che, dato un insieme di formule (detto KB), possiamo definire la relazione binaria $KB \models \beta$ (detta entailment) in cui la formula β risulta vera in tutte le interpretazioni in cui risulta vera la KB.

Definiamo un'ulteriore relazione binaria tra formule, detta **DERIVATION** (si indica con \vdash , ovvero $KB \vdash \beta$, in cui la formula β può essere derivata programmaticamente dalla KB (cioè $KB \vdash \beta$ se esiste un programma (o algoritmo) che può derivare β a partire dalla KB).

Un programma si dice **SOUNDNESS** se $KB \vdash \beta$ implica $KB \models \beta$ (cioè se ogni volta che una formula β viene derivata dalla KB, allora tale formula è anche conseguenza logica (entailment) della KB).

Inversamente, un programma si dice **COMPLETENESS** se $KB \models \beta$ implica $KB \vdash \beta$ (cioè se una formula β è conseguenza logica della KB, allora il programma riesce a derivarla).

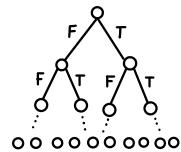
Teorema: in logica proposizionale, l'entailment è decidibile.

Dim:

Si enumerano tutte le possibili interpretazioni. Ovvero, consideriamo n simboli proposizionali, ognuno dei quali può assumere 2 possibili interpretazioni (vero o falso).

- Se $n=1$, allora abbiamo solo 2 possibili interpretazioni (un simbolo proposizionale può essere vero o falso).
- Se $n=2$, allora abbiamo 4 possibili interpretazioni.
- Se n generico, allora abbiamo 2^n possibili interpretazioni, che risulta essere un numero finito.

CVD



N.B.: la decidibilità dell'entailment può essere verificata in un tempo $O(2^n)$. Per migliorare questa situazione, possiamo ridurre lo studio della decidibilità di entailment ad un problema di soddisfacibilità.

Teorema: $KB \models \beta$ se e solo se $KB \Rightarrow \beta$ è vera in tutte le interpretazioni.

Innanzitutto, diamo alcune nozioni tecniche:

Def (equivalenza): due formule α e β si dicono **LOGICAMENTE EQUIVALENTI**, ovvero $\alpha \equiv \beta$, se l'insieme delle interpretazioni che rendono vero α coincide con l'insieme delle interpretazioni che rendono vero β .

N.B.: $KB \Rightarrow \beta$ è logicamente equivalente ad $\beta \vee \neg KB$ (cioè $KB \Rightarrow \beta \equiv \beta \vee \neg KB$)

Def (formula valida): una formula β si dice **VALIDA** se risulta vera per ogni interpretazione.

Def (formula soddisfacibile): una formula β si dice **SODDISFACIBILE** se risulta vera per almeno una interpretazione.

Def (formula insoddisfacibile): una formula β si dice **INSODDISFACIBILE** se risulta falsa per ogni interpretazione.

N.B.: una formula β è valida se e solo se $\neg \beta$ è insoddisfacibile.

Def (letterale): si dice **LETTERALE** un simbolo proposizionale oppure un simbolo proposizionale negato.

Def (clausola): si dice **CLAUSOLA** (clause) una disgiunzione di letterali (o)

Def (forma normale congiuntiva): una formula si dice in **FORMA NORMALE CONGIUNTIVA** (conjunctive normal form) quando viene scritta attraverso una congiunzione di clausole.

N.B.: una clausola risulta vera se esiste almeno un letterale vero, mentre una formula in forma normale congiuntiva risulta vera solo se tutte le clausole risultano vere.

ES (forma normale congiuntiva):

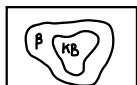
$$(A \vee \neg B \vee \neg C) \wedge (\neg A \vee B) \wedge (C \vee \neg D)$$

clausola letterale congiunzione disgiunzione

N.B.: possiamo riscrivere tale formula come set di clausole dove ogni clausola è un set di letterali (cioè $\{[A, \neg B, \neg C], [\neg A, B], [C, \neg D]\}$).

Alternativamente, possiamo indicare ogni letterale con un numero che viene negato se il letterale è negato (cioè $\{[1, -2, -3], [-1, 2], [3, -4]\}$), dove abbiamo indicato $A=1, B=2, C=3$ e $D=4$).

Vediamo come possiamo trasformare il problema della decidibilità di entailment in un problema di soddisfacibilità.



Ovvero, vogliamo dimostrare che data una qualunque interpretazione che rende vera la KB, allora questa rende vera anche β (significato di $KB \models \beta$). Perciò, dobbiamo realizzare una singola formula la cui soddisfacibilità verifica l'entailment.

L'idea è quella di seguire la tecnica di dimostrazione per refutazione (ovvero dimostrazione per assurdo). In particolare, vogliamo ottenere una formula insoddisfacibile per il teorema "KB $\models \beta$ se e solo se $KB \Rightarrow \beta$ " (equivalente a "KB $\models \beta$ sse $\beta \vee \neg KB$ è soddisfacibile").

Ovvero, vogliamo dimostrare per assurdo che " $KB \wedge \neg \beta$ è insoddisfacibile se e solo se $KB \models \beta$ ".

In questo modo basta trovare una singola formula che renda vero " $KB \wedge \neg \beta$ " e' insoddisfacibile se e solo se $KB \models \beta$ " per dimostrare che il teorema " $KB \models \beta$ se e solo se $KB \Rightarrow \beta$ " e' falso.

Vogliamo quindi cercare un assegnamento che renda vera la formula. Questa implementazione puo' essere tradotta in un problema di soddisfacimento dei vincoli, in cui le variabili sono i simboli proposizionali ed i vincoli sono le clausole.

Vediamo alcuni algoritmi che permettono di risolvere questo problema.

- **MODEL CHECKING** (oppure Davis-Putnam): l'algoritmo di model checking segue l'idea del backtracking, facendo degli accorgimenti sui possibili assegnamenti da studiare per minimizzare il costo della ricerca della soluzione.

Questo algoritmo permette di verificare la soddisfabilita' di una formula.

A volte e' possibile determinare la veridicità di una formula senza doverne studiare necessariamente tutte le possibili soluzioni, riducendo in modo significativo lo spazio di ricerca.

In particolare si implementano 3 heuristiche:

- 1) terminazione prematura: si verifica una terminazione prematura se troviamo un'assegnazione che renda la formula vera, oppure un'assegnazione che renda almeno una clausola falsa (ad esempio, se consideriamo la formula in CNF $\{\{A, C\}, \{A, B\}\}$ e verifichiamo che il valore di A sia vero, allora tale formula risulterà vera indipendentemente dai valori assegnati a B e C (allo stesso modo, se A e' falso e B e' falso, allora la formula sarà falsa per ogni valore assegnato a C)).

- 2) simboli puri: un simbolo proposizionale si dice puro se compare sempre come letterale positivo oppure come letterale negativo.

In questo caso, se la formula ammette un'interpretazione che la rende vera (ovvero se una formula e' soddisfacibile), allora esiste un'interpretazione in cui ogni simbolo puro assume il valore necessario a rendere vere tutte le clausole in cui compare (ad esempio, data la formula $\{\{A, \neg B\}, \{\neg B, \neg C\}, \{C, A\}\}$ dove A e B sono i simboli puri (poiché A appare sempre come letterale positivo e B appare sempre come letterale negativo) abbiamo che, se tale formula e' soddisfacibile, allora esiste un modello in cui A e' vera (poiché compare sempre come letterale positivo) e B e' falsa (poiché compare sempre come letterale negativo)).

- 3) clausola unitaria (unit clause): una clausola si dice unitaria se ha un singolo letterale non assegnato. In questo caso effettuiamo l'assegnamento che rende tale clausola vera (ad esempio, se consideriamo la formula $\{\{\neg B, \neg C\}, \{C\}\}$ allora, assegnando a C il valore vero (poiché la seconda clausola e' unitaria e C e' un letterale positivo), otteniamo che la prima clausola diventa unitaria (poiché B non e' stato ancora assegnato) ed assegniamo a B il valore falso (poiché dobbiamo rendere la prima clausola vera e B e' un letterale negativo)).

Vediamo come possiamo implementare tale algoritmo:

```
function DPLL-SATISFIABLE?(s) returns true or false
inputs: s, a sentence in propositional logic
clauses ← the set of clauses in the CNF representation of s
symbols ← a list of the proposition symbols in s (symbols in s not yet assigned)
return DPLL(clauses, symbols, {})

function DPLL(clauses, symbols, model) returns true or false
if every clause in clauses is true in model then return true
if some clause in clauses is false in model then return false
P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
if P is non-null then return DPLL(clauses, symbols - P, model ∪ {P=value})
P, value ← FIND-UNIT-CLAUSE(clauses, model)
if P is non-null then return DPLL(clauses, symbols - P, model ∪ {P=value})
P ← FIRST(symbols); rest ← REST(symbols)
return DPLL(clauses, rest, model ∪ {P=true}) or
DPLL(clauses, rest, model ∪ {P=false}))
```

- **STOCASTIC LOCAL SEARCH**: sono algoritmi locali e randomizzati che riprendono l'idea degli algoritmi di ricerca locali.

L'idea e' quella di massimizzare il numero di clausole soddisfatte (in questo modo, otteniamo un assegnamento che rende vere tutte le clausole e quindi che soddisfa la formula). Un semplice algoritmo che implementa questa idea e' detto **WALKSAT** e riprende il principio dell'algoritmo hill-climbing. In particolare, ipotizziamo che data una particolare assegnazione si ottengono k clausole insoddisfatte. Allora si sceglie una clausola insoddisfatta e si inverte il valore assegnato ad un simbolo della clausola. In questo modo, se otteniamo un numero minore di clausole insoddisfatte, allora abbiamo effettuato una buona assegnazione. In particolare, la scelta del simbolo da invertire viene effettuata in modo casuale.

Vediamo come possiamo implementare tale algoritmo:

```
function WALKSAT(clauses, p, max-flips) returns a satisfying model or failure
inputs: clauses, a set of clauses in propositional logic
p, the probability of choosing to do a "random walk" move, typically around 0.5
max-flips, number of value flips allowed before giving up

model ← a random assignment of true/false to the symbols in clauses
for each i = 1 to max-flips do
    if model satisfies clauses then return model
    clause ← a randomly selected clause from clauses that is false in model
    if RANDOM(0, 1) ≤ p then
        flip the value in model of a randomly selected symbol from clause
    else flip whichever symbol in clause maximizes the number of satisfied clauses
return failure
```

Quando, tale algoritmo restituisce un modello allora possiamo concludere che la formula in input è soddisfacibile.

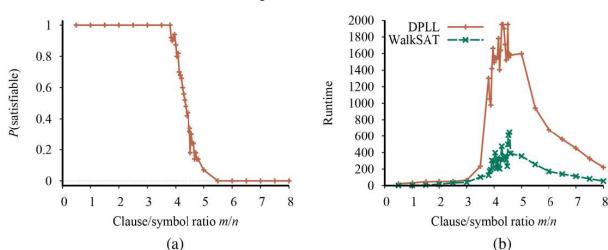
Il problema si verifica quando, invece, viene restituito un fallimento, poiché in questo caso non possiamo affermare che la formula è insoddisfacibile (infatti, si ha un fallimento se la formula è insoddisfacibile oppure se non è stato in grado di calcolarne la soddisfacibilità nel tempo a disposizione). Perciò non può essere usato per il problema dell'entailment, poiché vogliamo verificare l'insoddisfacibilità della formula $\text{KB} \wedge \neg B$.

Questo algoritmo è caratterizzato dal fenomeno della transizione di fase, ovvero esiste un valore critico (dato dal rapporto tra il numero di clausole ed il numero di simboli) oltre il quale la probabilità di avere formule soddisfacibili diventa improvvisamente nullo ed il costo computazionale esplode.

In particolare, dati n simboli proposizionali (quindi 2^n possibili letterali) ed m clausole, ognuna di lunghezza K , allora generiamo una formula scegliendo casualmente K letterali (che non sono già stati scelti) tra i 2^n possibili.

N.B.: un simbolo non può apparire più volte in una clausola ed una clausola non può apparire più volte in una formula.

Indichiamo tale formula generata casualmente con la notazione $\text{CNF}_K(n,m)$, e misuriamone la probabilità di soddisfacibilità:



Vediamo che (se $K=3$) per un basso valore del rapporto $\frac{m}{n}$ otteniamo una probabilità di soddisfacibilità vicino ad 1, mentre per un alto valore del rapporto $\frac{m}{n}$ otteniamo una probabilità di soddisfacibilità vicino a 0.

In particolare, esiste un punto (vicino ad $\frac{m}{n}=4,3$), detto transizione di fase, in cui si ha un brusco passaggio da avere una probabilità di soddisfacimento elevata ad una probabilità quasi nulla.

N.B.: 2-CNF (ovvero si considerano tutte le clausole di lunghezza 2) può essere risolto in tempo polinomiale.

• **THEOREM PROVING**: l'idea è quella di effettuare una serie di passaggi logici che permettono di verificare la veridicità di una formula.

Vogliamo combinare le formule contenute nella KB (attraverso delle regole) per ottenere nuove formule e verificare se soddisfano il problema.

Quindi, si applicano una serie di regole generiche, dette **REGOLE D'INFERENZA**, che permettono di modificare la KB a partire da formule presenti nella KB. Vediamone alcune:

- Modus Ponens: $\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$ (si legge come "se $(\alpha \Rightarrow \beta) \in \text{KB}$ ed $\alpha \in \text{KB}$, allora $\text{KB} \wedge \beta \equiv \text{KB}$ ").

Ovvero, se nella KB sono presenti le formule $\alpha \Rightarrow \beta$ ed α , allora possiamo aggiungere la formula β alla KB senza modificarne le interpretazioni (ovvero si ottiene una KB logicamente equivalente a quella di partenza).

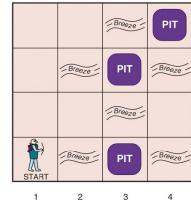
- And elimination: $\frac{\alpha \wedge \beta}{\alpha}, \frac{\alpha \wedge \beta}{\beta}$ (si legge come "se $(\alpha \wedge \beta) \in \text{KB}$, allora $\text{KB} \wedge \alpha \equiv \text{KB}$ (oppure $\text{KB} \wedge \beta \equiv \text{KB}$)").

Ovvero, se nella KB c'è presente la formula $\alpha \wedge \beta$, allora possiamo aggiungere la formula α (o la formula β) alla KB ottenendo una nuova KB logicamente equivalente a quella di partenza).

- Biconditional elimination: $\frac{\alpha \Leftrightarrow \beta}{\alpha \Rightarrow \beta}$ (che possiamo scrivere anche $\frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Rightarrow \beta}$).

- Contrapositione: $\frac{\alpha \Rightarrow \beta}{\neg \alpha \Rightarrow \neg \beta}$

ES (Wumpus world):



Consideriamo un mondo in cui un agente (inizialmente localizzato nella cella $(1,1)$) può muoversi in orizzontale o in verticale (regole del mondo). In alcune celle sono presenti delle buche su cui l'agente può cadere, facendolo morire.

La particolarità è che in tutte le celle adiacenti (sopra, sotto, sinistra e destra) ad una buca c'è presente una brezza di vento. L'agente possiede un sensore che gli permette di individuare le brezze evitando così le buche.

Utilizziamo dei simboli proposizionali per indicare se in una cella c'è presente o non c'è presente un elemento (ad esempio P_{11} è il simbolo che specifica se nella cella $(1,1)$ c'è presente una buca). Quindi, definiamo 16 simboli per le buche e 16 per le brezze (una per cella).

Innanzitutto, istruiamo la KB con le regole del gioco (ad esempio in una cella c'è presente una brezza se e solo se c'è presente una buca nelle celle adiacenti) definite da un simbolo proposizionale R_i .

