

Intelligenza Artificiale

Riassunti tratti dal libro *Artificial Intelligence: A Modern Approach*
a.a. 2017/2018
Matteo Petrone

Risolvere Problemi con la Ricerca

In questo capitolo si introduce un particolare agente basato su obiettivi, detto **agente risolutore dei problemi**. Questi utilizzano rappresentazioni **atomiche**, in cui gli stati del mondo sono considerati come entità prive di struttura visibile agli algoritmi.

Vengono di seguito analizzati algoritmi di ricerca: li definiamo **non informati** se non possiedono alcuna informazione sul problema oltre alla sua definizione, **informati**, invece, se dispongono di alcune informazioni su dove cercare la soluzione permettendo così di ottenere prestazioni migliori.

Si ipotizza inizialmente che l'ambiente sia **osservabile**, **deterministico** (ogni azione ha un solo risultato) e **discreto** (non è necessaria la probabilità). Un agente di questo tipo è detto ad **anello aperto**.

Gli agenti intelligenti devono massimizzare il loro indice di prestazione. Questa operazione risulta più facile se l'agente sceglie un **obiettivo**. Gli obiettivi, in generale, aiutano a organizzare il comportamento dell'agente limitando i suoi scopi e quindi le sue azioni. La formulazione dell'obiettivo è quindi il primo passo per la risoluzione del problema.

Si hanno poi 3 fasi principali:

- **Formulazione del problema:** fase in cui, fissato l'obiettivo, si decide quali stati o azioni considerare.
- **Ricerca:** processo che cerca una sequenza di azioni che raggiunge l'obiettivo.
- **Esecuzione:** un algoritmo di ricerca prende un problema come *input* e restituisce una *soluzione*, ovvero una sequenza di azioni. Una volta trovata la soluzione l'agente esegue le azioni determinate.

Un problema può essere definito formalmente da cinque componenti:

- Lo **stato iniziale s** in cui si trova l'agente. Ad esempio *In(Arad)*.
 - Una descrizione delle **azioni** possibili dell'agente: dato uno stato s, *Actions(s)* restituisce tutte le azioni eseguibili in s. Ognuna di queste dice **applicabile** in s.
 - Una descrizione di ciò che è compiuto da ciascuna azione. Tale descrizione prende il nome di **modello di transizione** ed ha una funzione *Result(s, a)* che restituisce lo stato risultante dall'esecuzione dell'azione a nello stato s.
- Insieme, stato iniziale, azioni e modello di transizione definiscono lo **spazio degli stati**. Questo è rappresentabile attraverso un grafo orientato (**Graph Search**).
- Il **goal test** determina semplicemente se uno stato è uno stato-obiettivo.
 - La funzione **path cost**, che assegna un costo numerico ad ogni cammino. Lo **step cost**, ovvero il costo di passo per l'azione a che porta dallo stato s allo stato s', si indica $c(s, a, s')$.

I cinque componenti appena descritti definiscono l'input di un problema, che rende in output una sequenza di azioni che portano dallo stato iniziale allo stato obiettivo.

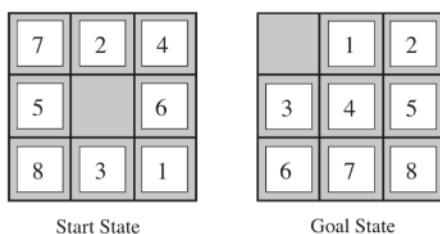
Formulazione dei Problemi

Di solito si usa non tenere in considerazione, almeno nei modelli semplici, molte variabili del modo reale. Nell'esempio in cui si debba andare in macchina da Arad a Bucarest, ogni stato (ovvero ogni città attraversata) non tiene conto di moltissime condizioni reali (radio in auto, polizia, compagni di viaggio), così come nelle azioni non si considerano carburante, meteo, autovelox. Questo processo di rimozione dei dettagli prende il nome di **astrazione**. Questa si dice *valida* se possiamo espandere ogni situazione astratta in una reale, e si dice *utile* se semplifica molto il problema originale.

Vediamo alcuni esempi di formulazione del problema.

La risoluzione dei problemi è stata applicata a molti problemi giocattolo, con l'intento di mettere alla prova diversi metodi risolutivi.

Si consideri ad esempio il **rompicapo a 8 tasselli**.



Lo scopo è raggiungere ovviamente il *Goal State* in figura, a partire da uno stato iniziale come, ad esempio, quello nella parte sinistra della figura.

Si consideri questa formulazione del problema:

- **Stati**: ogni stato specifica la posizione di ciascuno degli otto tasselli e dello spazio vuoto.
- **Stato Iniziale**: qualunque stato può essere uno stato iniziale.
- **Azioni**: la formulazione più semplice consiste nell'assegnare le azioni allo spazio vuoto che può andare *Su*, *Giù*, *Dx*, *Sx*. A seconda della sua posizione, si considerano sottoinsiemi.
- **Modello di Transizione**: dato uno stato e un'azione, restituisce lo stato risultante.
- **Goal Test**: determina se lo stato corrente è l'obiettivo.
- **Path Cost**: ogni passo costa 1, il cammino sarà la somma dei passi.

Il rompicapo ha $\frac{9!}{2} = 181440$ stati raggiungibili ed è facilmente raggiungibile.

Lo scopo del **problema delle 8 regine** è di piazzare sulla scacchiera 8 regine in modo che non si attacchino reciprocamente.

Vi sono due tipi di formulazione per questo problema: una **formulazione incrementale** che utilizza operatori che estendono progressivamente la descrizione dello stato a partire dallo stato vuoto; quindi ogni azione aggiunge una regina alla scacchiera.

Una **formulazione a stato completo** inizia con le 8 regine già sulla scacchiera e le sposta per ottenere l'obiettivo.

Vediamo una formulazione incrementale:

- **Stati**: ogni piazzamento sulla scacchiera da 0 a 8 regine è uno stato.
- **Stato Iniziale**: scacchiera vuota.
- **Azioni**: aggiunta di una regina in una casella vuota.
- **Modello di Transizione**: restituisce la scacchiera con una regina in più nella casella specificata.

- **Goal Test:** scacchiera con 8 regine e nessuna attaccata.

In questa formulazione abbiamo $64 \cdot 63 \cdots 57 \approx 1,8 \times 10^{14}$ possibili sequenze da investigare.

Una formulazione migliore sarebbe quella in cui si proibisce il posizionamento in una casella già sotto attacco. Migliorando la formulazione:

- **Stati:** configurazioni di n regine ($0 \leq n \leq 8$), una per colonna, a partire da sinistra, in modo tale che nessuna regina ne attacchi un'altra.
- **Azioni:** aggiungere regina nella prima colonna libera a partire da sinistra, in modo tale che non attacchi o non sia attaccata.

Questa variazione nella formulazione riduce lo spazio degli stati da $1,8 \times 10^{14}$ a soli 2.057.

Cercare le soluzioni

Una **soluzione** di una ricerca è una sequenza di azioni. Le possibili sequenze di azioni a partire dallo stato iniziale rappresentano un **albero di ricerca**: lo stato iniziale corrisponde alla radice, le azioni rappresentate dai rami e i nodi che corrispondono agli stati (alternativamente possiamo dire che i nodi rappresentano sequenze di azioni, quindi un albero avrebbe infiniti nodi).

Supponiamo che lo stato iniziale sia *In(Arad)*: innanzitutto si controlla se si tratta di uno stato obiettivo, poi, in caso non sia uno stato obiettivo, si deve considerare di intraprendere varie azioni espandendo lo stato attuale e generare un nuovo spazio degli stati. Di tutte le possibilità si decide quale espandere ulteriormente.

L'insieme dei nodi foglia che possono essere espansi ulteriormente è detto **frontiera**.

Il processo di espansione della frontiera continua finché non vi sono più stati da espandere. Tutti gli algoritmi di ricerca sono più o meno simili in questo, si diversificano solo per la **strategia di ricerca** (metodo di scelta del successivo nodo da espandere).

Per tenere traccia dell'albero di ricerca costruito (**bookkeeping**) è necessaria una struttura dati. Per ciascun nodo n si ha:

- $n.State$: stato dello spazio degli stati a cui corrisponde il nodo.
- $n.Father$: nodo padre
- $n.Action$: l'azione applicata al padre per ottenere il nodo n .
- $n.Path_Cost$: il costo, denotato con $g(n)$, del cammino che va dalla radice a n .

La frontiera viene memorizzata attraverso una **coda con priorità**.

Possiamo valutare l'efficienza di un algoritmo secondo quattro parametri:

- **Completezza:** l'algoritmo garantisce di trovare una soluzione, se questa esiste?
- **Ottimalità:** trova la soluzione ottima? (Quella con il costo minore)
- **Complessità Temporale:** quanto impiega?
- **Complessità Spaziale:** quanta memoria occupa?