Successivamente, ad ogni mossa dell'agente informiamo la KB (attraverso l'istruzione tell) riguardo alle percezioni ricevute (ad esempio, durante la prima mossa si visita la cella $(1,1)$ e si informa la KB riguardo al fatto che in quella cella non c'è presente una buca ($\text{tell}(\text{KB}, \neg P_{11})$) e che non c'è presente una brezza ($\text{tell}(\text{KB}, \neg B_{11})$).

Vediamo in particolare come viene modificata la KB, ipotizzando che l'agente abbia visitato la cella $(2,1)$ ($\text{tell}(\text{KB}, B_{21})$):

• $R_1: \neg P_{11}$ (nella cella $(1,1)$ non c'è presente una buca).

• $R_2: B_{11} \Leftrightarrow (P_{12} \vee P_{21})$ (nella cella $(1,1)$ c'è una brezza se e solo se nella cella $(1,2)$ o nella cella $(2,1)$ c'è una buca).

• $R_3: B_{21} \Leftrightarrow (P_{11} \vee P_{22} \vee P_{31})$

• $R_4: \neg B_{11}$ (nella cella $(1,1)$ non c'è presente una brezza).

• $R_5: B_{21}$

A questo punto possiamo fare delle richieste alla KB (utilizzando l'istruzione ask) riguardo le celle adiacenti all'agente, per decidere come muoversi senza cadere in una buca (ad esempio ask(KB, $\neg P_{12}$) verifica se nella cella (1,2) non c'è presente una buca).

Quindi, dobbiamo creare nuove regole per dimostrare che nella cella (1,2) non c'è una buca, a partire dalle regole già presenti nella KB:

- R₆: $(B_{12} \Rightarrow (P_{12} \vee P_{21})) \wedge ((P_{12} \vee P_{21}) \Rightarrow B_{12})$ (biconditional elimination su R₂)
- R₇: $(P_{12} \vee P_{21}) \Rightarrow B_{12}$ (and elimination su R₆)
- R₈: $\neg B_{11} \Rightarrow (P_{12} \vee P_{21})$ (contrapposizione su R₇)
- R₉: $\neg (P_{12} \vee P_{21})$ (modus ponens su R₄, R₈)
- R₁₀: $\neg P_{12} \wedge \neg P_{21}$ (De Morgan su R₉)
- Goal: $\neg P_{12}$ (and elimination su R₁₀)

In particolare, esiste una regola d'inferenza universale, detta **RISOLUZIONE**. Vediamo come si rappresenta:

- Caso semplice: $\frac{A \vee B, \neg A}{B}$

- Caso generale: $\frac{\Phi, \Psi}{(\Phi - \{X\}) \cup (\Psi - \{X\})}$, dove Φ e Ψ sono due clausole ed X è un letterale tale che $X \in \Phi$ ed $\neg X \in \Psi$.

Ad esempio $\frac{AvBvC, \neg AvDvE}{BvCvDvE}$ (ovvero possiamo elidere i letterali complementari)

N.B.: la risoluzione è sufficiente per risolvere il problema di soddisficiabilità di una formula.

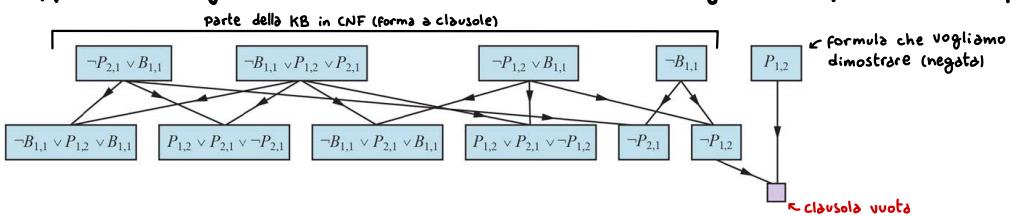
N.B.: la risoluzione non è generativamente completa, ovvero $A \wedge B \models A \wedge B$ non è soddisfacibile (poiché non abbiamo letterali complementari).

Vediamo un algoritmo per implementare la risoluzione:

```
function PL-RESOLUTION(KB, α) returns true or false
    inputs: KB, the knowledge base, a sentence in propositional logic
            α, the query, a sentence in propositional logic

    clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
    new ← {}
    while true do
        for each pair of clauses  $C_i, C_j$  in clauses do
            resolvents ← PL-RESOLVE( $C_i, C_j$ )
            if resolvents contains the empty clause then return true
            new ← new ∪ resolvents
        if new ⊆ clauses then return false
        clauses ← clauses ∪ new
```

Trasformiamo le regole del precedente esempio nella forma a clausole (ovvero si trasformano gli operatori in disgiunzioni logiche) ed applichiamo l'algoritmo della risoluzione. Notiamo che l'algoritmo confronta tutte le possibili coppie di clausole (in figura viene mostrata solo una parte della procedura per semplicità).



Perciò, si cercano i letterali complementari tra le clausole confrontate e si generano nuove clausole che vengono aggiunte

della KB. L'algoritmo termina quando viene generata una clausola vuota, poiché in questo caso abbiamo ottenuto una contraddizione (nell'esempio abbiamo $\{\{P_{1,2}\}, \{\neg P_{2,1}\}\}$ che non può essere risolto in nessun caso e quindi è insoddisfacibile) e quindi si ottiene una dimostrazione per assurdo (infatti $KB \models \alpha$ se e solo se $KB \wedge \neg \alpha$ è insoddisfacibile).

Def (chiusura): dato S un insieme di clausole, allora indichiamo con $RC(S)$ la **CHIUSURA** di S rispetto alla risoluzione. Tale $RC(S)$ denota l'insieme di clausole ottenute applicando ripetutamente la risoluzione.

Teorema (Ground Resolution): sia S un insieme di clausole ed $\{\}$ l'insieme vuoto. Se S è insoddisfacibile, allora $\{\} \in RC(S)$.

Dim:

Si dimostra per contrapposizione. Ovvero, se $RC(S)$ non contiene la clausola vuota, allora S è soddisfacibile.

1) Indichiamo con P_1, P_2, \dots, P_n i simboli proposizionali in S e costruiamo un modello A di S come segue:

Per $i = 1, \dots, n$:

Se $RC(S)$ contiene una clausola che possiede il letterale $\neg P_i$ ed inoltre tutti gli altri letterali della clausola sono falsi nell'assegnamento scelto per $P_1 \dots P_{i-1}$:

Allora $P_i \leftarrow \text{False}$ (ovvero assegna falso a P_i nel modello A).

Altrimenti: $P_i \leftarrow \text{True}$ (ovvero assegna vero a P_i nel modello A).

2) Dimostriamo che A è un modello di S . allora per qualche

Ipotizziamo, per assurdo, che A non è un modello di S . Quindi, esiste un indice i tale che per qualche assegnazione alcune clausole di S diventano false (i indica il primo indice per cui si verifica questa situazione).

Allora le clausole possono essere della forma (F, F, \dots, F, P_i) oppure $(F, F, \dots, F, \neg P_i)$.

(letterali falsi prima di P_i ; altrimenti la clausola sarebbe vera)

In particolare, devono esistere entrambe le clausole contemporaneamente (altrimenti potremmo assegnare un valore corretto al simbolo proposizionale P_i per rendere la clausola vera).

Ma se esistono entrambe le clausole, allora abbiamo che P_i e $\neg P_i$ sono letterali complementari e quindi i precedenti letterali (tutti falsi) appartengono ad $RC(S)$ (ovvero potevano essere eliminati con la risoluzione).

Questo comporta che i non è il primo indice per cui alcune clausole diventano false (assurdo poiché abbiamo ipotizzato il contrario). In particolare non esiste nessun indice i per cui accade questa situazione e quindi non è possibile creare una clausola falsa. Perciò A è un modello.

CVD

N.B.: questo teorema garantisce il funzionamento dell'algoritmo della risoluzione.

Poiché, però, la logica ha una espressività elevata, allora si ha un alto costo computazionale. Quindi, possiamo semplificare la ricerca sfruttando due particolari classi di clausole:

- clausole definite: sono clausole in cui c'è presente esattamente un letterale positivo.

- clausole di Horn: sono clausole in cui c'è presente al più un letterale positivo

Es (clausola definita):

$$H \vee \neg B_1 \vee \neg B_2 \equiv H \vee \neg(B_1 \wedge B_2) \equiv \overbrace{B_1 \wedge B_2}^{\text{body}} \Rightarrow \overbrace{H}^{\text{head}}$$

Che in programmazione logica possiamo scrivere $H \leftarrow B_1, B_2$

Possiamo utilizzare queste 2 classi di clausole per semplificare l'algoritmo di ricerca della soddisficiabilità di una formula.

Esistono 2 diverse implementazioni che utilizzano clausole definite:

- Forward Chaining: si parte dai fatti (cioè dai dati disponibili) a cui si applicano regole di inferenza per ottenere nuovi fatti fino al raggiungimento del goal.
- Backward Chaining: si ragiona in modo inverso, ovvero si parte dal goal per ottenere i dati a disposizione. Perciò, si parte dalle conseguenze delle regole d'inferenza e si verifica la veridicità delle loro premesse, fino a che non si ottiene un fatto (oppure un insieme di fatti) che rendono vere le premesse. Ad esempio, indichiamo il goal con Q ed i fatti con A e B . Allora vediamo come ottenere Q dalla KB (cioè $KB \models Q$) in modo ricorsivo:

1. $Q \Leftarrow P$ (per dimostrare Q è necessario dimostrare P)
2. $P \Leftarrow L \wedge M$ (per dimostrare P è necessario dimostrare L ed M)
3. $M \Leftarrow L \wedge B$ (per dimostrare M è necessario dimostrare L ed B)
4. $L \Leftarrow A \wedge P$ (per dimostrare L è necessario dimostrare A ed P)
5. $L \Leftarrow A \wedge B$ (in alternativa, per dimostrare L è necessario dimostrare A ed M)
6. A (A è il fatto)
7. B (B è il fatto)

• MODELLI PROBABILISTICI

Non sempre è possibile utilizzare la logica per rappresentare domini complessi (come quelli reali).

Es (malattia della mucca pazza):

Vogliamo studiare il seguente sistema (che rappresenta un determinato paziente):

- Malattia: Creutzfeld-Jakob (indicato con CK);
 - Causa: ingestione di un hamburger (indicato con H);
 - Sintomo: perdita della memoria (indicato con M);
- Perciò CK, H ed M rappresentano i fatti proposizionali.

Traduciamo in logica

Il problema che si verifica è che non possiamo tradurre in logica la regola che permette di definire se un paziente è affetto dalla malattia (la regola è che le cause (mangiare l'hamburger) comportano la malattia (mucca pazza) che provoca i sintomi (perdita di memoria), ovvero $H \rightarrow CK \rightarrow M$).

Infatti:

- $M \not\Rightarrow CK$ tale regola non è valida in generale, perché osservare il sintomo non garantisce di rilevare la malattia
- N.B.: per rendere vera tale regola dovremmo scrivere una relazione complessa che identifica tutte le malattie che derivano da quel particolare sintomo, cioè $M \Rightarrow CK \vee \text{Alzheimere} \vee \dots$

(ovvero, non è detto che la perdita di memoria comporta con certezza la malattia della mucca pazza)

- $CK \not\Rightarrow M$ anche in questo caso tale relazione non è valida in generale, poiché non sempre il sintomo si verifica subito.

Possiamo risolvere questo problema di rappresentazione del dominio utilizzando un approccio probabilistico. In particolare, si utilizza questo approccio quando siamo di fronte a **PROBLEMI DI INCERTEZZE** (come nell'esempio).

In particolare è possibile utilizzare la probabilità per definire una misura (in questo modo è possibile quantificare la verità di un fatto).

Infatti, mentre in logica il fatto (ad esempio la malattia) è esprimibile soltanto in termini di veridicità (cioè, può essere soltanto vero o falso), in probabilità possiamo associare anche un valore al fatto, detto grado di credenza (**BELIEFS**), che rappresenta quanto crediamo che il fatto sia vero. Perciò, possiamo utilizzare i beliefs per rappresentare le regole logiche.

Dato un fatto d'interesse X (ad esempio la malattia) ed un'informazione di fondo I (ad esempio interrogazione del paziente per avere informazioni), allora possiamo indicare con $\text{bel}(x|I) \in [0,1]$ il valore di belief associato al fatto X .

Definiamo i beliefs in due modi:

- Axiomi di Cox-James. I beliefs devono soddisfare i seguenti assiomi:

1) se $\text{bel}(x|I) > \text{bel}(y|I)$ ed $\text{bel}(y|I) > \text{bel}(z|I)$ allora $\text{bel}(x|I) > \text{bel}(z|I)$

2) $\exists F$ (funzione) tale che $\text{bel}(x|I) = F(\text{bel}(\neg x|I))$

3) $\exists G$ (funzione) tale che $\text{bel}(x \wedge y|I) = G(\text{bel}(x|I), \text{bel}(y|I \wedge x))$

Teorema: i beliefs devo anche soddisfare gli assiomi di probabilità.

- Sistemi di scommesse (De Finetti). Si definisce una scommessa $m:1$ sul fatto X . Questo significa che se X si verifica allora si vince m , altrimenti si perde 1. In questo caso avremo $\text{bel}(x) = \frac{1}{1+m}$.

N.B.: se m è molto grande allora l'evento è molto raro ($\text{bel}(x) \rightarrow 0$, ovvero crediamo poco nell'evento X). Al contrario, se m è molto piccolo (vicino a 0), allora l'evento è quasi certo ($\text{bel}(x) \rightarrow 1$).

In particolare, possiamo definire la scommessa olandese (Dutch book), una scommessa in cui è garantito perdere.

Esempio: Consideriamo un sistema (gara di cavalli) con 4 possibili scommesse (l'evento A è vero se il cavallo A vince e l'evento B è vero se il cavallo B vince).

	A	B	<small>non vince nessuno</small>	<small>vince A o B ma non entrambi</small>	
quote ← odds <small>(m:1)</small>	$\frac{2}{3} : 1$	$1 : 1$	$3 : 1$	$\frac{3}{7} : 1$	
beliefs <small>(bel = $\frac{1}{1+m}$)</small>	0,6	0,5	0,25	0,7	
denaro scommesso ← #bets	60	50	25	70	

		A	B	$\neg A \wedge \neg B$	$\neg(A \wedge B)$	Tot
T	T	+40	+50	-25	-70	-5
T	F	+40	-50	-25	+30	-5
F	T	-60	+50	-25	+30	-5
F	F	-60	-50	+75	+30	-5

← indica tutte le possibili vincite (a seconda dei traguardi ottenuti dai cavalli: A e B (T indica se il cavallo vince, mentre F indica se il cavallo perde)) che si possono verificare scommettendo i soldi indicati nella prima tabella su tutti i 4 eventi. Notiamo che in ogni caso perdiamo soldi (Dutch Book) il + indica una vittoria, mentre il - indica una sconfitta.

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B) \text{ con } P(A \wedge B) = 1 - P(\neg(A \wedge B)) \text{ ed } P(A \vee B) = 1 - P(\neg(\neg(A \wedge B)))$$

$$\text{Perciò } 1 - P(\neg(\neg(A \wedge B))) = P(A) + P(B) - 1 + P(\neg(A \wedge B)) \text{ e}$$

$$\text{Sostituendo con i valori della prima tabella otteniamo } 1 - 0,25 = 0,6 + 0,5 - 1 + 0,7 \text{ ovvero } 0,75 = 0,8 \text{ (Assurdo)}$$

Quindi, utilizzando gli assiomi di probabilità non si ottiene un'identità.

Teorema (di De Finetti): il sistema di scommesse è un Dutch Book se e solo se i beliefs associati a tali scommesse violano gli assiomi di probabilità.

Vediamo i concetti principali della probabilità:

- $P(A)$ indica la tabella che assegna ad ogni possibile valore (detto realizzazione) di un evento A la probabilità che questo si verifichi.
- A è una variabile aleatoria categorica, ovvero l'insieme delle realizzazioni di A (cioè dei valori che A può assumere) è un insieme finito.