Nell'AI, poiché l'albero è spesso rappresentato implicitamente dallo stato iniziale, dalle azioni, dal modello di transizione ed è spesso infinito, la complessità si esprime usando tre quantità: b è il **branching factor**: numero massimo di figli di un nodo; d è la **depth**: profondità del nodo obiettivo più vicino allo stato iniziale (in riferimento all'albero che si genera); m è la **lunghezza massima** dei cammini nello spazio degli stati.

Il **tempo** si misura spesso con il numero di nodi generati durante la ricerca e lo **spazio** con il numero massimo di nodi memorizzati in memoria.

Per valutare l'efficienza di un algoritmo possiamo considerare solamente il **costo di ricerca**, che tipicamente dipende dalla complessità temporale.

Strategie di Ricerca non Informata

Vi sono varie strategie che ricadono nella ricerca *non informata* (**blind search**). Con il termine “non informata” si intende che tutte le strategie di ricerca non dispongono di informazioni aggiuntive sui nodi oltre a quelle fornite nella definizione del problema: possono solo espandere nuovi nodi e distinguerli dai nodi obiettivo.

Tutte le strategie di ricerca si distinguono per l'*ordine* con il quale vengono espansi i nodi.

La **ricerca in ampiezza** è una strategia in cui si espande la radice, poi i nodi figli e poi ancora i figli. Ad ogni livello di profondità tutti i nodi devono essere espansi. I nodi da espandere vengono gestiti attraverso una coda FIFO (gestione della frontiera).

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Valutando la complessità si nota che è completo: se il nodo obiettivo si trova a profondità d , l’algoritmo lo troverà a profondità d dopo aver esplorato tutti i nodi precedenti. Considerando la complessità spaziale e temporale non sembra essere molto efficiente: Se la soluzione è a profondità d avrà $O(b^d)$ (anche per quella spaziale perché la frontiera contiene b^d elementi).

La **ricerca in profondità** opera nel seguente modo: espande tutti i nodi a partire da un nodo padre e, diversamente dalla BFS, non li controlla tutti, ma controlla solo il primo, che a sua volta sarà espanso. La complessità temporale (utilizzando un albero piuttosto che un grafo) è $O(b^m)$, mentre quella spaziale migliora molto diventando $O(b \cdot m)$ poiché la DFS memorizza solo un percorso dalla radice ad una foglia, insieme ai nodi “fratelli” al nodo appena espanso poiché devono essere anch’essi esplorati. Dopo aver esplorato un nodo, questo può essere eliminato una volta esplorati tutti i suoi figli.

Per migliorare il costo temporale della DFS, essendo questa disastrosa nel caso di albero con profondità infinita, posso pensare di fornirle un limite di profondità che deve considerare nell’espansione dei nodi. Ad esempio: con limite ℓ , Depth-Limited-Search(*problem*, ℓ) chiamerà DFS modificata che terrà conto del limite ℓ . Con questo metodo la complessità temporale diventa $O(b^\ell)$, quella spaziale $O(b \cdot \ell)$. Questo algoritmo sarà completo se $d \leq \ell$, e ottimo se $d = \ell$.

La **ricerca ad approfondimento iterativo** è una strategia generale usata spesso in combinazione con la DFS per trovare il limite ideale per quest’ultima. Il limite viene incrementato di 1 ad ogni iterazione finché non viene trovato un nodo obiettivo, ovvero finché ℓ non raggiunge il valore d . Ha requisiti di memoria modesti: $O(b \cdot d)$, è completa in caso di *branching factor* finito e spazio degli stati finito, ottima quando il costo di cammino è funzione non decrescente della profondità del nodo.

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

La **ricerca a costo uniforme** risulta ottima per qualunque path-cost (la DFS e Iterative-Deepening è ottima solo quando il costo di passo è costante). Invece di espandere il nodo n meno profondo, espande il nodo con il *minimo costo di cammino* $g(n)$. Questo avviene memorizzando la frontiera in una *coda di priorità* ordinata secondo g .

Oltre al tipo di coda utilizzata, vi sono altre due differenze rispetto alla BFS:

1. Il test obiettivo è applicato ad un nodo quando questo è selezionato per l'espansione e non quando è generato per la prima volta (un nodo appena generato potrebbe trovarsi in un cammino sub-ottimo).
2. Si aggiunge un test nel caso in cui si sia trovato un cammino migliore per raggiungere un nodo nella frontiera.

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

```

node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
frontier  $\leftarrow$  a priority queue ordered by PATH-COST, with node as the only element
explored  $\leftarrow$  an empty set
loop do
  if EMPTY?(frontier) then return failure
  node  $\leftarrow$  POP(frontier) /* chooses the lowest-cost node in frontier */
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  add node.STATE to explored
  for each action in problem.ACTIONS(node.STATE) do
    child  $\leftarrow$  CHILD-NODE(problem, node, action)
    if child.STATE is not in explored or frontier then
      frontier  $\leftarrow$  INSERT(child, frontier)
    else if child.STATE is in frontier with higher PATH-COST then
      replace that frontier node with child
```

La ricerca a costo uniforme espande i nodi in ordine di costo del loro cammino ottimo. È completo e ottimo.

Strategie di ricerca informata o euristica

Vediamo come una strategia di **ricerca informata** che sfrutta conoscenza specifica del problema possa trovare soluzioni in modo più efficiente di una strategia non informata.

L'approccio generale che verrà preso in considerazione è la **ricerca best-first**. Si tratta di un'istanza di una generica ricerca su grafo nella quale, però, il nodo da espandere viene scelto attraverso una **funzione di valutazione** $f(n)$. Questa funzione viene costruita come stima di costo, verrà perciò selezionato prima il nodo con valore più piccolo.

La sua implementazione è identica a quella della ricerca a costo uniforme (vedi sopra).

La scelta di f determina la strategia di ricerca. La maggior parte degli algoritmi best-first include come componente di f una **funzione euristica** denominata $h(n)$: costo stimato del cammino più conveniente dallo stato del nodo n a uno stato obiettivo.

Le funzioni euristiche sono la forma più comune per aggiungere conoscenza all'algoritmo di ricerca. Consideriamo le funzioni euristiche come funzioni non negative e tali che se n è nodo obiettivo, allora $h(n) = 0$.



Vediamo ora due modi di usare l'informazione euristica per guidare la ricerca.

La **ricerca best-first greedy** (“golosa”) cerca sempre di espandere il nodo più vicino all’obbiettivo, sulla base del fatto che è probabile che questo porti alla soluzione più velocemente. Di conseguenza la funzione di valutazione può avere come sola componente la funzione euristica $f(n) = h(n)$. La golosità dell’algoritmo implica la non ottimalità di quest’ultimo.

La forma più diffusa di ricerca best-first è la **ricerca A***. In questa la valutazione dei nodi viene eseguita combinando $g(n)$, il costo per raggiungere il nodo n , e $h(n)$, il costo per andare da lì all’obbiettivo. Quindi: $f(n) = g(n) + h(n)$.

Di conseguenza, possiamo dire che $f(n)$ rappresenta il *costo stimato della soluzione più conveniente che passa per n*.

Se vogliamo trovare la soluzione meno costosa, è quindi ragionevole trovare il nodo con il valore $g(n) + f(n)$ più piccolo.

A patto che la funzione euristica $h(n)$ rispetti certe condizioni, la ricerca A* è sia *completa* che *ottima*. Vediamo quali sono queste condizioni.

Condizioni per l’ottimalità: ammissibilità e consistenza

1. La prima condizione richiesta per l’ottimalità è che $h(n)$ sia un’**euristica ammissibile**, ovvero che non sbagli mai per eccesso la stima del costo per arrivare all’obbiettivo. Poiché $g(n)$ è il costo effettivo del cammino per arrivare a n , ne consegue immediatamente che $f(n)$ non sbaglia mai per eccesso (se $h(n)$ è ammissibile) il costo di una soluzione.
2. La seconda condizione, leggermente più forte, denominata **consistenza** è richiesta per le applicazioni di A* di ricerca su grafo: un’euristica $h(n)$ si dice consistente se, per ogni nodo n e ogni successore n' di n generato da un’azione a , il costo stimato per raggiungere l’obbiettivo partendo da n non è superiore al costo di passo per arrivare da n a n' sommato al costo stimato per raggiungere l’obbiettivo da n' , ovvero se: $h(n) \leq c(n, a, n') + h(n')$.

Si noti che non è altro che una disuguaglianza triangolare.

La consistenza implica l’ammissibilità, ma non vale il viceversa.

Ottimalità di A*

Come abbiamo visto, la ricerca A* su albero è ottima se $h(n)$ è ammissibile, mentre quella su grafo è ottima se $h(n)$ è consistente. Analizziamo questa seconda affermazione:

1 se $h(n)$ è consistente, i valori di $f(n)$ lungo ogni cammino sono non decrescenti.