Esempio: A può assumere un valore dell'insieme dei colori {red, green, blue}):

a	$P(A=a)$
red	0,5
green	0,2
blue	0,3

indica la probabilità che A assuma quella realizzazione.

N.B.: $P(a)$ indica una particolare riga della tabella (ad esempio $P(\text{red})$ indica la prima riga della tabella).

$$\bullet \text{ Teorema (di Bayes)}: P(A|B) = \frac{P(B|A)P(A)}{P(A)} \quad (\text{forma tabellare di } P(A=a|B=b) = \frac{P(B=b|A=a)P(A=a)}{P(A=a)}) \quad \forall a, b \text{ (realizzazioni)}$$

$$\bullet \text{ Chain rule: } P(A, B|C) = P(A|B, C)P(B|C)$$

$$\text{N.B.: } P(A, B) = P(A|B)P(B)$$

$$\bullet \text{ Marginalizzazione: } P(A) = \sum_B P(A, B) \quad (\text{ovvero } \forall a \text{ si ha } P(A=a) = \sum_{b \in \text{Dom}(B)} P(A=a, B=b)).$$

$$\bullet P(A) = \sum_B P(A|B)P(B)$$

Esempio (malattia della mucca pazzia (consideriamo lo studio su tutta la popolazione mondiale)):

Applichiamo questi concetti all'esempio della malattia della mucca pazzia. Supponiamo di conoscere i seguenti valori:

$$P(H|CJ) = P(H=\text{true}|CJ=\text{true}) = 0,9 \quad (\text{indica che il 90\% della popolazione malata ha mangiato un hamburger})$$

$$P(H) = P(H=\text{true}) = 0,5 \quad (\text{indica che metà popolazione mangia hamburger})$$

$$P(CJ) = P(CJ=\text{true}) = 10^{-5} \quad (\text{detta incidenza}) \quad (\text{indica che lo 0,00001\% della popolazione è malata})$$

$$\text{Perciò, possiamo valutare la probabilità di contrarre la malattia mangiando un hamburger: } P(CJ|H) = \frac{P(H|CJ)P(CJ)}{P(H)} = \frac{0,9 \cdot 10^{-5}}{0,5} = 1,8 \cdot 10^{-5}$$

Quindi, descriviamo il ragionamento. Innanzitutto chiamiamo:

- **PRIOR**: indica un dato conosciuto prima di aver effettuato un'osservazione (ad esempio $P(CJ)$ è un dato conosciuto a priori, prima di aver parlato con i pazienti).
- **POSTERIOR**: indica un dato conosciuto dopo aver effettuato un'osservazione (ad esempio $P(CJ|H)$ è un dato conosciuto a posteriori, dopo aver parlato con i pazienti).
- **LIKELIHOOD** (verosimiglianza): indica la probabilità dell'evidenza (ad esempio $P(H|CJ)$ indica la probabilità che il paziente abbia mangiato l'hamburger (che rappresenta l'evidenza) assumendo che il paziente sia malato).

Allora possiamo scrivere il ragionamento come Posterior \propto Likelihood \times Prior

N.B.: questo ragionamento riconduce alla formula di Bayes. Infatti, dalla formula di Bayes possiamo scrivere:

$$P(CJ|H) = \frac{P(H|CJ) P(CJ)}{P(H)} \quad \text{dove } P(H) \text{ è un fattore di normalizzazione, perciò può essere trascurato} \quad P(H) = P(H|CJ=1)P(CJ=1) + P(H|CJ=0)P(CJ=0).$$

In questo modo otteniamo la proporzione definita per il ragionamento ($P(CJ|H) = P(H|CJ) P(CJ)$).

Questa soluzione permette di aggiornare il valore di prior effettuando delle osservazioni (inizialmente conosciamo un certo valore $P(CJ)$ che può cambiare dopo avere ottenuto informazioni dal paziente (cioè aver mangiato un hamburger)).

In particolare questo ragionamento è incrementale, ovvero possiamo ottenere nuove informazioni dalle osservazioni già effettuate.

Perciò, data una nuova evidenza avremo Posterior \propto Likelihood \times Posterior, ovvero otteniamo Probabilità di una query data una nuova evidenza \propto Likelihood di una nuova evidenza \times Probabilità di una query data una vecchia evidenza.

In questo modo possiamo aggiornare il belief.

Ad esempio, se interroghiamo nuovamente il paziente per conoscere la presenza dei sintomi (perdita di memoria):

$$P(CJ|M, H) = \frac{P(M|CJ, H)P(CJ|H)}{P(M|H)} \quad \text{otteniamo una nuova evidenza che permette di aggiornare il belief}$$

Perciò $P(CJ|M, H) \propto P(M|CJ, H) \times P(CJ|H)$ (aggiorniamo il belief con l'informazione riguardo alla perdita di memoria).

In generale $P(h|e_1, \dots, e_t) \propto P(e_t|h, e_1, \dots, e_{t-1}) \times P(h|e_1, \dots, e_{t-1})$ dove h è l'ipotesi ed e_1, \dots, e_t sono le evidenze.

Indipendenza marginale: A e B si dicono marginalmente indipendenti, e si indica con $A \perp B$, se $P(A=a|B=b) = P(A=a)$ (con $P(B=b) > 0$).

N.B.: questo significa che B è irrilevante (ovvero non dà alcuna informazione) per predire A.

N.B.: possiamo anche dire che A e B sono marginalmente indipendenti se $P(A=a, B=b) = P(A=a)P(B=b)$.

Indipendenza condizionale: dato C, allora A e B si dicono condizionalmente indipendenti, e si indica con $A \perp B | C$ (relazione ternaria), se

$$P(A=a|B=b, C=c) = P(A=a|C=c) \quad (\text{con } P(B=b|C=c) > 0).$$

N.B.: questo significa che se conosciamo C, allora B è irrilevante per predire A.

N.B.: la relazione di indipendenza condizionale è simmetrica ($x \perp y | z \Leftrightarrow y \perp x | z$)

Questi concetti di indipendenza permettono di semplificare il ragionamento (ad esempio, se sappiamo che un paziente è malato e vogliamo predire se ha perso la memoria, allora non ci interessa più sapere se ha mangiato un hamburger (poiché questa informazione può predire solo la malattia del paziente, che conosciamo di già), perciò diventa irrilevante, quindi $M \perp H | CJ$ (cioè H è indipendente da M per conoscere CJ)).

N.B.: in generale A, B e C sono set di variabili, cioè $A, B, C \subseteq U$ (dove U rappresenta l'universo, ovvero l'insieme di tutte le variabili).

Vediamo un'eccezione in cui l'indipendenza condizionale non ha senso:

consideriamo $P(CJ|H, M, BT)$ dove BT è una nuova variabile (Blood Transfusion) che rappresenta un'altra possibile causa della malattia.

Allora $P(CJ|H, M, BT) \propto P(BT|CJ, H, M)P(CJ|H, M)$ ($h=CJ, e_1=H, e_2=M, e_3=BT$)

In particolare, CJ è una conseguenza sia di H che di BT, mentre M è una conseguenza di CJ, ovvero $H \xrightarrow{BT} CJ \rightarrow M$.

Perciò H e BT sono due possibili cause di CJ e quindi sono probabilisticamente dipendenti: per spiegare la causa (perciò $BT \perp H | CJ$).

BELIEF NETWORKS

Sono algoritmi che permettono di studiare ed implementare i ragionamenti probabilistici appena visti.

Queste reti sono composte da 3 principali aspetti:

1) semantica;

2) inferenza;

3) apprendimento.

In generale, un **BELIEF NETWORK** è un grafo che rappresenta una famiglia di distribuzioni, in cui:

- i nodi rappresentano le variabili;

- gli archi mancanti rappresentano le indipendenze condizionali.

Esistono vari tipi di grafi, ognuno dei quali possiede una semantica diversa. In particolare, concentriamoci sui 2 principali tipi di grafi:

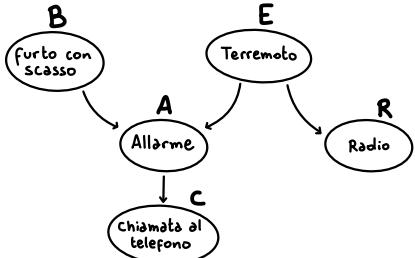
• **grafi orientati** (directed networks);

• **grafi non orientati** (undirected networks);

1) Semantica: (diversa per ogni tipo di rete)

- **GRAFI ORIENTATI**: sono anche detti Bayesian Network (ovvero reti Bayesiane). Tali grafi devono essere aciclici (DAG).

ES (allarme):

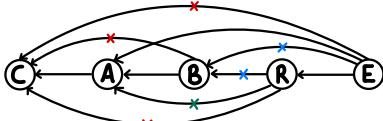


Vogliamo studiare la probabilità congiunta, valida per tutte le distribuzioni:

$$P(C, A, B, R, E) = P(C|A, B, R, E) P(A|R, E) P(B|R, E) P(R|E) P(E)$$

(ottenuta con la chain rule)

Che possiamo rappresentare con il seguente grafo, dove i nodi rappresentano le variabili.



$$P(C, A, B, R, E) = P(C|A, B, R, E) P(A|R, E) P(B|R, E) P(R|E) P(E)$$

In particolare, abbiamo studiato una delle possibili combinazioni del sistema ed abbiamo visto quali sono le indipendenze condizionali tra le variabili. Ad esempio C dipende solamente da A , perciò $P(C|A, B, R, E) = P(C|A)$ e quindi $\{C\} \perp \{B, R, E\} | \{A\}$.

Dato un universo $U = \{X_1, X_2, \dots, X_N\}$ (dove X_1, X_2, \dots, X_N sono le variabili) ed un insieme di archi orientati E , allora $DAG(U, E)$ descrive una famiglia di distribuzioni che soddisfano $P(U) = P(X_1, \dots, X_N) = \prod_{i=1}^N P(X_i | Pa_i(X_i))$, dove $Pa_i(X_i)$ è il padre di X_i nel DAG.

ES:



$$P(U) = P(X_1 | X_2, X_3, X_4) P(X_2 | X_3, X_4) P(X_3 | X_4) P(X_4) = P(X_1 | X_4) P(X_2 | X_4) P(X_3 | X_4) P(X_4)$$

cioè $\{X_1\} \perp \{X_2, X_3\} | \{X_4\}$ ed $\{X_2\} \perp \{X_3\} | \{X_4\}$.

Vogliamo vedere se, ad esempio, è valida la relazione $\{X_1\} \perp \{X_2\} | \{X_4\}$. Possiamo seguire tecniche diverse per risolvere il quesito.

Ad esempio, utilizzando gli assiomi di probabilità possiamo scrivere:

$$P(X_1, X_2 | X_4) = \frac{\sum_{X_3} P(X_1, X_2, X_3, X_4)}{P(X_4)} = \dots = P(x_1 | x_4) P(x_2 | x_4)$$

(dove utilizziamo le relazioni ④ e ② per dimostrare la soluzione).

Abbiamo trovato che $\{X_1\} \perp \{X_2\} | \{X_4\}$ è conseguenza logica di $\{X_1\} \perp \{X_2, X_3\} | \{X_4\}$ ed $\{X_2\} \perp \{X_3\} | \{X_4\}$, cioè ④ \wedge ② \models ③.

Perciò, se per ogni distribuzione P vengono soddisfatte ④ e ②, allora P soddisfa anche ③.

N.B.: questa conseguenza logica è indecidibile, poiché le distribuzioni sono infinite. Perciò non è possibile verificare se per ogni distribuzione che verifica ④ e ②, allora verifica anche ③.

Quindi, in generale non è possibile risolvere questo quesito (cioè ④ \wedge ② \models ③). Però possiamo cercare una relazione sul grafo che permette di approssimare la relazione di indipendenza condizionale.

Ricordiamo che l'indipendenza condizionale è una relazione ternaria tra insiemi di variabili $X \perp Y | Z$, con $X, Y, Z \subseteq U$.

Allora, vogliamo definire la relazione grafica (**D-SEPARATION**), indicata con $X \perp_D Y | Z$, ovvero una relazione per cui è valida $X \perp Y | Z$ (non vale il contrario poiché abbiamo visto che è un problema indecidibile).

Per definire la relazione grafica dobbiamo studiare vari casi (consideriamo l'esempio dell'allarme).

Caso 1 (serial connection): abbiamo una serie di cause ed effetti. In questo caso l'evidenza blocca il cammino.



Ipotizziamo di essere a conoscenza del fatto che sia suonato l'allarme (A), allora il fatto di ricevere la chiamata (C) per notificare l'allarme non fornisce nessuna nuova informazione sul fatto che ci sia stato un furto (B) e viceversa.

$$\text{Dimostriamo che tale relazione ha valenza matematica: } P(C|B, A) = \frac{P(C, B, A)}{P(B, A)} = \frac{P(C|A) P(A|B) P(B)}{P(A|B) P(B)} = P(C|A)$$

Quindi abbiamo dimostrato che, mettendo in evidenza A, allora vale la relazione $[B] \perp [C] | \{A\}$.

Caso 2 (diverging connections): una causa genera più effetti (ovvero un nodo ha solo archi uscenti). In questo caso l'evidenza blocca il cammino.

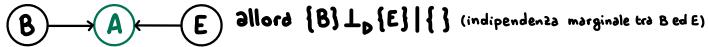


Ipotizziamo di essere a conoscenza del fatto che sia avvenuto il terremoto (E), allora il fatto che sia suonato l'allarme (A) o che sia stato comunicato alla radio (R) non fornisce nessun'altra informazione utile.

$$\text{Dimostriamo che tale relazione ha valenza matematica: } P(E, A, R) = P(E) P(A|E) P(R|E) \Rightarrow P(A|R, E) = \frac{P(E) P(A|E) P(R|E)}{P(R|E) P(E)} = P(A|E)$$

Quindi abbiamo dimostrato che, mettendo in evidenza E, allora vale la relazione $[A] \perp [R] | \{E\}$.

Caso 3 (converging connections): un effetto è generato da più cause (ovvero un nodo ha solo archi entranti). In questo caso l'evidenza abilita il cammino.



Inizialmente il fatto che i ladri tentino un furto (B) è indipendente dal fatto che avvenga un terremoto (E). Questi due eventi, però, diventano dipendenti quando suona l'allarme (A), ovvero non vale la relazione $[B] \perp_D [E] | \{A\}$ (explaining away).

Possiamo mostrare formalmente che inizialmente, in assenza di evidenza, $P(B, E) = P(B) P(E)$.

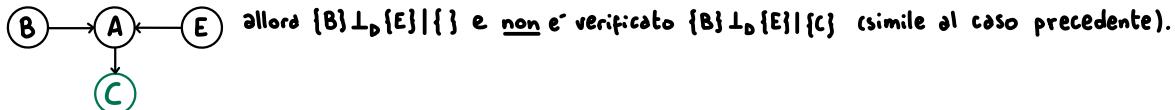
$$\text{Mentre, se fissiamo un valore di A (ovvero si mette in evidenza), allora } P(B, E | A=a) = \frac{P(B, E, A=a)}{P(A=a)} = \frac{P(B) P(E) P(A=a | B, E)}{P(A=a)}$$

Percio', in generale otteniamo che $P(B, E | A=a) \neq 1$ e quindi la relazione $\{B\} \perp \{E\} | \{A\}$ non e' valida.

N.B.: quando un nodo ha tutti gli archi entranti (convergenti) si dice **COLLIDER** (si verifica in explaining away).

Nell'esempio A e' il collider nel cammino (non orientato) B-A-E.

Caso 4: in questo caso l'evidenza in un discendente del collider ha lo stesso effetto dell'evidenza nel collider.



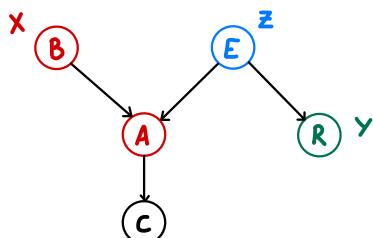
Def (D-separation): dato un DAG D (grafo orientato aciclico), allora chiamiamo **D-SEPARATION** la relazione ternaria sugli insiemi dei vertici di D indicata con $X \perp_D Y | Z$ (con $X, Y, Z \subseteq U$). Tale relazione e' valida se ogni cammino tra un nodo $x \in X$ ed un nodo $y \in Y$ e' bloccato da Z .

N.B.: il cammino viene considerato nella versione non orientata del grafo D.

Def (cammino bloccato): un cammino P si dice **BLOCCATO** da Z (con $Z \subseteq U$) se e solo se si verifica almeno una delle seguenti condizioni:

- 1) P ha un vertice in Z che non e' un collider per P.
- 2) Esiste un vertice C che e' un collider per P, tale che C ed i discendenti di C non appartengono a Z.

ES (allarme):

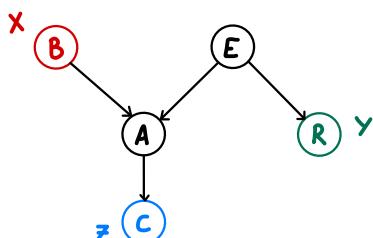


Dobbiamo controllare tutti i cammini che congiungono i nodi di X con i nodi di Y (considerando il grafo non orientato associato a quello in figura) e controllare se questi sono bloccati da Z.

In questo caso, i cammini da considerare sono B-A-E-R ed A-E-R.

Percio', in entrambi i casi dobbiamo passare da Z, ma Z non e' un collider (poiche' ha tutte le frecce divergenti). Quindi Z risulta un bloccante e quindi e' valida la relazione di D-separation.

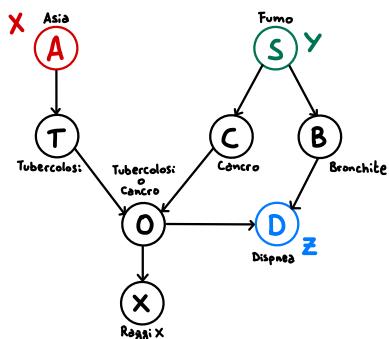
ES (allarme):



In questo caso, il cammino da considerare sara' B-A-E-R.

In particolare, tale cammino risulta aperto (cioe' non bloccato). Questo perche' nel cammino e' presente il nodo A che risulta essere un collider del cammino stesso (se infatti consideriamo il cammino B-A-E-R nel grafo non orientato e poi guardiamo le frecce associate vediamo che il nodo A ha solo frecce convergenti), ma il suo nodo discendente C appartiene a Z. Inoltre il cammino non possiede nessun vertice in Z.

ES (malattie polmonari):



In questo esempio abbiamo piu cammini che uniscono i nodi di X con i nodi di Y.

In particolare consideriamo A-T-O-C-S ed A-T-O-D-B-S.

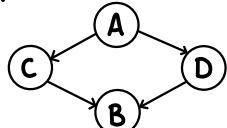
Affinché sia valida la relazione di D-separation e' necessario che tutti i cammini risultino bloccati. Altrimenti, se almeno un cammino e' aperto, allora gli insiemi X ed Y non sono D-separabili. Vediamo che il cammino A-T-O-C-S contiene un collider (cioe' il nodo O) il cui discendente (nodo D) appartiene a Z, perciò risulta essere un cammino aperto.

Inoltre, il cammino A-T-O-D-B-S contiene un collider (cioe' il nodo D) che appartiene a Z, quindi anche questo risulta essere un cammino aperto.

Teorema: sia D un DAG che e' un directed belief network per il dominio $U = (X_1, \dots, X_n)$ (cioe' D e' una rappresentazione grafica della famiglia delle distribuzioni $P(X_1, \dots, X_n) = \prod_{i=1}^n P(X_i | Pa(X_i))$). Se $X \perp_D Y | Z \quad \forall X, Y, Z \subseteq U$, allora $X \perp Y | Z$ (quindi $P(X|Y, Z) = P(X|Z)$).

N.B.: in generale, utilizzando DAG non vale il viceversa. Cioe' non e' sempre possibile realizzare una rappresentazione grafica che soddisfi la relazione ternaria di indipendenza condizionale.

ES:



Supponiamo $\{A\} \perp \{B\} | \{C, D\}$ ed $\{C\} \perp \{D\} | \{A, B\}$.

In questo caso non e' mai possibile realizzare un grafo orientato che soddisfi le relazioni, poiche' in qualunque verso si posizionino le frecce avremo sempre un collider. In particolare, posizionando le frecce come in figura viene soddisfatta la prima condizione ma non la seconda.

-GRAFI NON ORIENTATI: sono anche detti Markov Network (ovvero reti di Markov).

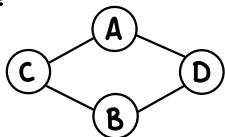
Sia G un grafo non orientato. Allora possiamo definire la distribuzione congiunta come $P(U) = \frac{\prod_{c \in C} \Phi(c)}{Z}$ dove C e' l'insieme delle cricche di G, $\Phi(c)$ e' una funzione potenziale tale che $\Phi: V \rightarrow \mathbb{R}^+$ ed Z e' una funzione detta **FUNZIONE DI PARTIZIONE** (cioe' e' un normalizzatore che garantisce $\sum_u P(u) = 1$).

In generale, pero', tale problema risulta essere difficile ed intrattabile. Ma queste reti risultano essere molto efficaci nella nozione di

separazione, poiché non sono presenti archi orientati. Infatti, possiamo banalmente definire il concetto di U-separation come segue:

Def(U-separation): dato un grafo non orientato G , allora chiamiamo **U-SEPARATION** la relazione ternaria sugli insiemi dei vertici di G indicata con $X \perp_U Y \mid Z$ (con $X, Y, Z \subseteq U$). Tale relazione è valida se rimuovendo Z dal grafo G (cioè se consideriamo il sottografo indotto da $U \setminus Z$), allora X ed Y sono disconnessi (cioè non c'è nessun cammino tra un nodo $x \in X$ ed un nodo $y \in Y$).

Es:



Supponiamo $\{A\} \perp \{B\} \mid \{C, D\}$ ed $\{C\} \perp \{D\} \mid \{A, B\}$.

Se rimuoviamo i nodi C e D , allora A e B risultano disconnessi.

Allo stesso modo, se rimuoviamo i nodi A e B , allora C e D risultano disconnessi.

Teorema: sia G un grafo non orientato. Se $X \perp_U Y \mid Z$, allora $X \perp Y \mid Z$.

2) Inferenza

Il problema dell'inferenza consiste nel calcolare $P(Q|e)$ (cioè calcolare la probabilità di una Query data l'evidenza), dove $Q \subseteq U$ rappresenta la Query ed $e \subseteq U$ rappresenta l'evidenza.

Ad esempio, $P(C|S)$ rappresenta la probabilità di avere un cancro (C) sapendo che il paziente fuma (S).

In generale, il problema dell'inferenza è difficile ed intrattabile (NP-completo). Esistono molti tipi di algoritmi che permettono di calcolare la probabilità condizionata $P(Q|e)$. In particolare, studiamo l'algoritmo del junction tree adattato al problema dell'inferenza probabilistica.

JUNCTION TREE PER L'INFERENZA PROBABILISTICA:

L'idea di questo algoritmo è quella di effettuare uno scambio di messaggi tra i nodi del junction tree.

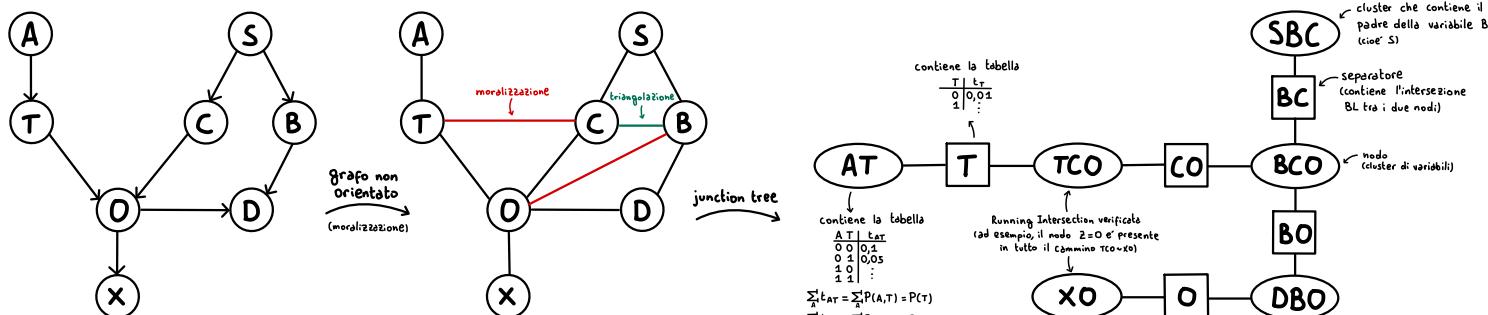
Tale algoritmo richiede che:

- I nodi sono dei sottoinsiemi di U (cioè sono clusters di variabili);
- L'unione di tutti i nodi deve essere U ;
- Per ogni variabile $X_i \in U$ deve esistere un cluster che contiene X_i ed $Pa(X_i)$;
- I nodi ed i separatori (rappresentati da un nodo quadrato che indica l'intersezione tra due nodi del junction tree) contengono tabelle;
- Si effettua la Running Intersection.

N.B: per realizzare il junction tree associato al grafo, dobbiamo prima di tutto trasformare il grafo in un grafo non orientato.

Per effettuare correttamente questa trasformazione dobbiamo eseguire l'operazione di **MORALIZZAZIONE**, ovvero si congiungono tutti i padri che non sono connessi. Successivamente possiamo applicare l'algoritmo già studiato sul grafo non orientato ottenuto.

Esempio (malattie polmonari): $U = \{A, T, E, C, S, B, D, X\}$



N.B: la Running Intersection viene garantita con il max spanning tree, mentre l'esistenza di un cluster che contiene un nodo X_i ed i suoi padri viene garantita con la moralizzazione (questa infatti garantisce che padri e figli si trovino nella stessa critica).

Una volta costruito il junction tree, dobbiamo inizializzare le tabelle associate ad ogni nodo e ad ogni separatore con dei valori estratti dalla rete. Successivamente, possiamo effettuare lo scambio di messaggi tra nodi.

Perciò, i passaggi di questo algoritmo sono:

1) Inizializzazione delle tabelle (indicate con t). Dati i nodi del junction tree V ed W (in particolare, V ed W sono dei clusters di variabili),

allora indichiamo con t_V la tabella di V (possiede una riga per ogni possibile assegnamento su V) e con t_W la tabella di W .

Su queste tabelle possiamo eseguire le seguenti operazioni:

- **Prodotto.** Dati v^* una tupla per V ed w^* una tupla per W , allora $t_V * t_W(v^*, w^*) = t_{V \cup W}(v^*, w^*)$ (rappresenta una tabella in $V \cup W$).

- **Marginalizzazione.** Dato $K \subseteq V$, allora $t_K = \sum_{V \setminus K} t_V$.

Possiamo implementare l'inizializzazione delle tabelle con i seguenti step:

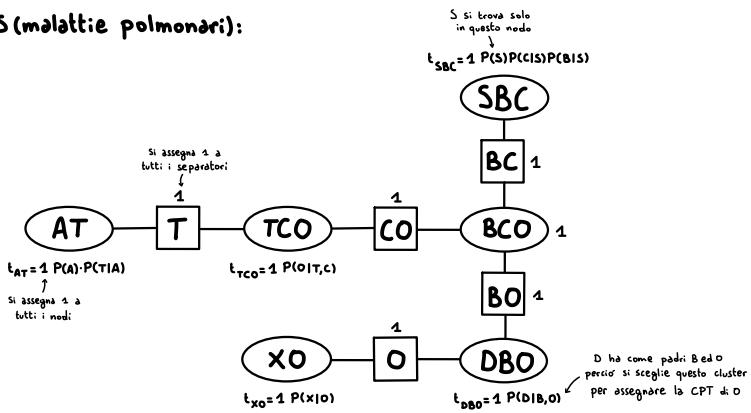
1) $\forall V$ (cluster) si assegna $t_V = 1$ (1 indica la tabella che contiene tutti i valori 1).

2) $\forall S$ (separatore) si assegna $t_S = 1$.

3) $\forall X_i$ (variabile), si sceglie un cluster V tale che $\{X_i\} \cup Pa(X_i) \subseteq V$ e si assegna $t_V = t_V * P(X_i | Pa(X_i))$.

(CPT)
Conditional Probability Table

ES (malattie polmonari):



Per ogni variabile dobbiamo assegnare la CPT al cluster che contiene sia la variabile che tutti i suoi padri. Se esiste un solo cluster che contiene quella variabile allora si sceglie obbligatoriamente quel cluster.

N.B: se dopo l'inizializzazione delle tabelle facciamo il rapporto tra il prodotto di tutte le tabelle dei nodi ed il prodotto di tutte le tabelle dei separatori, otteniamo la probabilità congiunta su tutto il dominio: $P(U) = \frac{\prod_i P(v_i)}{\prod_j P(s_j)}$.

2) Assorbimento (scambio di messaggi). Dati due nodi del junction tree V ed W, a cui sono associate le tabelle t_v ed t_w , ed un separatore S, a cui è associata la tabella t_s , allora il messaggio inviato dal nodo V al nodo W sarà dato dalle seguenti operazioni:

- 1) si calcola il messaggio: $t_s^* = \sum_{v \in S} t_v$ (dove t_s^* indica l'aggiornamento della tabella di S).
- 2) si aggiorna la tabella di W: $t_w^* = t_w \cdot \frac{t_s^*}{t_s}$ (dove t_w^* indica l'aggiornamento della tabella di W).
- 3) si memorizza t_s^* in S.

ES:



N.B: il messaggio può essere scambiato anche in direzione opposta (da W verso V).

Teorema (collegamento consistente): un collegamento è consistente se $\sum_{v \in S} t_v = t_s = \sum_{w \in S} t_w$ (cioè se il messaggio scambiato in entrambe le direzioni (da V a W ed da W a V) è lo stesso).

N.B: in questo caso (ovvero se il collegamento è consistente) l'assorbimento non ha effetti.

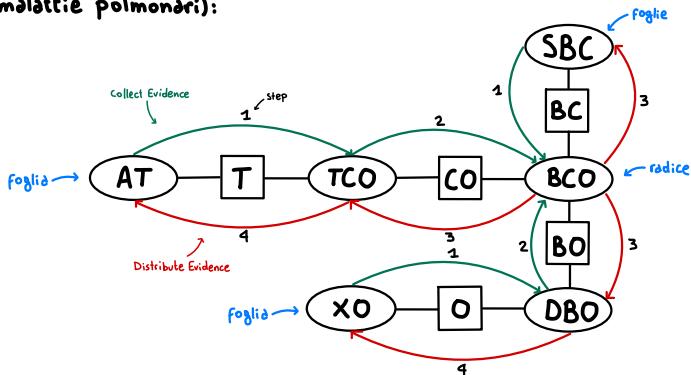
Teorema: l'assorbimento in entrambe le direzioni rende il collegamento consistente.

Perciò, l'idea per effettuare lo scambio di messaggi è quella di scegliere arbitrariamente un nodo del grafo non orientato e farlo diventare la radice dell'albero. Di conseguenza verranno automaticamente definite delle foglie dell'albero. In questo modo possiamo effettuare due operazioni:

- Collect Evidence, ovvero si esegue l'assorbimento dalle foglie verso la radice dell'albero (in parallelo);
- Distribute Evidence, ovvero si esegue l'assorbimento dalla radice verso le foglie dell'albero (in parallelo);

Al termine di queste due operazioni abbiamo ottenuto un albero consistente, poiché ogni coppia di nodi adiacenti ha effettuato l'assorbimento in entrambe le direzioni.

ES (malattie polmonari):



3) Apprendimento:

L'apprendimento riguarda due aspetti importanti, ovvero l'apprendimento dei parametri e l'apprendimento della struttura del grafo (cioè degli archi della struttura Bayesiana). In particolare, studiamo l'apprendimento dei parametri.