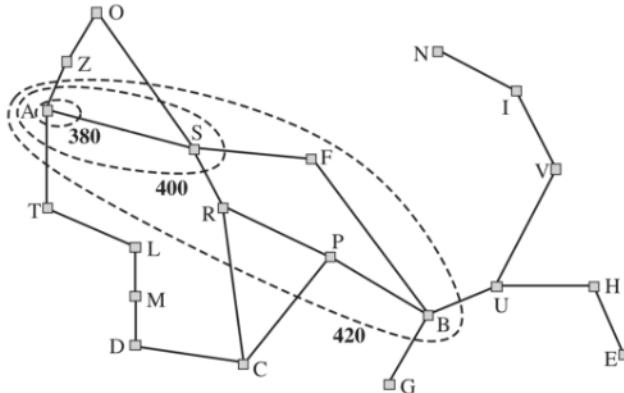
Dimostrazione: supponiamo n' successore di n , allora vale $g(n') = g(n) + c(n, a, n')$ per qualche a , e risulta:

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

2 Il passo successivo è provare che *ogni volta che A* seleziona un nodo n per l’espansione, il cammino ottimo che porta a tale nodo è stato trovato*.

Se non fosse così, ci sarebbe un altro nodo di frontiera n' sul cammino ottimo dal nodo di partenza a n , per la proprietà di separazione del grafo; poiché f è non decrescente lungo qualsiasi cammino, n' avrebbe un f -costo minore di n e sarebbe selezionato prima.

Per immaginare meglio il concetto degli f -costi non decrescenti possiamo disegnare dei confini nello spazio degli stati. La figura sotto riportata mostra un esempio: tutti gli stati nel confine etichettato con 400 hanno f -costo minore o uguale a 400. Di conseguenza, dato che A* espande sempre il nodo con f -costo minore, la ricerca A* si allargherà a ventaglio a partire dal nodo iniziale, aggiungendo nodi in fasce concentriche di f -costo crescente.



L'algoritmo A* si definisce **ottimamente efficiente**, nel senso che tra tutti quelli visti, è quello che garantisce di espandere meno nodi degli altri, pur non trascurando la soluzione ottima.

Funzioni Euristiche

Esaminiamo le funzioni euristiche per il rompicapo a 8 tasselli.

Se per trovare soluzioni più corte vogliamo usare la ricerca A*, è necessario trovare una funzione euristica che non sovrastimi mai il numero di passi che ci separano dall'obiettivo. Le due euristiche più considerate per il gioco in questione sono:

- h_1 : numero di tasselli fuori posto.
- h_2 : la somma delle distanze di tutti i tasselli dalla loro posizione corrente a quella nella configurazione obiettivo. La distanza che useremo per i tasselli, che possono muoversi in verticale o orizzontale, sarà la **distanza Manhattan**.

Entrambe queste euristiche non sovrastimano mai il costo vero della soluzione.

Teorema

Se un'euristica h_2 è più informata di un'euristica h_1 (ovvero se $h_2(n) \geq h_1(n) \forall n$), allora ogni nodo espanso da $A^*(h_2)$ è espanso anche da $A^*(h_1)$ e diciamo che h_2 **domina** su h_1 .

Questo Teorema evidenzia come euristiche più informate permettano di espandere meno nodi.

Riprendendo l'esempio del rompicapo, abbiamo che $h_2(n) \geq h_1(n) \forall n$. Quindi l'uso della distanza Manhattan permette di espandere meno nodi.

Se abbiamo 3 euristiche diverse $h_1(n), h_2(n), h_3(n)$ tali che ciascuna di esse è maggiore delle altre due, al variare di n , posso considerare una funzione euristica per ogni passo così definita: $h_4(n) = \max\{h_1(n), h_2(n), h_3(n)\}$

Per caratterizzare la qualità di un'euristica è quello di usare misure empiriche, quali:

- $N = \#(\text{nodi espansi})$

- **Penetranza** = $\frac{d}{N}$. Penetranza(oracolo) = 1

- **Branching factor effettivo b^*** : se il numero totale di nodi generati da A^* per un particolare problema è N , e la profondità è d , allora b^* è il branching factor che un albero uniforme di profondità d dovrebbe avere per contenere $N+1$ nodi.

Quindi: $N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$

Un'euristica ben progettata dovrebbe avere $b^* \approx 1$

È anche possibile derivare euristiche ammissibili dal costo della soluzione di un **sottoproblema** dato, che sarà un limite inferiore al costo totale. Questo avviene attraverso i **database pattern**, memorizzando i costi esatti di ogni possibile istanza di sottoproblema: nel nostro caso, ogni possibile configurazione dei primi quattro tasselli e dello spazio vuoto.

Fatto questo possiamo ottenere un'euristica ammissibile h_{DB} per ogni stato completo incontrato durante una ricerca, semplicemente estraendo dal database la corrispondente configurazione del sottoproblema. Il database è costruito eseguendo una ricerca all'indietro dallo stato obiettivo e memorizzando il costo di ogni nuova configurazione incontrata.

Ci si potrebbe domandare se fosse possibile sommare l'euristica ottenuta dal database che coinvolge i tasselli 1-2-3-4 e quella che coinvolge i tasselli 5-6-7-8, dato che i due sottoproblemi, apparentemente, non dovrebbero sovrapporsi. In realtà questo non è ammissibile, poiché molte mosse che fanno parte del primo database sono già coinvolte in realtà in quello 5-6-7-8. Se però prendiamo dei **database disgiunti**, questo diventa possibile.

Oltre la Ricerca Classica

Fino ad ora abbiamo analizzato problemi con ambienti osservabili, deterministici. Adesso vediamo cosa accade quando queste ipotesi vengono rilassate.

I primi due algoritmi eseguono puramente una ricerca locale nello spazio degli stati, valutando e modificando uno o più degli stati correnti.

Algoritmi di Ricerca Locale e Problemi di Ottimizzazione

Gli algoritmi visti fin qui sono progettati per esplorare sistematicamente lo spazio degli stati. Quando viene raggiunto uno *stato obiettivo*, diciamo che il *cammino* per arrivarcì è la *soluzione* del problema.

Tuttavia, il cammino non è sempre rilevante ai fini del problema. Si consideri ad esempio il problema delle 8 regine: non importa come siano gli stati intermedi e come si arrivi alla soluzione, l'importante è che si poszionino sulla scacchiera 8 regine che non si attaccino.

Se il cammino verso l'obiettivo non ha importanza è possibile considerare una diversa classe di algoritmi, che del cammino non se ne curano proprio: gli algoritmi di **ricerca locale** operano su un singolo nodo corrente e, in generale, si spostano solo su nodi ad esso adiacenti.

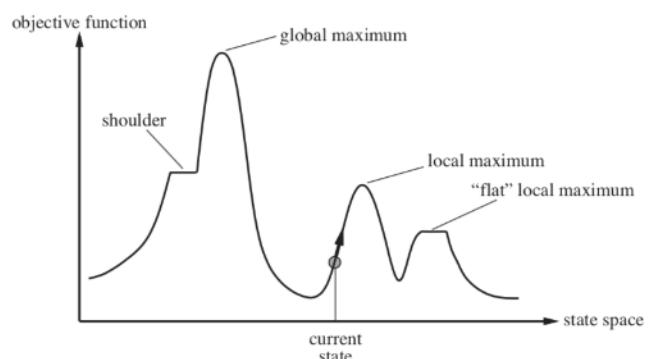
Gli algoritmi di ricerca locale hanno due importanti vantaggi: occupano poca memoria e possono trovare soluzioni ragionevoli in spazi infiniti. Tuttavia non sono buoni per trovare soluzioni ottime (sono sub-ottimali).

Sono molto utili per i **problemi di ottimizzazione**: problemi in cui lo scopo è trovare il miglior stato in accordo ad una **funzione obiettivo**.

Per comprendere bene la ricerca locale è utile considerare il **panorama dello spazio degli stati**: questo ha sia una posizione

(definita dallo stato) ed una altezza (definita da costo o funzione obiettivo). Quindi se l'altezza corrisponde al costo, dovrò trovare il minimo globale del panorama, se rappresenta la funzione obiettivo il massimo globale.

Un algoritmo di ricerca locale **completo** trova sempre un obiettivo, se esiste. Un algoritmo di ricerca locale **ottimo** trova sempre un max/min globale.



Hill Climbing

L'algoritmo di ricerca Hill Climbing segue sempre le salite più ripide nella funzione obiettivo. Si tratta di un ciclo che muove lo stato *current* continuamente verso l'alto, nella direzione dei valori crescenti. Termina quando non ha vicini di valore (obiettivo) più alto.

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

```
current ← MAKE-NODE(problem.INITIAL-STATE)
loop do
  neighbor ← a highest-valued successor of current
  if neighbor.VALUE ≤ current.VALUE then return current.STATE
  current ← neighbor
```

Gli algoritmi di ricerca locale, come questo, tipicamente usano una formulazione a stato completo. Nel problema delle 8 regine, ad esempio, lo stato iniziale del problema prevede già le 8 regine posizionate sulla scacchiera, una per colonna.