L'apprendimento dei dati viene effettuato tramite il metodo di **MASSIMA VEROLOGIANTZA** su un dataset di realizzazioni osservate.

N.B: assumiamo che tale dataset sia completo, ovvero che tutte le possibili realizzazioni di una variabile siano state osservate almeno una volta.

Per garantire tale assunzione possiamo applicare l'operazione di **LAPLACE SMOOTHING** sul dataset, ovvero affianchiamo a tale dataset un dataset immaginario in cui ogni realizzazione si verifica una sola volta. In questo modo otteniamo $\hat{\theta}_j = \frac{n_j + 1}{n + K}$ $\forall j = 1 \dots K$, dove n_j indica il numero di linee del dataset in cui si osserva la realizzazione j (cioè indica il numero di volte in cui si verifica j), n indica la dimensione del dataset principale ed K indica il numero di possibili realizzazioni.

ES (dado):

Dataset	Dataset immaginario
5	1
3	2
2	3
1	4
4	5
4	6

Abbiamo 6 possibili realizzazioni che indicano le facce del dado.

Se non consideriamo il dataset immaginario: $\hat{\theta}_2 = P(x=2) = \frac{3}{7}$, $\hat{\theta}_6 = \frac{0}{7} = 0$ (dataset non completo poiché la variabile 6 non è stata osservata)

Se effettuiamo Laplace Smoothing: $\hat{\theta}_2 = \frac{3+1}{7+6} = \frac{4}{13}$, $\hat{\theta}_6 = \frac{0+1}{7+6} = \frac{1}{13}$ (dataset completo poiché sono state osservate tutte le variabili)

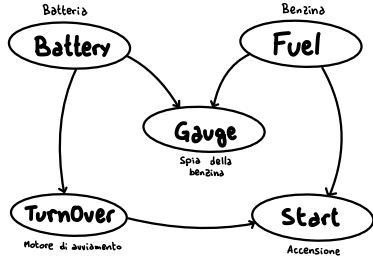
N.B: dobbiamo anche assumere che i dati sono indipendenti ed identicamente distribuiti (i.i.d.).

La funzione di verosimiglianza si indica come $L(\theta) = P(D, \theta)$ dove D indica il dataset e θ indica l'insieme di tutti i parametri.

Poiché abbiamo assunto che i dati sono i.i.d. possiamo scrivere $L(\theta) = P(D, \theta) = \prod_{i=1}^n P(X^{(i)}; \theta) = \prod_{i=1}^n P(X_1^{(i)}, X_2^{(i)}, \dots, X_N^{(i)}; \theta) = \prod_{i=1}^n \prod_{j=1}^N P(X_j^{(i)} | P_{\theta j}; \theta_j)$.

N.B: indichiamo i parametri con $\theta_{jk} = P(X_k=j | P_{\theta j}=k)$ dove $j=1 \dots \# \text{valori per } X_k$, $k=1 \dots \# \text{realizzazioni di tutti i padri}$. Di conseguenza indichiamo la massima verosimiglianza come $\hat{\theta}_{jk} = \frac{n_{jk}}{\sum_k n_{jk}}$ dove n_{jk} indica il numero di volte in cui è stato osservato $X_k=j$ insieme ai padri $P_{\theta j}=k$.

ES (accensione di un auto):



Case	TurnOver	Fuel	Start
1	normal	yes	yes
2	slowly	yes	yes
3	normal	yes	yes
4	no	yes	no
5	normal	yes	yes
6	normal	yes	no
7	slowly	yes	no
8	normal	yes	yes
9	normal	no	no
10	normal	yes	no
11	slowly	no	no
12	normal	no	no
13	no	yes	yes
14	normal	yes	yes
15	normal	yes	yes
16	no	no	no

Dataset
(mancano le colonne di Battery e Gauge)

Notiamo che le possibili realizzazioni delle variabili Fuel, Start, Battery e Gauge sono 2 (yes o no), mentre le possibili realizzazioni della variabile TurnOver sono 3 (normal, slowly, no).

Laplace Smoothing per evitare la situazione in cui una delle realizzazioni di Start non è stata osservata

Si conta il numero di volte in cui si ha contemporaneamente Fuel=yes e TurnOver=normal e poi si conta quante volte si verifica Start=yes nei possibili casi contati in precedenza

$$P(\text{Start}=\text{yes} | \text{Fuel}=\text{yes}, \text{TurnOver}=\text{normal}) = \frac{6+1}{8+2} = \frac{7}{10}$$

SUPERVISED LEARNING (apprendimento supervisionato):

Questo tipo di apprendimento viene impiegato su diversi problemi come il riconoscimento di immagini oppure la categorizzazione di un testo.

In particolare viene impiegato su problemi di natura predittiva.

Sia uno spazio di istanze X (ad esempio tutte le possibili foto) ed uno spazio di uscita Y (cioè l'insieme delle possibili predizioni).

L'obiettivo è quello di trovare una funzione $h: X \rightarrow Y$ tale che $h(x)$ rappresenta una predizione su un'istanza $x \in X$.

Se Y è uno spazio discreto (ad esempio, vogliamo predire se la foto è una foto di cane, di gatto, di scimmia, ecc...), allora diciamo che il problema è un **PROBLEMA DI CLASSIFICAZIONE**.

Se, invece, Y è uno spazio continuo (ad esempio vogliamo predire il prezzo di una casa conoscendo la dimensione, la collocazione, il numero di stanze, ecc...), allora diciamo che il problema è un **PROBLEMA DI REGRESSIONE**.

L'idea è quella di utilizzare un dataset costituito da esempi, ovvero da coppie di istanze $x \in X$ e label $y \in Y$. Cioè $D = \{(x^{(i)}, y^{(i)}) \mid i=1 \dots n\}$ dove $(x^{(i)}, y^{(i)})$ rappresenta l'esempio.

Dato tale dataset D e data una classe di funzioni H (detta spazio delle ipotesi), allora dobbiamo scegliere la funzione $h \in H$ che predica bene.

N.B: per spiegare questo concetto si introduce una funzione $L: Y \times Y \rightarrow \mathbb{R}^+$ detta loss tale che $L(h(x), y)$ indica il prezzo di predire $h(x)$ quando è vera l'etichetta y . I casi più semplici sono, ad esempio, $L(h(x), y) = 1$ se $h(x) \neq y$ (per problemi di classificazione) oppure $L(h(x), y) = (h(x) - y)^2$ (per problemi di regressione).

Quindi, trovare una funzione $h \in H$ che predice bene significa trovare una funzione che comporti una loss piccola.

Perciò, il problema si riduce a trovare $\arg \min_{h \in H} \{ \mathbb{E}[L(h(x), y)] \}$. Tale problema risulta essere uno strano problema di ottimizzazione, poiché la funzione obiettivo non è stata osservata ($P(x|y)$ è sconosciuta) e quindi non è risolubile.

Possiamo risolvere questo problema utilizzando l'**EMPIRICAL RISK MINIMIZATION (ERM)**, ovvero $\arg \min_{h \in H} \left\{ \frac{1}{n} \sum_{i=1}^n L(h(x^{(i)}, y^{(i)})) \right\}$

(ovvero si fa la minimizzazione tra i possibili loss)

N.B: tale problema ERM può essere NP-completo.

Esistono molti algoritmi di Supervised Learning che si differenziano dalle varie scelte effettuate (ad esempio di H ed L). I principali algoritmi sono:

- Lazy: $h(x) = y^{(i)}$ se $i^* = \arg \min_i \{d(x, x^{(i)})\}$ (si guarda il valore dell'istanza più vicina) Tale algoritmo però soffre del problema della dimensionalità (non funziona per problemi a più dimensioni).

- Modelli grafici:



- Algoritmi Online (ad esempio Perceptron).

- Alberi.

- Metodi di Ensemble (ad esempio Random Forest o AdaBoost).

- Neural Networks (ad esempio Deep Learning)

Vediamo alcuni di questi algoritmi.

• **PERCEPTRON**: tale algoritmo assume come spazio d'ipotesi la classe di funzioni $H = \{h: \mathbb{R}^p \rightarrow \{-1, 1\} \mid h = \text{sgn}(w^T x + b)\}$ dove $w \in \mathbb{R}^p$, $b \in \mathbb{R}$ sono i parametri dell'algoritmo. I dati sono del tipo $D = \{(x^{(i)}, y^{(i)}) \mid i=1 \dots n\}$, ovvero sono coppie di valori dove $y^{(i)}$ rappresenta la label del dato, mentre $x^{(i)}$ rappresenta la feature del dato (cioè la variabile usata per predire).

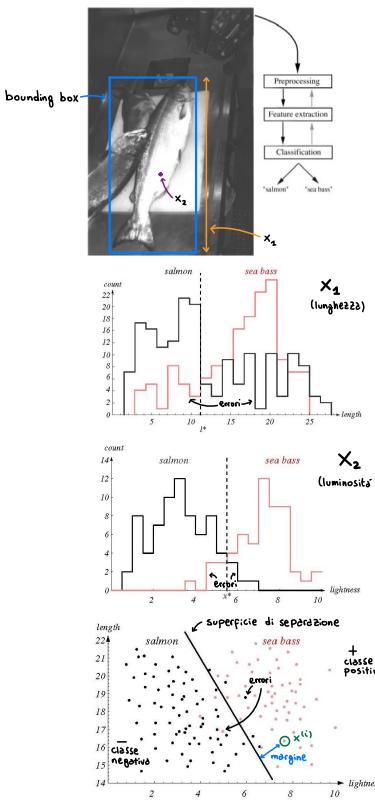
vettori di dimensione p
di valori reali

trasposto

ES (smistamento di pesci):

Consideriamo un sistema di smistamento pesci in cui dobbiamo distinguere tra salmoni e branzini.

Una telecamera rileva i pesci presenti su un nastro trasportatore e li separa in base alla loro tipologia.



Per effettuare questo riconoscimento si suddivide il lavoro nelle seguenti fasi:

- Preprocessing:** si individua il bounding box (ovvero l'area dell'immagine) in cui si trova il pesce;
- Feature extraction:** dalla foto si estraendo delle features che compongono il vettore x (ad esempio, se $p=2$ allora consideriamo un vettore $x=(x^1, x^2)$ dove x_1 e' la lunghezza del bounding box ed x_2 e' la luminosita' media del bounding box).
- Classification:** si puo' realizzare un istogramma per stimare la class conditional densities, ovvero la probabilita' che una certa feature appartenga ad una determinata label (dove l'insieme delle label sara' $y = \{\text{salmon}, \text{sea bass}\}$). Ad esempio, per stimare $P(x_1 | y=\text{salmon})$ notiamo dal primo istogramma che abbiamo maggiore probabilita' a destra del valore $\text{length}=15$. Se invece stimiamo $P(x_2 | y=\text{salmon})$ notiamo dal secondo istogramma che abbiamo maggiore probabilita' a destra del valore $\text{lightness} \approx 5$. Inoltre, notiamo che nel secondo istogramma possiamo effettuare meno errori di classificazione (poiche' ad esempio, se il valore di $x_2=8$ allora sara' sicuramente un salmonone). Mettendo insieme i risultati otteniamo un iperpiano (in questo caso e' una retta poiche' $p=2$ e quindi siamo in 2 dimensioni). Il numero di errori sono quelli che vengono classificati erroneamente nella zona sbagliata (ad esempio i punti neri che si trovano a destra della retta che rappresentano i branzini classificati erroneamente come salmoni).

L'idea e' quindi quella di cercare un iperpiano (o superficie di separazione) con equazione $w^T x + b = 0$.

L'obiettivo (ERM) e' quello di trovare il migliore iperpiano per il dataset D, ovvero $\min_{w,b} \sum_{i=1}^n \mathbb{1}_{\{h(x^{(i)}) \neq y^{(i)}\}}$ (cioe' si cerca i valori migliori di w e b che minimizzano il numero di errori).

Se prendiamo un punto $x^{(i)}$ sul grafico possiamo definire:

Def(margine funzionale): chiamiamo **MARGINE FUNZIONALE** di $(x^{(i)}, y^{(i)})$ il valore $\delta^{(i)} := y^{(i)} \cdot (w^T x^{(i)} + b)$.

N.B.: se $w^T x^{(i)} + b > 0$ (ovvero siamo nella classe positiva) ed $y^{(i)} = 1$ (ovvero $y^{(i)} = \text{salmon}$), allora $\delta^{(i)} > 0$. Stessa cosa vale se $w^T x^{(i)} + b < 0$ (ovvero siamo nella classe negativa) ed $y^{(i)} = -1$ (ovvero $y^{(i)} = \text{branzino}$).

Al contrario, se i segni sono discordi, allora $\delta^{(i)} < 0$. Questo significa che abbiamo effettuato una classificazione errata.

Def(classificazione corretta): diciamo che una classificazione e' **CORRETTA** se e solo se $\delta^{(i)} > 0$.

N.B.: idealmente vorremo realizzare un iperpiano che abbia tutti margini positivi, cioe' $\delta^{(i)} > 0 \quad \forall i=1 \dots n$.

Def(margine geometrico): chiamiamo **MARGINE GEOMETRICO** il valore $\frac{\delta^{(i)}}{\|w\|}$, che rappresenta la distanza del punto $x^{(i)}$ dall'iperpiano.

N.B.: se moltiplichiamo i parametri w e b dell'equazione dell'iperpiano per una costante otteniamo la stessa equazione (ovvero $w^T x + b = 0$ ed $100w^T x + 100b = 0$ sono la stessa equazione), ma il margine funzionale viene amplificato del valore della costante (detto fattore di zoom).

Per eliminare tale fattore spesso si sceglie la soluzione con $\|w\|=1$. Per fare cio' si cerca il **margine del dataset**.

Def(margine del dataset): chiamiamo **MARGINE DEL DATASET** il valore $\gamma = \max_{w,b} \min_{i=1 \dots n} y^{(i)} \cdot (w^T x^{(i)} + b)$ tale che $\|w\|=1$ (detto problema dell'iperpiano massimo margine)

Def(dataset linearmente separabile): diciamo che un dataset e' **LINEARMENTE SEPARABILE** se $\gamma > 0$.

Vediamo quindi lo pseudocodice dell'algoritmo:

```

function Perceptron(D):
    initialize:  $w_0=0, b_0=0$ 
     $t=0$  (indice d'iterazione)
     $R=\max_i \|x^{(i)}\|$  (raggio della sfera che contiene il dataset)

repeat:
     $nerr=0$  (contatore del numero di errori commessi dall'algoritmo in un'epoca)
    for  $i=1, \dots, n$ 
        if  $y^{(i)} \cdot (w^T x^{(i)} + b) < 0$  (falso positivo o falso negativo)
            then:  $w_{t+1} = w_t + y^{(i)} x^{(i)}$ 
                   $b_{t+1} = y^{(i)} R^2$ 
                   $nerr=nerr+1$ 
                   $t=t+1$ 
    until  $nerr=0$ 

```

L'idea dell'algoritmo e' quella di spostare l'iperpiano ogni volta che si commette un errore facendo sì che il margine diventi positivo (se l'errore e' un falso positivo allora si sottrae il vettore x all'iperpiano (poiche' in realtà era negativo), altrimenti, se l'errore e' un falso negativo allora si aggiunge il vettore x all'iperpiano (poiche' in realtà era positivo)). Tale algoritmo e' **greedy** (non garantisce l'ottimo). L'algoritmo termina quando si esegue una intera epoca senza commettere errori ($nerr=0$). Cioe' pero' non e' sempre garantito.

Infatti, se l'algoritmo non è linearmente separabile, allora non ha soluzioni e quindi non termina.

Teorema: dato un dataset non banale (ovvero un dataset che ha almeno un esempio positivo ed un esempio negativo), supponiamo che

$\exists w_x, b_x$ tali che $y^{(i)} \cdot (w_x^T x^{(i)} + b_x) > \gamma$ $\forall i$ con $\|w_x\| = 1$. Allora il numero di errori commessi dall'algoritmo Perceptron è al più $(\frac{2R}{\gamma})^2$.

Dim:

Consideriamo $R=1$ senza perdere di generalità.

Chiamiamo $\hat{w} = \begin{bmatrix} w \\ b \end{bmatrix}$ ed $\hat{x} = \begin{bmatrix} x \\ 1 \end{bmatrix}$. In questo modo possiamo scrivere $w^T x + b = \hat{w}^T \hat{x}$.

Di conseguenza possiamo scrivere l'aggiornamento $\begin{cases} w_{t+1} = w_t + y^{(i)} x^{(i)} \\ b_{t+1} = y^{(i)} R^2 \end{cases}$ con una sola equazione $\hat{w}_t = \hat{w}_{t-1} + y^{(i)} \hat{x}^{(i)}$

Quindi abbiamo: $\underbrace{\hat{w}_x^T \hat{w}_t}_{> \gamma}$

$$\hat{w}_x^T \hat{w}_t = \hat{w}_x^T \hat{w}_{t-1} + y^{(i)} \hat{w}_x^T \hat{x}^{(i)} > \hat{w}_x^T \hat{w}_{t-1} + \gamma$$

Per induzione avremo:

$$t=0, \text{ allora } \hat{w}_0 = 0$$

$$t=1, \text{ allora } \hat{w}_x^T \hat{w}_1 > \gamma$$

$$t=2, \text{ allora } \hat{w}_x^T \hat{w}_2 > 2\gamma$$

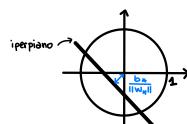
⋮

$$t \text{ generico, allora } \hat{w}_x^T \hat{w}_t > t\gamma \quad \text{Poiché altrimenti non ci sarebbe stato l'errore} \leq 0$$

$$\text{Inoltre, scriviamo } \|\hat{w}_t\|^2 = \|\hat{w}_{t-1}\|^2 + 2 y^{(i)} \hat{w}_{t-1}^T \hat{x}^{(i)} + \|x^{(i)}\|^2 < \|\hat{w}_{t-1}\|^2 + \|x^{(i)}\|^2$$

$$\text{Per induzione otteniamo } \|\hat{w}_t\|^2 < t^2 \quad \text{Per ipotesi } \|\hat{w}_0\| = 1$$

$$\text{Mettendo insieme ① e ② otteniamo } t\gamma < \hat{w}_x^T \hat{w}_t \leq \|\hat{w}_x^T\| \cdot \|\hat{w}_t\| \leq \|\hat{w}_x^T\| \cdot \sqrt{t^2} = (1+b_x) \sqrt{t}$$



Tutti i punti si trovano all'interno della sfera di raggio $R=1$.

Inoltre, poiché il dataset è non banale, allora l'iperpiano taglia sicuramente la sfera in 2 punti.

La distanza dell'iperpiano dall'origine sarà $\frac{b_x}{\|\hat{w}_x\|} < R=1$ con $\|\hat{w}_x\|=1$ per ipotesi, perciò deve essere $b_x < 1$.

Di conseguenza avremo $t\gamma < 2\sqrt{t}$ CVD

N.B.: questo teorema garantisce la terminazione dell'algoritmo poiché implica che il numero di errori sia finito.

Una variante di questo algoritmo è detta **VOTED PERCEPTRON**.

Tale variante viene utilizzata nel caso in cui il dataset non sia linearmente separabile. Per realizzarla basta sostituire il repeat con un ciclo che termina in un certo numero di passi (si limita il numero massimo di epoch).

Inoltre inseriamo un contatore che tiene traccia del numero di volte in cui un iperpiano non ha fatto errori.

function VotedPerceptron(D):

```
initialize:  $w_0=0, b_0=0$ 
           $t=0$  (indice d'iterazione)
           $R=\max_i \|x^{(i)}\|$  (raggio della sfera che contiene il dataset)
           $C_0=0$  (conta il numero di classificazioni corrette dell'iperpiano t)
```

for epoch=1, .., max_epochs

nerr=0 (contatore del numero di errori commessi dall'algoritmo in un'epoch)

for i=1,...,n

if $y^{(i)} \cdot (w_t^T x^{(i)} + b) < 0$ (falso positivo o falso negativo)

then: $w_{t+1} = w_t + y^{(i)} x^{(i)}$

$b_{t+1} = y^{(i)} R^2$

nerr=nerr+1

t=t+1

C_{t+1}=1

else $C_t=C_t+1$

return $[(w_r, b_r, C_r) \text{ con } r=1 \dots t]$

Tale algoritmo ritorna una lista di tutti gli iperpiani con i rispettivi valori C che rappresentano il numero di volte in cui un iperpiano ha effettuato una corretta classificazione.

In questo modo possiamo combinare le soluzioni facendo una votazione sul peso C_r , ovvero $h(x) = \operatorname{sgn}\left(\sum_{r=1}^t C_r \operatorname{sgn}(w_r^T x + b_r)\right)$ detta **VOTE MODE** (si dà più peso agli iperpiani con C_r più grandi).

Un'altra tecnica per combinare tali soluzioni è detta **AVERAGE MODE** e si ottiene con $h(x) = \operatorname{sgn}\left(\frac{1}{t} \sum_{r=1}^t C_r (w_r^T x + b_r)\right)$.

Un'altra variante del perceptron è detta **DUAL PERCEPTRON** e si ottiene ridefinendo il problema dell'iperpiano massimo margine.

Da questo possiamo calcolare $h(x) = \operatorname{sgn}\left(\sum_{i=1}^n \alpha^{(i)} y^{(i)} x^T x^{(i)} + b\right)$ ed $w = \sum_{i=1}^n \alpha^{(i)} y^{(i)} x^{(i)}$, dove $\alpha^{(i)}$ conta il numero di volte in cui $(x^{(i)}, y^{(i)})$ è stato classificato male durante il training dei dati.

ES (classificazione di documenti):

Supponiamo che x sia il vettore indicatore di un set, ad esempio consideriamo come set un insieme di parole e vogliamo fare una classificazione di documenti testuali. Allora $x_j = \begin{cases} 1 & \text{se la } j\text{-esima parola del vocabolario è presente nel documento} \\ 0 & \text{altrimenti} \end{cases}$ (l'applicazione usata nei filtri anti-spam, in cui si classificano come positive le mail spdm (che contengono certe parole) e negative mail non spdm).

In questo caso $x^T x^{(i)} = \sum_j x_j x_j^{(i)}$, ovvero per ogni parola si guarda se è presente nel documento di test e nel documento di train (ovvero si guarda il numero di parole in comune).

N.B.: possiamo sostituire il prodotto scalare con una funzione Kernel che comporta la realizzazione di una classe di algoritmi detta **KERNEL MACHINES**. Dato $K: X \times X \rightarrow \mathbb{R}$ una funzione che compara (in senso di similarità) due istanze $x, z \in X$ tale che $\exists \phi: X \times X \rightarrow \mathbb{R}$ per la quale $K(x, z) = \langle \phi(x), \phi(z) \rangle$.

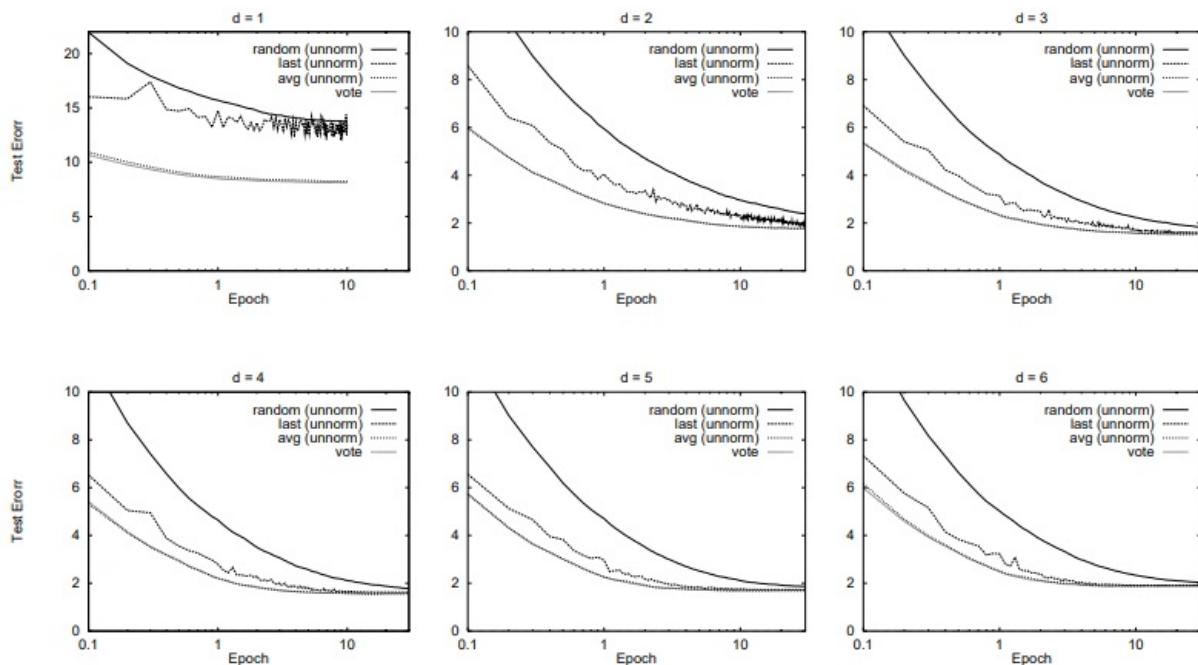
KERNEL TRICK: ϕ può essere molto grande (anche ∞).

RBF KERNEL (Radial Basis Function Kernel): $K(x, z) = e^{-\delta \|x - z\|^2}$ (un punto z è simile ad x se si trova vicino a questo)

In questo modo possiamo realizzare superfici di separazione diversi da iperpidi (come iperboli, ellissi, ecc...).

In particolare possiamo realizzare composizioni di funzioni, ovvero componendo la feature map ϕ con un classificatore $h(x) = \text{sgn}(w^T \phi(x) + b)$ dove $\phi(x)$ è detta anche rappresentazione di x .

Possiamo anche apprendere $\phi(x)$ (learning representation), invece di scriverla attraverso un kernel.



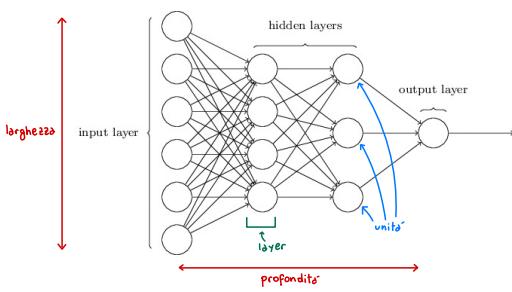
Limiti del perceptron:

• la forma primale ($w^T x + b$) può risolvere solamente problemi linearmente separabili;

• la forma duale ($\sum_i \alpha^{(i)} y^{(i)} K(x, x^{(i)})$ con ad esempio $K(x, x^{(i)}) = e^{-\gamma \|x - x^{(i)}\|^2}$) può comportare il problema dell'esplosione esponenziale.

N.B.: abbiamo visto che possiamo scrivere la forma duale anche come $\langle w, \phi(x) \rangle$ dove $K(x, z) = \langle \phi(x), \phi(z) \rangle$.

Vogliamo poter apprendere $\phi(x)$ costruendo una funzione che possiede dei parametri. Un'idea è quella di seguire il modello del neurone di McCulloch-Pitts $\sigma(w^T x + b)$ dove σ è una funzione non lineare (ad esempio una funzione gradino).



Perciò possiamo combinare K unità ponendo $\Phi_K(x) = \sigma(w_K^T x + b_K)$, dove $\Phi_K(x)$ è detta attivazione e σ è detta funzione di attivazione (creando così un layer).

Ripetendo questa operazione più volte possiamo costruire più layer (ognuno con un certo numero di unità). In questo modo possiamo realizzare un **MULTILAYERED PERCEPTRON** (oppure **FEEDFORWARD NEURAL NETWORK**) che rappresenta una estensione del perceptron.

N.B.: gli hidden layers sono rappresentazioni non visibili costruite sui dati di input.

A differenza dal Kernel, in questo caso la funzione $\phi(x)$ non è fissata e quindi può essere appresa (learning representation).

In questo modo possiamo predire l'output a partire dai soli dati di input (cioè non sono necessarie conoscenze).

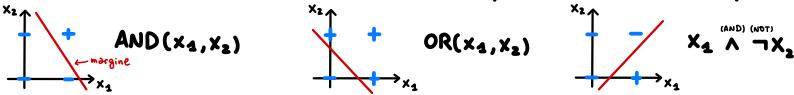
N.B.: $\Phi_K(x)$ utilizza una funzione σ non lineare per mappare la combinazione di unità lineari $w_K^T x + b_K$.

Teorema: se la "larghezza" di un singolo layer del MLP è abbastanza grande, allora si può approssimare qualsiasi funzione smooth.

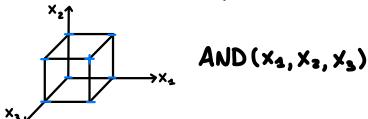
N.B: in particolare possiamo realizzare una qualsiasi funzione Booleana $\{0,1\}^P \rightarrow \{0,1\}$.

Consideriamo la funzione $\sigma = \neg$ (funzione gradino) ed un'architettura con un singolo layer.

Vediamo ad esempio come realizzare le funzioni AND ed OR per $P=2$.



Questo vale anche per dimensioni maggiori. Ad esempio, la funzione AND per $P=3$ sarà:



Perciò possiamo realizzare qualsiasi funzione booleana di base. In particolare possiamo scrivere ogni funzione Booleana f in forma normale congiuntiva (oppure in forma normale disgiuntiva).

Questo comporta l'utilizzo di una unità nascosta (hidden unit) per clausola (cioè si fa OR tra tutti gli input) ed un AND sull'unità di output. In questo modo possiamo implementare una qualsiasi funzione Booleana.

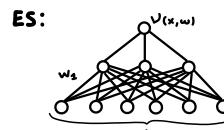
Il problema che si verifica è che il margine si riduce con l'aumentare della dimensione. Questo implica un limite sulla robustezza. Inoltre risulta poco efficiente (aumentando la dimensione, il numero di unità nascoste aumenta esponenzialmente).

Perciò, si preferisce realizzare architetture con "larghezza" minore ma con "profondità" maggiore. Quindi la "profondità" gioca un ruolo molto importante.

Torniamo a studiare l'apprendimento di reti neurali (Training Neural Nets).

Notazione: indichiamo con

- w tutti i pesi;
- $v(x, w)$ l'output (non è ancora la predizione)
- $H = \{\text{sgn}(v(x, w)), w \in \mathbb{R}^P\}$ è lo spazio delle ipotesi.



In questo caso ogni unità calcola $\phi(x) = \sigma(w_1^T x + b_1)$.

Mentre $v(x) = w_2^T \phi(x) + b_2$

Perciò $w = \{w_1, b_1, w_2, b_2\}$ (sono 25 pesi)

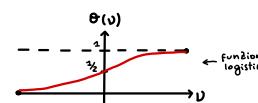
Def(funzione di risposta): chiamiamo **FUNZIONE DI RISPOSTA** la funzione $\Theta(v)$ che mappa l'uscita della rete v in un parametro di $P(y|x)$.

ES(predizione di un numero reale (regressione)):

$\Theta(v) = v$ (funzione identità), $y|x \sim N(v(x, w); 1)$ (distribuzione gaussiana normale)

ES(classificazione binaria):

$\Theta(v) = \frac{1}{1+e^{-v}} \in (0, 1)$ (funzione logistica), $y|x \sim \text{Bernoulli}(\Theta(v))$



In questo modo, il problema dell'apprendimento diventa un problema di massima verosimiglianza, in cui la rete è diventata un modello di una distribuzione che dipende da dei parametri (per trovare i migliori parametri si usa la massima verosimiglianza).