Sempre seguendo l'esempio delle regine, potremmo assumere di utilizzare come funzione euristica di costo h il numero di coppie di regine che si stanno attaccando a vicenda. Il minimo globale è ovviamente lo zero, caso della soluzione ottima.

L'algoritmo Hill Climbing è un algoritmo *goloso* in quanto “si dirige” sempre verso uno stato migliore, non curandosi di cosa vi sarà dopo.

Sfortunatamente, spesso l'algoritmo rimane bloccato, come nei seguenti casi:

- **Massimi Locali**: un massimo locale è un picco più alto degli stati vicini, ma inferiore al massimo globale.
- **Creste**: sequenza di massimi locali difficili da esplorare.
- **Plateau**: area piatta del panorama degli stati. Può essere un massimo locale piatto, oppure una **spalla**: zona piatta da cui si potrà salire ulteriormente.

Nel problema delle 8 regine, partendo da uno stato casuale, si blocca l'86% delle volte per uno di questi motivi, risolvendo solo il 14% delle istanze.

Il codice sopra riportato, in caso di plateau si blocca. Si potrebbe puntare in **mosse laterali**, nella speranza che si ratti di una spalla. Così si rischia però di cadere in un ciclo infinito. Una soluzione diffusa è in tal caso quella di imporre un limite per le mosse laterali. Questo alza le percentuali di successo fino al 94%.

Vi sono delle varianti come l'**Hill Climbing Stocastico** che sceglie a caso tra tutte le mosse che vanno verso l'alto.

Gli algoritmi visti fin qui sono **incompleti**: non è sempre detto che si raggiunga una soluzione ottima per l'obbiettivo. Si può tuttavia pensare di ad un **Hill Climbing con riavvio casuale**, che esegue Hill Climbing più volte su stati iniziali generati casualmente. Questo è completo, con probabilità di successo molto vicina a 1.

Simulated Annealing

Un algoritmo come l'Hill Climbing che rimane bloccato sui minimi locali non è certamente completo. Abbiamo visto che attraverso un'esplorazione randomizzata può diventare completo ma, di contro, sarebbe estremamente inefficiente.

Un algoritmo completo come l'Hill Climbing con esplorazione casuale che sia anche efficiente sarebbe ragionevole: ecco il **Simulated Annealing** (“tempra simulata”).

Si immagini un panorama degli stati in cui si debba minimizzare il costo. Supponiamo ad esempio di avere una pallina che corre lungo una curva (funzione di costo) e si inserisce in un minimo locale; se potessi scuotere il contenitore questa uscirebbe. Tuttavia non vogliamo scuotterlo tanto da farla uscire anche da un eventuale minimo globale.

L'Annealing è il processo con il quale i metalli vengono portati a temperature altissime e poi cristallizzati raffreddandoli gradualmente, in modo da ottenere stati a bassa energia.

Allo stesso modo si comporta l'algoritmo che, nella parte centrale, è come l'Hill Climbing anche se invece del *vicino migliore* viene scelto un successore casuale.

Se la situazione migliora, viene accettato, altrimenti lo accetta con una probabilità <1 . La probabilità decresce esponenzialmente con la cattiva mossa e con la Temperatura T che scende costantemente.

```
function SIMULATED_ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to “temperature”

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

$$P_i = \frac{1}{z} e^{-\frac{E_i}{kT}} < 1$$

Probabilità associata
alla configurazione *i*

Ricerca Local Beam

Memorizzare un solo nodo su cui lavorare può risultare una reazione estrema ai problemi di memoria. L'algoritmo **Local Beam Search** tiene traccia di k stati: comincia con k stati generati casualmente e ad ogni iterazione genera i successori di tutti i k stati. Se uno di questi è un obiettivo, termina.

Vi è una differenza sostanziale con l'Hill Climbing a riavvio casuale, poiché i processi di ricerca non sono indipendenti: ad ogni iterazione un'informazione utile è passata da un thread ad un altro che lavora in parallelo: gli stati che creano successori migliori "attraggono l'attenzione".

Nella sua forma più semplice questo algoritmo può soffrire di una "località" dei k stati somigliando quindi molto all'Hill Climbing. Una variante, la **Stochastic Beam Search**, invece di scegliere i migliori k successori, li sceglie casualmente.

Problemi di Soddisfacimento di Vincoli

Vediamo come risolvere un'ampia varietà di problemi in modo molto efficiente.

Per gli algoritmi visti fin ora, lo stato era privo di struttura interna: era una scatola nera.

Ora useremo per ciascuno stato una **rappresentazione fattorizzata**: questo possiede una serie di variabili, ognuna delle quali ha un valore assegnato. Un problema è risolto quando ogni variabile ha un valore che soddisfi tutti i vincoli su di essa.

Un problema così descritto è un **Problema di Soddisfacimento di Vincoli (CSP)**.

Un CSP è costituito da tre componenti:

- X è un insieme di variabili $\{X_1, \dots, X_n\}$.
- D è un insieme di domini $\{D_1, \dots, D_n\}$, uno per ogni variabile.
- C è un insieme di vincoli che specificano combinazioni di valori ammesse.

Ogni **dominio** D_i è costituito da un insieme di valori ammessi $\{v_1, \dots, v_k\}$ per la **variabile** X_i . Ogni **vincolo** C_i è costituito dalla coppia $\langle \text{ambito}, \text{relazione} \rangle$: *ambito* è una tupla di variabili che partecipano nel vincolo e *rel* è una relazione che definisce i valori che tali variabili possono assumere.

Ogni stato in un problema CSP è definito dall'**assegnamento** di valori ad alcune o tutte le variabili $\{X_1 = x_1, \dots, X_n = x_n\}$, con $x_i \in D_i$.

Un assegnamento può essere **consistente** se non viola alcun vincolo sulle variabili aggiornate, mentre si dice **completo** se a tutte le variabili viene assegnato un valore.

Una **soluzione** per un CSP è un assegnamento consistente e completo.

Esempio.

Consideriamo il territorio dell'Australia suddiviso in stati, come in figura. Abbiamo il compito di colorare (in rosso, verde o blu) i vari stati in modo tale che, presi due stati adiacenti, questi non abbiano lo stesso colore.

Definisco una variabile per ogni regione:

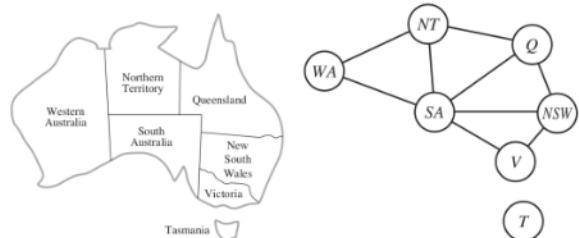
$X = \{WA, NT, Q, NSW, V, SA, T\}$. Ciascuna

variabile X_i ha un dominio $D_i = \{\text{rosso, verde, blu}\}$.

I vincoli richiedono che le regioni (quindi le variabili) vicine abbiano colore diverso:

$C = \{SA \neq WA, SA \neq NT, WA \neq NT, \dots\}$.

Può risultare utile visualizzare un CSP come un **grafo di vincoli**, come in figura, in cui i nodi sono le variabili e gli archi connettono le variabili che partecipano ad un vincolo.



Un enorme vantaggio di un CSP è la capacità nel trovare una soluzione grazie al fatto che può eliminare grandi porzioni nello spazio degli stati. Se ad esempio coloro una regione di *blu*, allora posso escludere che quelle adiacenti siano *blu*. Quindi, una volta determinato un assegnamento parziale posso immediatamente scartare raffinamenti sull'assegnamento.

Le variabili di CSP, nella forma più semplice, hanno domini **discreti e finiti**.

Un dominio discreto può essere **infinito**. In tal caso non è più possibile descrivere i vincoli enumerando tutte le possibili combinazioni ammesse: sarebbe necessario esprimere attraverso un **linguaggio di specifica dei vincoli**. Solo in caso di vincoli lineari il problema è risolvibile.

Un vincolo può essere **unario** se coinvolge solo una variabile o **binario** se ne coinvolge due. Un vincolo che coinvolge un numero arbitrario di variabili è detto **globale**.

Propagazione di Vincoli: Inferenza nei CSP

Nei CSP un algoritmo può sia cercare (come accadeva negli algoritmi di ricerca classici), sia svolgere un tipo specifico di **inferenza** (in *statistica*: procedimento di deduzione delle caratteristiche di una popolazione, a partire da una rilevazione effettuata su un campione limitato di essa, per mezzo della stima dei parametri). Questo tipo di inferenza si dice **propagazione dei vincoli**: consiste nell'utilizzare i vincoli per ridurre il numero di valori legali per una variabile, che a sua volta può ridurre i valori legali per un'altra variabile e così via. Il concetto fondamentale è la **consistenza locale**: se consideriamo ogni variabile come il nodo di un grafo ed ogni vincolo binario come un arco, il processo di *forzare la consistenza locale* in ogni parte del grafo causa l'eliminazione dei valori inconsistenti nel grafo stesso.

Esistono vari tipi di consistenza locale. Di seguito vengono analizzati.

Consistenza di Nodo

Una singola variabile si dice **nodo-consistente** se tutti i valori del suo dominio soddisfano i suoi vincoli unari. Una rete è nodo-consistente se lo sono tutte le sue variabili.

Ad esempio, se gli australiani del sud non amano il verde, la variabile SA che inizialmente ha il dominio $\{verde, blu, rosso\}$, applicatole la nodo-consistenza, riduce il dominio a $\{blu, rosso\}$.

Consistenza d'Arco

Una variabile in un CSP è **arco-consistente** se ogni valore del suo dominio soddisfa i suoi vincoli binari.

Formalmente: X_i è arco-consistente rispetto ad un'altra variabile X_j se per ogni valore nel dominio corrente D_i c'è un valore nel dominio D_j che soddisfa il vincolo binario sull'arco (X_i, X_j) .

Il più noto algoritmo di consistenza d'arco è **AC-3**:

Mantiene una *coda* degli archi da considerare, inizialmente contiene tutti quelli del CSP.

Estrae da questa un arco arbitrario (X_i, X_j) e rende X_i arco-consistente rispetto a X_j .

Se questa mossa lascia invariato D_i , l'algoritmo passa all'arco successivo; altrimenti, se

D_i si riduce, vengono aggiunti alla coda tutti gli archi (X_k, X_j) dove X_k è un vicino di X_i .

Questo perché il cambiamento di D_i potrebbe comportare ulteriori riduzioni in D_k .

Se D_i viene ridotto all'insieme vuoto, sappiamo che l'intero CSP non ha soluzione.