Si puo' dimostrare che la massima verosimiglianza è equivalente alla minimizzazione di una loss empirica, se tale funzione loss è stata scelta bene.

N.B: • nei problemi di regressione scegliamo la loss $\ell(\theta, y) = (\theta - y)^2$ (detta square loss);

• nei problemi di classificazione binaria scegliamo la loss $\ell(\theta, y) = y \log \theta + (1-y) \log(1-\theta)$ (detta Binary Cross Entropy).

Una volta scelta la loss in modo opportuno, allora il valore che vogliamo minimizzare sarà $L(w) := \frac{1}{n} \sum_{i=1}^n \ell(\theta(x^{(i)}, w), y^{(i)})$, (loss empirica). Perciò, il problema dell'apprendimento si riduce a minimizzare la loss empirica, cioè $\min_w L(w)$.

N.B: in generale $L(w)$ non è una funzione convessa e nella pratica non è neanche localmente convessa (perciò non possiamo applicare metodi di ottimizzazione convenzionali).

Per studiare tale minimizzazione utilizzeremo il metodo del gradiente stocastico (assomiglia al perceptron)

In particolare, la regola di aggiornamento sarà $w_{t+1} = w_t - \gamma \nabla_w L^{(i)}(w_t)$ dove ∇_w è il gradiente (ovvero $\nabla_w L = [\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \dots, \frac{\partial L}{\partial w_n}]$).

L'indice i viene scelto in modo casuale ($i \sim \text{Categorical}(1, \dots, n)$). L'iperparametro γ è detto learning rate.

Vediamo come calcolare il gradiente in un caso semplificato (cioè quando non ci sono unità nascoste):

$$\begin{array}{c} w_1 \quad w_2 \quad w_p \\ \text{---} \quad \text{---} \quad \text{---} \\ x_1 \quad x_2 \quad x_p \end{array} \quad \frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial \theta} \frac{\partial \theta}{\partial v} \frac{\partial v}{\partial w} \quad (\text{chain rule della matematica})$$

• Se prendiamo $\ell = y \log \theta + (1-y) \log(1-\theta)$, allora $\frac{\partial \ell}{\partial \theta} = \frac{y}{\theta} - \frac{1-y}{1-\theta} = \frac{\gamma-\theta}{\theta(1-\theta)}$

• se prendiamo $\theta = \frac{1}{1+e^{-v}}$, allora $\theta' = \theta(1-\theta)$

• $\frac{\partial v}{\partial w} = \frac{\partial}{\partial w} (w^T x + b) = x$

Perciò $\frac{\partial \ell}{\partial w} = \frac{\gamma-\theta}{\theta(1-\theta)} \theta(1-\theta) x = x(\gamma-\theta)$

Di conseguenza, la regola di aggiornamento sarà $w_{t+1} = w_t - \gamma(x(\gamma-\theta))$

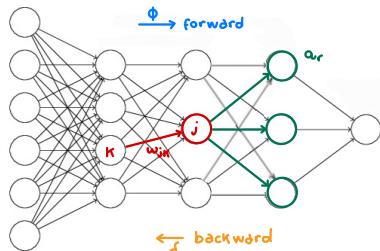
Per calcolare il gradiente $\nabla_w L(w)$ per il Multilayered Perceptron possiamo utilizzare l'algoritmo di **BACKPROPAGATION**. L'idea è quella di effettuare uno scambio di messaggi tra i nodi del grafo, ogni volta che viene modificato un peso (poiché quando si modifica un peso, questo modificherà tutte le unità dei layer superiori fino ad arrivare all'output).

N.B.: per una classificazione multiclasse sono necessari più iperpiani per suddividere le classi (ad esempio per 3 classi servono almeno 2 iperpiani).

In particolare, nel calcolo di v si aggiunge una unità per ogni classe. Ovvero $v(x) \in \mathbb{R}^K$ con $K = \#$ classi.



Generalizziamo il processo. Consideriamo il peso w_{jk} corrispondente all'arco che unisce le unità j ed k .



$$\alpha_j = \sum_{k \in P_{\text{out}}(j)} w_{jk} \phi_k + b_j \quad \text{FORWARD PROPAGATION} \quad (\text{si segue l'ordine topologico (cioè si parte dall'input)})$$

$$\phi_j = \sigma(\alpha_j)$$

Per calcolare il gradiente, consideriamo la loss su un singolo esempio. Allora, utilizzando la chain rule:

$$\frac{\partial \ell}{\partial w_{jk}} = \frac{\partial \ell}{\partial \alpha_j} \frac{\partial \alpha_j}{\partial w_{jk}}$$

$$\text{Definiamo } \delta_j := \frac{\partial \ell}{\partial \alpha_j} \text{ detto } \text{ERRORE GENERALIZZATO}.$$

si espande ancora
poiché modifica tutti
i successori di j

$$\text{Tale } \delta_j \text{ può essere calcolato in modo ricorsivo } \delta_j = \frac{\partial \ell}{\partial \alpha_j} = \frac{\partial \ell}{\partial \phi_j} \frac{\partial \phi_j}{\partial \alpha_j}.$$

$$\text{Perciò } \frac{\partial \ell}{\partial \phi_j} = \sum_r \frac{\partial \ell}{\partial \alpha_r} \frac{\partial \alpha_r}{\partial \phi_j} \text{ dove } \alpha_r = \sum_j w_{rj} \phi_j + b_r. \text{ Perciò } \frac{\partial \ell}{\partial \phi_j} = \sum_r \delta_r w_{rj}.$$

$$\text{Otteniamo che } \delta_j = \left(\sum_r \delta_r w_{rj} \right) \sigma'(\alpha_j) \quad \text{BACKWARD PROPAGATION} \quad (\text{si segue l'ordine topologico inverso (cioè si parte dall'output)}).$$

N.B.: otteniamo che ϕ si propaga in avanti (forward), mentre δ si propaga all'indietro (backward).

δ iniziale legato all'output

$$\text{Inoltre } \delta_0 = \frac{\partial \ell}{\partial v} = \frac{\partial \ell}{\partial \theta} \frac{\partial \theta}{\partial v} = \frac{y - \theta}{\theta(1-\theta)} \quad \theta = y - \theta \text{ con } \theta(v) = \frac{1}{1+e^{-v}}.$$

Se associamo bene la risposta alla loss (ad esempio, se la risposta è logistica allora usiamo come loss la cross-entropy (classificazione binaria), oppure se la risposta è l'identità allora usiamo come risposta la square-loss (regressione)), allora tale risultato è sempre valido.

In questo modo possiamo inizializzare la propagazione backward con δ_0 .

$$\text{Perciò il gradiente diventa } \frac{\partial \ell}{\partial w_{jk}} = \frac{\partial \ell}{\partial \alpha_j} \frac{\partial \alpha_j}{\partial w_{jk}} = \delta_j \phi_k$$

stochastic gradient descend

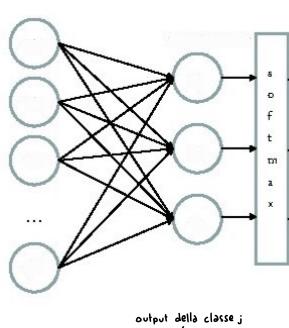
Per completare l'algoritmo di backpropagation si associa il risultato ottenuto con la regola di aggiornamento $w_{t+1} = w_t - \gamma \nabla_w \ell^{(t)}(w_t)$ detta **SGD**.

N.B.: i è un esempio campionato casualmente tra 1 ed n (cioè $i \sim \text{random}(1, n)$).

N.B.: tale algoritmo deve essere inizializzato con w_0 , ma non può essere inizializzato con il valore 0 ($w_0 \neq 0$).

Perciò, i pesi devono essere inizializzati con valori casuali e non nulli (ma è preferibile che siano vicini a 0). Un'idea può essere quella di inizializzare i pesi $w_{jk,0}$ con valori uniformi in un intervallo $(-d, d)$ con $d = \sqrt{\frac{A}{P_{\text{out}}(j) + \text{ch}(j)}}$ (A costante, $P_{\text{out}}(j)$ padri di j ed $\text{ch}(j)$ figli di j).

Per una classificazione multiclasse, dobbiamo:

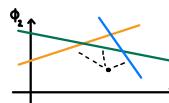


- 1) generalizzare v da uno scalare ad un vettore (v_1, v_2, \dots, v_K) dove $K = \#$ di classi.
- 2) per la funzione risposta, generalizziamo la funzione logistica, ovvero $\theta_j = \frac{e^{v_j}}{\sum_{r=1}^K e^{v_r}}$ detta **FUNZIONE SOFTMAX** (dove $\theta_j = P(y=j|x)$ ed $\sum_j \theta_j = 1$).
- 3) per la funzione loss, generalizziamo la cross-entropy, ovvero $\ell(\theta, y) = -\sum_{j=1}^K \mathbb{1}_{\{y=j\}} \log \theta_j$; detta **MULTICLASS CROSS-ENTROPY**

ES: $y=2, K=5$

$$\begin{matrix} 0 & 1 & 0 & 0 & 0 \\ \theta_1 & \theta_2 & \theta_3 & \theta_4 & \theta_5 \end{matrix} \quad \text{indicatore di } y.$$

Se prendiamo $\delta_0 = \frac{\partial \ell}{\partial v} = \mathbb{1}_{\{y=0\}} - \theta_0$ (che è la generalizzazione di $\delta_0 = y - \theta$)



si associa un iperpiano per ogni classe e si misura la distanza da ogni iperpiano. La classe associata al punto sarà quella con v più grande (direttamente proporzionale alla distanza).

DETtagli di ottimizzazione:

1) SGD è troppo rumoroso. Perciò si utilizza **MINIBATCHES**, ovvero $w_{t+1} = w_t - \gamma \nabla_w L_t$ dove $L_t = \sum_{i \in \text{MB}(t)} \ell(\theta^{(i)}, y^{(i)})$

2) Oltre ad $w_{t+1} = w_t - \gamma \nabla_w L_t$, esistono altri metodi per ottimizzare (come Adam, Momentum,...).

$\text{MB}(t)$ sottoinsieme casuale di $\{1, \dots, n\}$ di dimensione B (detta minibatch size)

ENSEMBLES: si utilizza una combinazione di classificatori che definiscono un risultato (simile alla tecnica con votazione).

Si possono realizzare diversi tipi di ensembles:

- Alberi di decisione (Random Forest): si realizza una foresta di alberi;

- Weak learners (Boosting): un weak learner è una funzione che predice correttamente con probabilità almeno $\frac{1}{2} + \gamma$ (con $\gamma > 0$).

Perciò, l'idea del Boosting è quella di utilizzare più weak learners per aumentare la probabilità di predizione corretta.

Un algoritmo che utilizza questa idea è detto **ADABOOST**. Tale algoritmo opera iterativamente sull'intero dataset (ogni iterazione sul dataset è detta **round** (simile alle epoche)). Ad ogni round viene costruito un nuovo predittore (cioè un nuovo weak learner).

Vediamo come realizzare lo pseudo codice:

ADABOOST():

for $t = 1 \dots T$ // $T = \#$ round del boosting (ad ogni round si costruisce un weak learner)

• Construct a distribution $w_t^{(i)}$ on $\{1 \dots n\}$ // $w_t^{(i)}$ indica l'importanza (o peso) dell'esempio i al round t (N.B. $\sum_{i=1}^n w_t^{(i)} = 1$ ed $w_t^{(i)} \geq 0 \forall i = 1 \dots n$).

• Construct a weak classifier $f_t: X \rightarrow \{-1, 1\}$ such that $\epsilon_t := P_{w_t} [f_t(x^{(i)}) \neq y^{(i)}] = \sum_{i=1}^n w_t^{(i)} \mathbb{1}_{\{f_t(x^{(i)}) \neq y^{(i)}\}}$ is small

Construct a final classifier f combining f_t for $t = 1 \dots T$

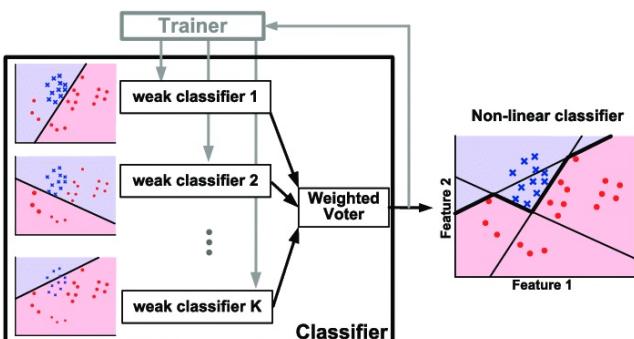
Vediamo come possiamo costruire la distribuzione $w_t^{(i)}$:

• Inizialmente (primo round) associamo lo stesso peso a tutti gli esempi, ovvero $w_1^{(i)} = \frac{1}{n}$ (uniforme);

• Al generico round $t+1$, dato $w_t^{(i)}$ (peso al round precedente) ed f_t (weak learner al passo precedente), allora applichiamo la regola di aggiornamento $w_{t+1}^{(i)} = \frac{w_t^{(i)}}{Z_t} \cdot \begin{cases} e^{-\alpha_t} & \text{se } f_t(x^{(i)}) = y^{(i)} \\ e^{\alpha_t} & \text{altrimenti} \end{cases}$ (cioè se il weak learner al passo precedente aveva predetto correttamente)

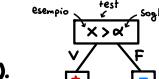
N.B: poiché $f_t: X \rightarrow \{-1, 1\}$, allora possiamo riscrivere $w_{t+1}^{(i)} = \frac{w_t^{(i)}}{Z_t} e^{-\alpha_t y^{(i)} f_t(x^{(i)})}$ dove $Z_t = \sum_{i=1}^n w_t^{(i)} e^{-\alpha_t y^{(i)} f_t(x^{(i)})}$ è un normalizzatore ed $\alpha_t = \frac{1}{2} \log \frac{1 - \epsilon_t}{\epsilon_t}$ dove $\epsilon_t = \sum_{i=1}^n w_t^{(i)} \mathbb{1}_{\{f_t(x^{(i)}) \neq y^{(i)}\}}$ è l'errore.

Costruiti i vari weak learners, possiamo creare il classificatore finale come $f = \text{sgn}(\sum_{t=1}^T \alpha_t f_t(x))$



Nell'esempio, ogni punto rappresenta un esempio (abbiamo esempi negativi in blu ed esempi positivi in rosso). La dimensione rappresenta il peso di ogni esempio.

Inizialmente tutti i pesi sono uguali (la dimensione dei punti: è la stessa). Per ogni esempio si effettua un **DECISION STUMP** (è un modello costituito da un albero di decisione ad 1 livello che permette di classificare l'esempio a seconda del suo valore (si fa un test)).



Ogni volta che un esempio è stato classificato correttamente gli viene

ridotto il peso, al contrario, se è stato classificato in modo sbagliato, gli viene aumentato il peso.

Perciò viene eseguita una seconda classificazione tenendo conto dei pesi e aggiornandoli.

Al termine avremo effettuato una serie di classificazioni e quindi mettendole insieme otteniamo la classificazione finale

Vediamo più in dettaglio lo pseudocodice di ADABOOST:

ADABoost(\mathcal{D}, T)

- 1 // $\mathcal{D} = \{(x^{(i)}, y^{(i)}) | i = 1, \dots, n\}$: dataset
- 2 // $T \in \mathbb{N}$: Rounds of boosting
- 3 Initialize $w_1^{(i)} = \frac{1}{n}$ for $i = 1, \dots, n$;
- 4 **for** $t = 1, \dots, T$
 - 5 // Train weak learner using distribution W_t
 - 6 $f_t = \text{FitWEAKLEARNER}(\mathcal{D}, W_t)$ // $f_t: \mathcal{X} \mapsto \{-1, 1\}$ (addestra il weak learner sul dataset D con una distribuzione W_t)
 - 7 $\epsilon_t = \sum_i w_t^{(i)} \mathbb{1}_{\{f_t(x^{(i)}) \neq y^{(i)}\}}$ (si misura l'accuratezza pesata di f_t)
 - 8 $\alpha_t = \frac{1}{2} \log \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$
 - 9 $w_{t+1}^{(i)} = \frac{w_t^{(i)}}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if } f_t(x^{(i)}) = y^{(i)} \\ e^{\alpha_t} & \text{otherwise} \end{cases} = \frac{w_t^{(i)}}{Z_t} e^{-\alpha_t y^{(i)} f_t(x^{(i)})}$ (si modificano i pesi)
 - 10 // Z_t normalizer ensuring $\sum_i w_t^{(i)} = 1$
 - 11 $F(x) = \sum_{t=1}^T \alpha_t f_t(x)$ // aka the margin
 - 12 $H(x) = \text{sign}(F(x))$
 - 13 **return** H

Teorema: Sia $\epsilon_t = \frac{1}{2} - \eta_t$ dove η_t è detto **EDGE** (ipotizziamo che ogni $\eta_t > \eta > 0$, poiché se $\eta_t = 0$ allora $\epsilon_t = \frac{1}{2}$ e ciò significa che stiamo prediccendo casualmente e quindi sbagliamo metà esempi (noi vogliamo fare meglio)).

Sia $\hat{R}[H] := \frac{1}{n} \sum_i \mathbb{1}_{\{H(x^{(i)}) \neq y^{(i)}\}}$ (detto **EMPIRICAL RISK**) l'errore di training.

$$\text{Allora } \hat{R}[H] \leq \frac{1}{T} 2 \sqrt{\epsilon_t (1 - \epsilon_t)} = \frac{1}{T} 2 \sqrt{1 - 4\eta_t^2} \leq e^{-2 \sum_{t=1}^T \eta_t^2} \leq e^{-2\eta^2 T}$$

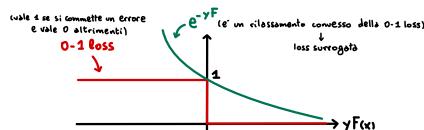
DIM:

$$\text{Osserviamo che il peso finale dell'esempio } i\text{-esimo sarà } w_T^{(i)} = \frac{1}{n} \frac{e^{-y^{(i)} \sum_{t=1}^T \alpha_t f_t(x^{(i)})}}{\sum_{t=1}^T Z_t} = \frac{1}{n} \frac{e^{-y^{(i)} F(x^{(i)})}}{\sum_{t=1}^T Z_t}.$$

$$\text{Inoltre } \sum_{i=1}^n w_T^{(i)} = 1, \text{ perciò } \sum_{i=1}^n \frac{1}{n} \frac{e^{-y^{(i)} F(x^{(i)})}}{\sum_{t=1}^T Z_t} = 1. \text{ Da ciò deriva che } \sum_{t=1}^T Z_t = \sum_{i=1}^n \frac{1}{n} e^{-y^{(i)} F(x^{(i)})}.$$

Osserviamo anche che $\hat{R}[h] := \frac{1}{n} \sum_i \mathbb{1}_{\{y^{(i)} F(x^{(i)}) < 0\}} \leq \frac{1}{n} \sum_i e^{-y^{(i)} F(x^{(i)})} = \prod_{t=1}^T z_t$

Questo passaggio e' importante perche' permette di minimizzare la 0-1 loss.



In generale, il problema di minimizzare la 0-1 loss e' intrattabile, ma, notando che e' limitata superiormente dalla funzione e^{-yF} allora possiamo studiarlo (si utilizza una **LOSS SURROGATA** (loss esponenziale)).

Percio' ADABOOST e' un minimizzatore (**GREADY**) della loss esponenziale.

Studiamo il normalizzatore $z_t = \sum_{i=1}^n w_t^{(i)} e^{-\alpha_t y^{(i)} F_t(x^{(i)})} = \sum_{i \in \text{incorrect}} w_t^{(i)} e^{-\alpha_t} + \sum_{i \in \text{correct}} w_t^{(i)} e^{-\alpha_t} = e^{-\alpha_t} \sum_{i \in \text{incorrect}} w_t^{(i)} + e^{-\alpha_t} \sum_{i \in \text{correct}} w_t^{(i)} = e^{-\alpha_t} \sum_i w_t^{(i)} + e^{-\alpha_t} (1 - \sum_i w_t^{(i)})$.

$$\text{Otteniamo che } z_t = \sum_i w_t^{(i)} \sqrt{\frac{1-\epsilon_t}{\epsilon_t}} + (1-\sum_i w_t^{(i)}) \sqrt{\frac{\epsilon_t}{1-\epsilon_t}} = 2\sqrt{\epsilon_t(1-\epsilon_t)}$$

Studiamo il minimo calcolando la derivata $\frac{\partial z_t}{\partial \alpha_t} = 0$, ovvero $\frac{\partial z_t}{\partial \alpha_t} = \sum_i w_t^{(i)} e^{-\alpha_t} - (1-\sum_i w_t^{(i)}) e^{-\alpha_t} = 0$. Otteniamo che $\alpha_t = \frac{1}{2} \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$.

Questa condizione permette di minimizzare il prodotto degli z_t e quindi permette di minimizzare la loss esponenziale.

Inoltre, essendo $z_t = 2\sqrt{\epsilon_t(1-\epsilon_t)}$, allora, minimizzando $\epsilon_t \in [0, \frac{1}{2}]$, minimizziamo anche z_t .

CVD

Tale algoritmo, al crescere del numero di iterazioni, migliora il margine e quindi risulta essere un ottimo algoritmo.

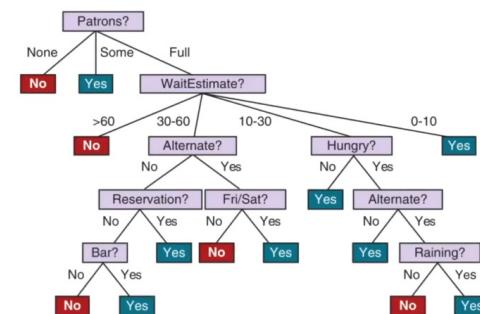
ADABOOST e' un algoritmo che fa parte della classe di algoritmi detti **GRADIENT BOOSTING**.

DECISION TREES (DT) (alberi di decisione):

ES (ristorante):

Vogliamo predire se conviene attendere di sedersi al tavolo di un ristorante o se andare via.

Example	Input Attributes										Output	
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est		
X ₁	Yes	No	No	Yes	Some	\$\$\$	No	Yes	French	0-10	y ₁ = Yes	
X ₂	Yes	No	No	Yes	Full	\$	No	No	Thai	30-60	y ₂ = No	
X ₃	No	Yes	No	No	Some	\$	No	No	Burger	0-10	y ₃ = Yes	
X ₄	Yes	No	Yes	Yes	Full	\$	Yes	No	Thai	10-30	y ₄ = Yes	
X ₅	Yes	No	Yes	No	Full	\$\$\$	No	Yes	French	>60	y ₅ = No	
X ₆	No	Yes	No	Yes	Some	\$\$	Yes	Yes	Italian	0-10	y ₆ = Yes	
X ₇	No	Yes	No	No	None	\$	Yes	No	Burger	0-10	y ₇ = No	
X ₈	No	No	No	Yes	Some	\$\$	Yes	Yes	Thai	0-10	y ₈ = Yes	
X ₉	No	Yes	Yes	Yes	No	Full	\$	Yes	No	Burger	>60	y ₉ = No
X ₁₀	Yes	Yes	Yes	Yes	Full	\$\$\$	No	Yes	Italian	10-30	y ₁₀ = No	
X ₁₁	No	No	No	No	None	\$	No	No	Thai	0-10	y ₁₁ = No	
X ₁₂	Yes	Yes	Yes	Yes	Full	\$	No	No	Burger	30-60	y ₁₂ = Yes	



Percio' si effettua una partizione ricorsiva dello spazio delle istanze sulla base del valore di un attributo. Le foglie rappresentano una decisione (NO=andiamo via, YES=attendiamo). Tale decisione viene effettuata valutando gli attributi.

L'albero di decisione, oltre a predire, mostra perche' sono state prese determinate decisioni. Ovvero, gli alberi di decisione sono interpretabili. Esistono altri algoritmi che implementano gli alberi di decisione.

Lo spazio delle ipotesi per gli alberi di decisione e' molto grande. Infatti, dati p attributi booleani, allora possiamo realizzare 2^p funzioni

x_1	x_2	\dots	x_p	f
0	0	\dots	0	1
0	0	\dots	1	0
\vdots	\vdots	\vdots	\vdots	\vdots
1	1	\dots	0	0

\hookrightarrow possiamo realizzare 2^p funzioni booleane distinte poiche' ogni funzione puo' assumere un certo valore (0 ed 1) in ogni riga (che sono 2^p per p attributi). Per ognuna di queste funzioni e' sempre possibile realizzare un albero di decisione e quindi lo spazio delle ipotesi \mathcal{H} e' dato da tutte le funzioni

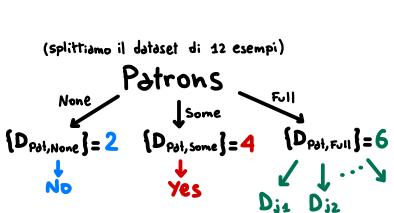
L'idea e' quella di trovare l'albero piu' piccolo possibile (altrimenti rischiamo di fare overfitting). Tale problema risulta NP-completo. Un algoritmo che puo' costruire questi alberi e' detto **GREADY CONSTRUCTION**. L'idea e' quella di usare come radice dell'albero l'attributo che riduce il piu' possibile l'impurita' (cioe' tutte le classi sono ugualmente rappresentate (ad esempio, un dataset che possiede 50% positivi e 50% in una classificazione binaria, allora risulta essere massimamente impuro. Al contrario, se il dataset possiede tutti positivi o tutti negativi, allora e' puro)).

Si introduce una quantita' detta **GAIN** che dipende dal dataset D e dall'attributo j. Cioe' $\text{Gain}(D, j) = \text{cost}(D) - \sum_{v \in \text{Dom}(j)} \frac{|\mathcal{D}_{v,j}|}{|\mathcal{D}|} \text{cost}(\mathcal{D}_v)$ dove $j = 1, \dots, p$ ed $\mathcal{D}_{v,j} = \{(x, y) \in \mathcal{D} | x_j = v\}$ (ad esempio $\mathcal{D}_{\text{Full}, \text{Patrons}}$ sono tutte le righe del dataset che hanno il valore Full per l'attributo Patrons).

Quindi, per trovare l'attributo migliore dobbiamo massimizzare il Gain, cioe' $j^* = \max\{\text{Gain}(D, j)\}$.

Nell'esempio, per l'attributo Patrons avremmo:

Example	Input Attributes										Output
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	
X ₁	Yes	No	No	Yes	Some	\$\$\$	No	Yes	French	0-10	y ₁ = Yes
X ₂	Yes	No	No	Yes	Full	\$	No	No	Thai	30-60	y ₂ = No
X ₃	No	Yes	No	No	Some	\$	No	No	Burger	0-10	y ₃ = Yes
X ₄	Yes	No	Yes	Yes	Full	\$	Yes	No	Thai	10-30	y ₄ = Yes
X ₅	Yes	No	Yes	No	Full	\$\$\$	No	Yes	French	>60	y ₅ = No
X ₆	No	Yes	No	Yes	Some	\$\$	Yes	No	Italian	0-10	y ₆ = Yes
X ₇	No	Yes	No	No	None	\$	Yes	No	Burger	0-10	y ₇ = Yes
X ₈	No	Yes	No	Yes	Some	\$	No	Yes	Thai	0-10	y ₈ = Yes
X ₉	No	Yes	Yes	Yes	Full	\$	Yes	No	Burger	>60	y ₉ = Yes
X ₁₀	Yes	Yes	Yes	Yes	Full	\$\$\$	No	Yes	Italian	10-30	y ₁₀ = No
X ₁₁	No	No	No	No	None	\$	No	No	Thai	0-10	y ₁₁ = No
X ₁₂	Yes	Yes	Yes	Yes	Full	\$	No	No	Burger	30-60	y ₁₂ = Yes



Notiamo che in "Some" abbiamo come classificazione sempre "Yes", perciò possiamo inserire una foglia. Allo stesso modo, per "None" abbiamo sempre il valore "No", perciò possiamo inserire una foglia. Al contrario, per il valore "Some" possiamo avere più classificazioni, perciò si splitta lo spazio delle istanze ricorsivamente su un altro attributo (scelto in modo da costruire l'albero più piccolo).

indica i valori che puo' assumere j (se e' booleano sono 2 valori (true e false)). Nel caso di Patrons abbiamo 3 possibili valori (None, Full, Some)

Vediamo lo pseudocodice:

DT-Learn(D, attributes):

```

if |D|=0 return Leaf(Most Common Class) // se il dataset D è vuoto si ritorna una classe di default
elif all examples in D have y(i)=y return Leaf(y) // se tutti i punti nel dataset hanno la classe y(i), allora si ritorna la foglia y
elif |attributes|=0 return Leaf(Most Common Class) // se il numero di attributi rimasti da guardare è 0, allora si ritorna una classe di default
else j* = argmaxj ∈ attributes {Gain(D, j)}
    tree = new DT with root j*
    for each v ∈ Dom(j*)
        Dv = {(x, y) ∈ D | xj* = v}
        subtree = DT-Learn(Dv, attributes - {j*})
        add subtree as a child of tree
    return tree

```

Vediamo come misurare l'impurità tramite la funzione di costo (Cost(D)).

Consideriamo il dataset $D = \{(x^{(i)}, y^{(i)}) | i=1, \dots, n\}$ con $y^{(i)} \in \{1, 2, \dots, K\}$ (dove K è il numero di classi). Chiamiamo $\hat{P}_k = \frac{1}{n} \sum_{i=1}^n \mathbb{1}_{\{y^{(i)}=k\}}$ la frazione di esempi nella classe k, e chiamiamo $k^* = \arg\min_k \hat{P}_k$ la classe più frequente.

Allora, le possibili misure di impurità sono:

1) Errore di classificazione: $1 - \hat{P}_{k^*}$ (rappresenta il numero di errori commessi se assegnamo la classe più frequente). Tale misura, però, non risulta essere discriminativa, perciò non viene usata.

2) Indice di Gini: $\sum_{k=1}^K \hat{P}_k (1 - \hat{P}_k)$ (ad esempio:

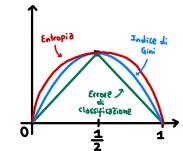
+	-	$0,8(1-0,8) + 0,2(1-0,2) = 0,16 + 0,16 = 0,32$
80%	20%	

+	-	$0,5(1-0,5) + 0,5(1-0,5) = 0,25 + 0,25 = 0,5$
50%	50%	

+	-	$0,99(1-0,99) + 0,01(1-0,01) = 0,0099 + 0,0001 = 0,0100$
99%	1%	

)

nell'ultimo caso si ha un valore più piccolo ed indica un dataset più puro



3) Entropia: $-\sum_{k=1}^K \hat{P}_k \log_2 \hat{P}_k$ (risulta essere leggermente più discriminativa rispetto all'indice di Gini).

Overfitting negli alberi di decisione (DT): è dovuto al fatto che gli alberi hanno un'alta varianza, ovvero modificando leggermente il training set si può provocare un'alta perturbazione dell'albero risultante.

Per risolvere questo problema si può calcolare la media sui risultati ottenuti. Perciò, si costruiscono degli ensembles (come ADABoost).

La tecnica più usata, però, risulta essere **RANDOM FOREST**, in cui si prende un numero casuale di alberi ed il risultato viene calcolato per votazione a maggioranza. Per realizzare ogni albero, tale algoritmo:

1) Prende insiemi casuali di dati. Questa tecnica è detta **BAGGING**, la quale prende n volte un campione dal dataset.

Lo pseudocodice del Bagging sarà:

BAGGING(D):

repeat n times

 Sample from D

2) Randomizza gli attributi visibili (invece di prendere tutto l'insieme di attributi (di dimensione p), se ne prende un sottoinsieme casuale (solitamente di dimensione 1-p)).