```

function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (Xi, Xj)  $\leftarrow$  REMOVE-FIRST(queue)
    if REVISE(csp, Xi, Xj) then
      if size of Di = 0 then return false
      for each Xk in Xi.NEIGHBORS - {Xj} do
        add (Xk, Xi) to queue
  return true

```

```

function REVISE(csp, Xi, Xj) returns true iff we revise the domain of Xi
  revised  $\leftarrow$  false
  for each x in Di do
    if no value y in Dj allows (x,y) to satisfy the constraint between Xi and Xj then
      delete x from Di
      revised  $\leftarrow$  true
  return revised

```

La complessità di AC-3 è legata al numero *n* di variabili, ciascuna con dimensione del suo dominio non superiore a *d* e con *c* vincoli binari.

Ogni arco può essere inserito nella coda al più *d* volte, perché ogni variabile ha al più *d* elementi da poter eliminare dal suo dominio.

Quindi nel caso peggiore si ha $O(cd^3)$, avendo $O(d^2)$ costo del `Revise` e $O(cd)$ il numero di chiamate del `Revise`.

È possibile estendere il concetto di arco-consistenza anche per vincoli *n-ari*: una variabile *X_i* è **arco-generalizzato consistente** rispetto ad un vincolo *n-ario* se per ogni valore *v_i* nel dominio di *X_i* esiste una tupla di valori che è membro del vincolo.

Consistenza di Cammino

La consistenza d'arco spesso non è sufficiente per trovare una soluzione. Questa restringe i domini utilizzando gli archi.

La **consistenza di cammino** restringe i vincoli binari utilizzando vincoli impliciti che sono inferiti considerando tripletti di variabili.

Formalmente: un insieme di variabili $\{X_i, X_j\}$ è path-consistent rispetto ad una terza variabile *X_m* se, per ogni assegnamento $\{X_i = a, X_j = b\}$ consistente con i vincoli su $\{X_i, X_j\}$, esiste un assegnamento di *X_m* che soddisfa i vincoli su $\{X_i, X_m\}$ e $\{X_m, X_j\}$.

K-Consistenza

Un CSP è **k-consistente** se per ogni insieme di *k* – 1 variabili e ogni loro assegnamento consistente, è sempre possibile assegnare un valore consistente a ogni *k* – esima variabile.

Un CSP è **fortemente k-consistente** se è k-consistente, (k-1)-consistente,..., 1-consistente.

Vi sono i **vincoli globali**: è un vincolo che interessa un numero arbitrario di variabili (non necessariamente tutte).

Questi vincoli si presentano molto spesso in problemi reali e possono essere gestiti con algoritmi speciali. Ad esempio il vincolo **Tuttediverse** richiede che le variabili specificate abbiano tutti valori diversi. Un semplice metodo per rilevare inconsistenze potrebbe essere: se il vincolo coinvolge *m* variabili e il loro dominio è composto da *n* possibili valori, con $m > n$, allora il vincolo non può essere soddisfatto.

L'algoritmo corrispondente a questo metodo: si rimuove prima di tutto ogni variabile coinvolta nel vincolo che abbia un solo valore e si rimuove questo valore da tutti i domini delle altre variabili coinvolte. Se ad un certo punto un dominio rimane vuoto o ci sono più variabili che valori rimasti ($m > n$), allora è stata trovata un'inconsistenza.

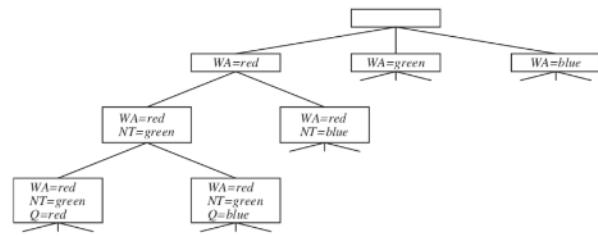
Ricerca con Backtracking per CSP

Molti CSP non possono essere risolti con la sola inferenza: è necessario avere una soluzione. Si discutono ora algoritmi che operano su assegnamenti parziali.

Si potrebbe applicare una ricerca in profondità limitata standard: uno stato sarebbe un assegnamento parziale e un'azione sarebbe l'aggiunta di $var = valore$ all'assegnamento. Per un CSP con n variabili con dominio di dimensione d si nota che qualcosa non va: il branching factor al primo livello è nd , perché uno qualsiasi dei d valori può essere assegnato a una delle n variabili \Rightarrow l'albero ha generato $n! \cdot d^n$ foglie.

Questa formulazione ha ignorato una proprietà fondamentale dei CSP: la **commutatività** (un problema è commutativo se l'ordine di applicazione delle azioni non modifica il risultato finale).

Di conseguenza per un CSP ci basta considerare *una sola* variabile in ogni livello dell'albero (vedi figura a lato: per ogni livello si aggiunge una variabile). In questo modo abbiamo solo d^n foglie.



Con **ricerca con backtracking** intendiamo una ricerca in profondità (DFS) che assegna valori a una variabile per volta e torna indietro quando non ci sono più valori legali da assegnare.

Vediamo l'algoritmo: sceglie ripetutamente una variabile non assegnata e poi prova uno ad uno tutti i valori del suo dominio, cercando una soluzione. Se viene rilevata un'inconsistenza, Backtrack restituisce un fallimento e la chiamata precedente prova un altro valore.

Nella ricerca classica non informata si sono migliorati gli algoritmi con l'aggiunta di funzioni euristiche specifiche per il dominio, derivandole dalla nostra conoscenza del problema.

I CSP possono essere risolti invece in modo efficiente anche senza possedere conoscenze specifiche sul problema, ma migliorando alcune funzioni utilizzate nell'algoritmo sopra riportato rispondendo ad alcune domande:

1. Quale variabile assegnare nel passo successivo (Select-Unassigned-Variable), e in quale ordine provare i possibili valori (Order-Domain-Values)?
2. Quali inferenze vanno eseguite a ciascun passaggio della ricerca (Inferenza)?
3. Quando una ricerca raggiunge un assegnamento che viola un vincolo, può evitare di ripetere tale fallimento?

Risposte:

1. Nell'algoritmo si ha la riga $var \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(csp)$
La strategia più banale per questa operazione è quella di assegnare la prima variabile nell'ordine $\{X_1, X_2, \dots\}$, nonostante questo approccio non dia risultati efficienti (**ordinamento statico**). Ad esempio, osservando la cartina dell'Australia, dopo gli assegnamenti $WA = \text{rosso}$, $NT = \text{verde}$, mi rendo conto immediatamente che SA non può che diventare *blu*, ma rispettando l'ordinamento statico la variabile scelta è Q .

```

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

L'idea di scegliere la variabile con meno valori legali nel proprio dominio è detta **euristica MRV** (Minimum Remaining Values), detta anche **fail-first**. Questa sceglie in pratica la variabile per cui è maggiore la probabilità di giungere ad un fallimento, potando così l'albero. Questo porta ad un miglioramento di fattore 1000.

Nella fase iniziale in cui si deve scegliere la regione da colorare per prima, l'euristica MRV non è d'aiuto poiché tutte le variabili hanno nel dominio tutti e tre i colori. In questo caso viene in aiuto la **degree heuristic**; questa euristica cerca di selezionare la variabile coinvolta nel maggior numero di vincoli con le altre variabili non assegnate, riducendo così il *branching factor* nelle scelte future. Per l'ordine con cui esaminare i possibili valori da assegnare ad una variabile è utile l'euristica del **valore meno vincolante**, che predilige il valore che lascia più libertà alle variabili adiacenti. Per ottenere questo risultato è necessario calcolare

$$rem(Y|P) := \#\text{(valori in } \text{dom}(Y) \text{ dato l'assegnamento } P \text{ dopo la propagazione di vincoli)}.$$

Si hanno due regole per scegliere il valore a da scegliere:

- *Product Rule*: $a^* = \arg \max_{a \in \text{Dom}(X)} \prod_{Y \in U} rem(Y|P \cup \{X = a\})$

- *Sum Rule*: come product rule ma al posto del prodotto si usa la somma.

Un'alternativa (Dechter & Pearl) è quella di approssimare il numero di soluzioni per il sottoproblema associato a ciascuna scelta di valore dal dominio. In questo modo posso considerare sottoproblemi con una variabile in meno, trasformando così un grafo in un albero analizzabile in tempo polinomiale piuttosto che esponenziale. Per contare il numero di soluzione per un sottoproblema posso usare l'algoritmo sopra e invece di ritornare la soluzione la aggiungo ad una lista.

2. Una delle forme di inferenza più semplici è la **forward checking** (verifica in avanti): ogni volta che una variabile X è assegnata, il processo di forward checking stabilisce la consistenza d'arco per essa: per ogni variabile non assegnata Y collegata ad X da un vincolo, cancella dal dominio di Y ogni valore non consistente con quello scelto per X . Possiamo considerare la forward checking come un modo efficiente per calcolare in modo incrementale le informazioni di cui l'euristica MRV necessita per il suo compito. La forward checking verifica molte inconsistenze, ma non tutte. Ad esempio nella terza riga della tabella, sia *NT* che *SA* sono obbligati ad essere *Blu*: inconsistenza poiché sono adiacenti. L'algoritmo **MAC** (*Maintaining Arc Consistency*) rileva questa inconsistenza: dopo che ad una variabile X_i è assegnato un valore, la procedura Inferenza richiama AC-3, ma invece di una coda di tutti gli archi del CSP considera solo gli archi $\{X_i, X_j\}$, con tutti gli X_j variabili non assegnate adiacenti a X_i . Da qui la AC-3 esegue la propagazione dei vincoli come di consueto.

Ricerca Locale per CSP

Gli algoritmi di ricerca locale si rivelano molto efficaci per molte tipologie di CSP.

La formulazione usata è quella a stato completo: lo stato iniziale assegna un valore ad ogni variabile e la ricerca provvede a cambiare il valore di una variabile per volta.

Poiché ovviamente la configurazione iniziale viola diversi vincoli, la ricerca locale punta ad eliminarli.

```

function MIN-CONFLICTS(csp, max-steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
           max-steps, the number of steps allowed before giving up

  current  $\leftarrow$  an initial complete assignment for csp
  for i = 1 to max-steps do
    if current is a solution for csp then return current
    var  $\leftarrow$  a randomly chosen conflicted variable from csp.VARIABLES
    value  $\leftarrow$  the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
  return failure

```

La scelta di un valore per una variabile è tipicamente scelto da un'euristica **min-conflicts**: si sceglie il valore che risulta nel minor numero di conflitti con le altre variabili. Questo algoritmo è in grado di risolvere il problema delle 8 regine in soli due passi. Inoltre si nota che il tempo di esecuzione dell'algoritmo è quasi indipendente dalla dimensione del problema. La particolare efficacia in questo problema è dovuto al fatto che le soluzioni sono distribuite densamente nello spazio degli stati.

Il panorama degli stati per un CSP sotto l'euristica min-conflicts presenta solitamente una serie di plateau. La ricerca all'interno del plateau, che consente di spostarsi lateralmente al suo interno, può aiutare la ricerca locale ad uscirne. Questo aggirarsi nel plateau può essere indirizzato con l'ausilio della **tabu search**: mantenere un piccolo elenco di stati visitati di recente che possa impedire all'algoritmo di tornarci.

Un'altra tecnica, denominata **constraint weighting** può aiutare a concentrare la ricerca sui vincoli importanti: ad ogni vincolo è assegnato un peso W_i , inizialmente 1 per tutti.

A ogni passo della ricerca, l'algoritmo sceglie di modificare una coppia variabile/valore che porterà al minimo peso totale di tutti i vincoli violati. I pesi sono poi aggiustati incrementando il peso di ciascun vincolo violato dell'assegnamento corrente.

La Struttura dei Problemi

Si esaminano adesso i modi in cui si può sfruttare la struttura del problema, rappresentata dal grafo dei vincoli, per trovare soluzioni rapidamente.

L'unico modo che si ha per rappresentare la complessità del mondo reale è sicuramente quello di scomporlo in molti sottoproblemi.

Prendendo di nuovo in esame l'esempio della mappa dell'Australia si nota che la Tasmania, staccata dagli altri stati, può essere di qualunque colore. In questi casi si dice che la Tasmania e il resto delle regioni compongono due **sottoproblemi indipendenti**. L'*indipendenza* si può appurare semplicemente cercando componenti connesse nel grafo dei vincoli.

Ogni componente corrisponde ad un sottoproblema CSP_i . Se l'assegnamento S_i è una soluzione di CSP_i , allora $\bigcup_i S_i$ è una soluzione di $\bigcup_i CSP_i$.

Supponiamo che ogni CSP_i abbia c variabili prese da un totale di n variabili, con c costante. Allora ci sono n/c sottoproblemi, ognuno dei quali richiede al più d^c lavoro, con d dimensione del dominio.

Il lavoro totale sarà $O(d^c \cdot n/c)$, cresce quindi linearmente con n .

Senza la scomposizione il costo sarebbe $O(d^n)$, che cresce esponenzialmente.

Purtroppo non sempre i sottoproblemi sono completamente indipendenti.

Vi sono comunque strutture dati diverse dal grafo che facilitano la risoluzione del problema in tempo lineare: un grafo di vincoli è un **albero** quando due variabili qualsiasi sono collegate da un solo cammino.

Si introduce un nuovo tipo di consistenza: la **consistenza d'arco orientato** (o **DAC**).

Un CSP si dice arco-orientato consistente con un ordinamento di variabili X_1, X_2, \dots, X_n se e solo se ogni variabile X_i è arco-consistente con ogni X_j con $j > i$. DEF

Per risolvere un CSP ad albero si sceglie per prima cosa una variabile come radice e si sceglie un ordinamento delle variabili tale che ognuna appaia dopo suo padre (**ordinamento topologico**).

Ogni albero con n nodi ha $n - 1$ archi, perciò possiamo rendere l'albero arco-orientato consistente in $O(n)$, ognuno dei quali deve confrontare fino a d possibili valori di dominio per due variabili, con tempo totale $O(nd^2)$.

```

function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components X, D, C

  n  $\leftarrow$  number of variables in X
  assignment  $\leftarrow$  an empty assignment
  root  $\leftarrow$  any variable in X
  X  $\leftarrow$  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]  $\leftarrow$  any consistent value from Di
    if there is no consistent value then return failure
  return assignment

```

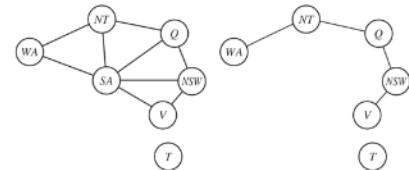
Una volta ottenuto un grafo arco-orientato consistente , possiamo semplicemente percorrere la lista di variabili e scegliere qualsiasi valore rimanente (questo perché essendo le variabili a due a due arco-consistenti, assegnando un valore al padre, so che vi sarà un valore legale per il figlio).

Visti gli enormi vantaggi nella risoluzione di CSP ad albero, vediamo adesso come poter **ridurre grafi di vincoli generici ad alberi**.

Vi sono due modi principali: **rimozione dei nodi** o **fusione dei nodi**.

La **rimozione dei nodi** prevede che si assegnino dei valori ad alcune variabili in modo che quelle rimanenti formino un albero. Ad esempio nella carta dell'Oceania assegno un valore a *SA*. Fatto questo, ogni soluzione per il CSP dopo rimozione di *SA* e dei suoi vincoli sarà consistente col valore prescelto per *SA*.

Naturalmente il valore scelto per la variabile (o sottoinsieme di variabili) iniziale potrebbe essere non corretto. Dovrei quindi provare ogni valore, attraverso il seguente algoritmo:



1. Scegliete un sottoinsieme *S* delle variabili del CSP tale che il grafo dei vincoli diventi un albero dopo la sua rimozione. *S* prende il nome di **cycle cutset** (insieme di taglio dei cicli).
2. Per ogni possibile assegnamento delle variabili in *S* che soddisfa tutti i vincoli su *S*:
 - A. Rimuovere dal dominio delle variabili rimanenti tutti i valori non consentiti con gli assegnamenti in *S*.
 - B. Se il CSP risultante ha una soluzione, restituitela insieme all'assegnamento per *S*.

Se il *cycle cutset* ha dimensioni *c*, il tempo di esecuzione totale è $O(d^c \cdot (n - c)d^2)$: dobbiamo provare ciascuna delle d^c combinazioni di valori delle variabili in *S*, e per ogni combinazione dobbiamo risolvere un problema su un albero di dimensione $n - c$.

Notare come al diminuire di *c* la ricerca diventi Backtracking-free e il risparmio di tempo diventi enorme. Trovare il più piccolo *c* è NP-Hard.

La **fusione dei nodi** è basata sulla **tree decomposition** del grafo di vincoli in un insieme di sottoproblemi collegati, detto **cluster tree**. Ogni sottoproblema viene risolto indipendentemente e le soluzioni sono poi combinate (paradigma *divide et impera*).

Una scomposizione deve soddisfare tre requisiti:

- Ogni variabile del problema originale deve comparire in almeno uno dei sottoproblemi.
- Se due variabili sono collegate da un vincolo nel problema originale, devono comparire insieme in almeno uno dei sottoproblemi.
- Se una variabile compare in due sottoproblemi sull'albero, deve essere presente anche in tutti i sottoproblemi che compongono il cammino che li collega.

Ogni sottoproblema viene risolto in modo indipendente; se uno di essi non ha soluzione, non ce l'ha neanche il problema originale.

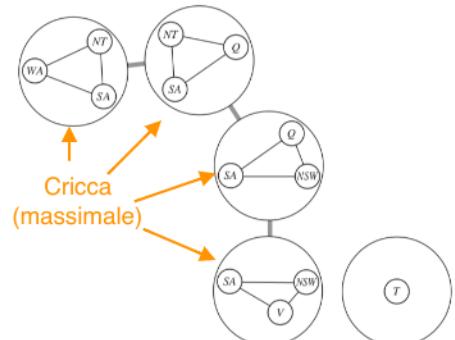
A questo punto si può costruire una soluzione globale come segue:

- Consideriamo ogni sottoproblema come una “mega-variabile” il cui dominio è l’insieme di tutte le sue soluzioni.
- Risolviamo i vincoli che collegano tra loro i sottoproblemi attraverso il potente algoritmo Tree-CSP-Solver () .

Un grafo di vincoli ammette diverse decomposizioni ad albero; la scelta deve tenere conto che i sottoproblemi devono essere più piccoli possibile.

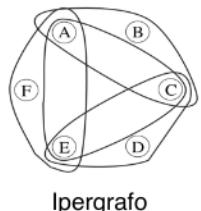
Per la modalità di conversione di un grafo in un cluster tree, teniamo conto di alcune definizioni: definiamo **cricca** un sottoinsieme di nodi che ha la proprietà di formare grafo completo (un grafo si dice completo quando ogni vertice è collegato a tutti i vertici rimanenti).

Definiamo **cricca massimale** una cricca che non è contenuta massimamente in un’altra cricca.



Fin ora abbiamo visto grafi con vincoli binari, rappresentabili attraverso un grafo semplice.

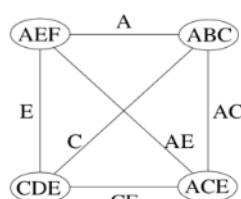
Una rappresentazione utile per CSP con vincoli di grado maggiore di due è l'**ipergrafo**: un grafo in cui i nodi rappresentano sempre le variabili e gli *iperarchi* raggruppano i nodi (ovvero le variabili) che partecipano allo stesso scopo.



Vi sono poi due varianti di questa rappresentazione utili per facilitare la loro gestione negli algoritmi: **grafo di vincoli primale** e **grafo di vincoli duale**.

Il **grafo primale** è una semplice estensione del grafo con vincoli binari visto finora.

Il **grafo duale** rappresenta ciascun “ambito” attraverso un nodo e associa un arco etichettato tra tutte le possibili coppie di nodi i cui “ambiti” condividono almeno una variabile.



Ottenuto un grafo duale a partire da un problema con vincoli non-binari, possiamo ottenere l’albero corrispondente al duale in modo da trovare la soluzione del problema originario in tempo lineare. Molto spesso costruendo un grafo duale alcune variabili risultano essere *ridondanti* (ovvero la loro rimozione non comporta variazioni nell’insieme di tutte le possibili soluzioni). Un vincolo e i suoi corrispondenti archi possono essere eliminati se le variabili che etichettano un arco sono condivise da ogni arco lungo un percorso alternativo tra due nodi.

Nell’esempio a lato del grafo duale, posso eliminare CDE ed i suoi archi E,C,A.

Grafo Duale

A questo proposito, enunciamo una proprietà che assicura la legittimità di rimozione di un arco di un ipergrafo (ovvero un sottografo): **running intersection property**: $\forall u, v \in V$, se $x = (u \wedge v) \neq 0$, allora x è vincolato ovunque nel cammino che connette u e v ; in altre parole, un arco di un grafo duale soddisfa la proprietà se per ogni due nodi che condividono una variabile, c’è almeno un cammino di archi etichettati ciascuno contenente le variabili condivise.

Un arco di un ipergrafo (ovvero un sottografo) che rispetta la proprietà running intersection property è detto join-graph. Un join-graph che è un albero è detto **Junction Tree**.

Per trovare un Junction Tree si può usare il max-spanning tree (Kruskal).



Agenti Logici

Nel quale progettiamo agenti che possono costruire rappresentazioni del mondo, applicare processi di inferenza per derivare nuove rappresentazioni e usarle per dedurre cosa fare.

Gli agenti analizzati per la Ricerca (locale o classica) conoscono cose, ma solo in senso limitato e non flessibile. Ad esempio nel problema del rompicapo a 8 tasselli il modello di transizione è nascosto all'interno del codice, chiuso rispetto al suo dominio. Un agente di questo tipo di fronte ad ambienti parzialmente osservabili ha la sola possibilità di elencare tutti i possibili stati concreti: prospettiva poco promettente per ambienti molto grandi.

Si arriva ora a sviluppare la **logica** come classe generale di rappresentazioni per supportare agenti basati sulla conoscenza. Tali agenti possono ricombinare informazioni per una miriade di scopi.

Si presenta ora una struttura generale dell'agente, poi una descrizione di un nuovo ambiente (Wumpus), i principi generali della logica e della logica proposizionale ed infine alcune tecnologie ben sviluppate per l'inferenza.

Agenti Basati sulla Conoscenza

Il componente più importante degli agenti basati sulla conoscenza è la **base di conoscenza (KB, Knowledge Base)**. Informalmente diciamo che la KB è composta da **formule** e che ciascuna di esse rappresenta un'asserzione sul mondo. Talvolta questa prende il nome di **assioma** se viene data per buona e non è derivata da altre formule.

La KB deve prevedere dei meccanismi per aggiungere nuove formule (asserzioni sull'ambiente) e per le interrogazioni.

I nomi standard per queste due azioni sono TELL (asserisci) e ASK (chiedi): entrambe possono comportare un processo di inferenza, ovvero la derivazione di ulteriori formule a partire da quelle note.

L'inferenza deve soddisfare il requisito fondamentale che la risposta ad ogni richiesta (ASK) posta alla KB sia una conseguenza di quello che le è stato precedentemente detto (TELL).

Per fissare il concetto di interazione tra agente e KB si consideri un programma agente:

Come tutti gli agenti, prende in input una percezione e restituisce un'azione. L'agente mantiene in memoria una KB che interollerà facendo sempre tre cose:

1. Le comunica le percezioni ricevute (con TELL).
2. Le chiede (ASK) quali azioni eseguire.
3. Una volta che è stata scelta un'azione, attraverso un processo di ragionamento, l'agente la registra nella KB (con TELL).

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

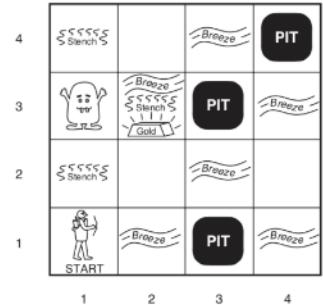
Il Mondo del Wumpus

Si tratta di una caverna composta da stanze buie collegate tra loro. In una di queste vi è un mostro (il Wumpus), che può essere ucciso dall'agente attraverso l'unica freccia a sua disposizione. In una stanza vi è un mucchio d'oro, in altre tre un pozzo senza fondo da cui l'agente, una volta entrato, non potrebbe più uscire.

Definizione PEAS dell'ambiente:

- **Misura di Prestazione:** +1000 se l'agente riesce ad uscire con l'oro, -1000 se si cade nel pozzo o viene divorato. -1 per ogni azione, -10 per l'uso della freccia.
- **Ambiente:** griglia 4x4 di stanze. L'agente comincia nella [1,1].
- **Attuatori:** L'agente può svolgere le azioni: *avanti*, *ruota a sx*, *ruota a dx*. Muore se entra in un pozzo o entra nella stanza del Wumpus. Vi è l'azione *scocca* per la freccia, l'azione *afferra* per l'oro e l'azione *esci* per uscire nella [1,1].
- **Sensori:** Ha cinque sensori, ognuno con un bit di informazione:
 - Nella stanza del Wumpus e in quelle adiacenti percepisce *Fetore*.
 - Nelle stanze adiacenti ai pozzi percepisce una *Brezza*.
 - Nel riquadro con oro percepisce uno *Scintillio*.
 - Quando sbatte contro un muro percepisce un *Urto*.
 - Quando il Wumpus viene ucciso emette un *Ululato* percepibile ovunque.

Sul libro è presente un esempio che spiega passo passo un procedimento di inferenza per questo ambiente.



Logica

Si introducono i concetti fondamentali della rappresentazione e del ragionamento logico. Abbiamo detto che la KB è composta da formule. Queste sono espresse secondo una **sintassi** del linguaggio di rappresentazione, che specifica quali di esse sono "ben formate". Una logica deve anche definire una **semantica**, grazie alla quale sarà possibile definire la **verità** di una o più formule rispetto ad ogni mondo possibile (che d'ora in poi chiameremo modello). Un **modello** è un'astrazione matematica del mondo reale che fissa il valore di verità di ogni formula.

Se una formula α è vera in un modello m , diciamo che m **soddisfa** α .

Indicheremo con $M(\alpha)$ l'insieme di tutti i modelli di α , ovvero l'insieme di tutti i modelli in cui α è vera.

Trattiamo adesso il *ragionamento logico*.

Se da α **consegue logicamente** β scriviamo $\alpha \models \beta$. Formalmente,

$\alpha \models \beta$ SSE, in ogni modello in cui α è vera, anche β lo è: $\alpha \models \beta \iff M(\alpha) \subseteq M(\beta)$



Tornando all'esempio del Wumpus, supponiamo che l'agente non abbia percepito nulla in [1,1] e abbia avvertito una brezza in [2,1]. Queste percezioni, unite alla conoscenze delle regole del mondo del Wumpus da parte dell'agente, costituiscono la KB. Ovviamente la KB è falsa nei modelli che contraddicono le conoscenze dell'agente.

Nel nostro esempio i modelli in cui la KB è vera sono solo tre, delimitati con la linea continua in figura.

Per comprendere meglio l'uso delle formule consideriamo le seguenti conclusioni:

$$\alpha_1 = \text{"Non c'è pozzo in [1,2]"}$$

$$\alpha_2 = \text{"Non c'è pozzo in [2,2]"}$$

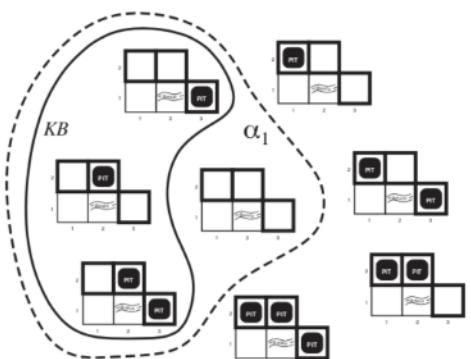
I modelli corrispondenti sono tratteggiati in figura.

Si verifica che:

In ogni modello in cui KB è vera, lo è anche α_1 , allora

$KB \models \alpha_1$.

Quindi l'agente può concludere che non vi è un pozzo in [1,2].

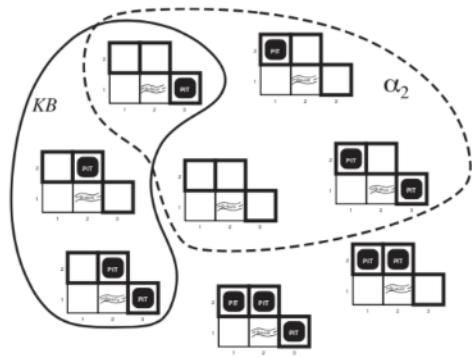


In alcuni modelli in cui KB è vera, α_2 è falso, allora $KB \not\models \alpha_2$.

Quindi l'agente non può concludere che non vi sia un pozzo in [2,2].

Questo esempio illustra come la conseguenza logica sia fondamentale per derivare conclusioni, ovvero per eseguire **inferenze logiche**.

L'algoritmo di inferenza riportato nelle due figure precedenti è detto **model checking**, perché enumera tutti i possibili modelli per verificare che α sia vera in tutti i modelli in cui è vera la KB , ovvero verifica che $M(KB) \subseteq M(\alpha)$.



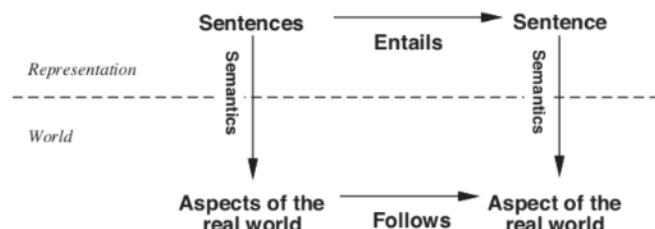
Formalmente, se un algoritmo di inferenza i può derivare α da KB , scriviamo $KB \vdash_i \alpha$.

Un algoritmo di inferenza che deriva solo formule che sono conseguenze logiche è detto **corretto**.

Un algoritmo di inferenza si dice **completo** se può derivare ogni formula che è conseguenza logica.

Si è descritto un *processo di ragionamento* le cui conclusioni sono garantite vere in qualsiasi mondo in cui sono vere le premesse. In particolare, se KB è vera nel mondo reale, allora ogni formula α derivata da KB con un procedimento di inferenza corretto è anch'essa vera nel mondo reale.

Osservando la figura sotto, mentre un processo di inferenza lavora "a livello" di sintassi, ovvero a livello di configurazioni fisiche interne alle formule, vi è una corrispondenza con la relazione esistente nel mondo reale, in cui qualche aspetto accade in virtù del fatto che ne accada un altro.



L'ultimo aspetto che rimane da considerare è quello del **grounding** (ancoramento): il legame, se esiste, tra ragionamento logico e ambiente reale in cui si trova l'agente (facendo riferimento alla figura sopra: tra *Representation* e *World*).

Come facciamo quindi a sapere che la KB è vera nel mondo reale?

La risposta è che il legame è creato dai sensori dell'agente.

Il significato e la verità delle formule percettive sono così definite dai processi di percezione e costruzione sintattica che le producono.

Logica Proposizionale

La **sintassi** della logica proposizionale definisce le formule accettabili. Le **formule atomiche** consistono di un singolo simbolo proposizionale. Ogni simbolo rappresenta una proposizione e può essere vera o falsa. Ad esempio $W_{1,3}$ è la proposizione che indica la posizione del Wumpus in [1,3].

Si possono costruire **formule complesse** partendo da formule più semplici attraverso l'uso dei connettivi logici: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$.

La **semantica** specifica le regole usate per determinare il valore di verità di una formula nei confronti di un particolare modello. Nella logica proposizionale un modello fissa semplicemente il valore di verità di ogni simbolo proposizionale.

Ad esempio, se le formule in una KB fanno uso dei simboli $P_{1,2}, P_{2,2}, P_{3,1}$, un modello possibile è $m_1 = \{P_{1,2} = \text{false}, P_{2,2} = \text{false}, P_{3,1} = \text{true}\}$.

Per le formule atomiche è semplice definire il valore di verità.

Per le formule complesse si usano le regole semplici di logica applicate attraverso i connettivi.

Un esempio di KB.

Costruiamo una KB per il mondo del Wumpus utilizzando la logica proposizionale.

Si considerano solo gli aspetti immutabili del mondo.

Per ogni posizione $[x,y]$ si hanno i simboli:

- $P_{x,y}$ è vera se esiste un pozzo in $[x,y]$.
- $W_{x,y}$ è vera se esiste un Wumpus in $[x,y]$, morto o vivo.
- $B_{x,y}$ è vera se l'agente percepisce una brezza in $[x,y]$.
- $S_{x,y}$ è vera se l'agente percepisce un fetore in $[x,y]$.

Scriviamo le formule sufficienti per ricavare la proposizione $\neg P_{1,2}$ (non ci sono pozzi in $[1,2]$). Ogni formula è etichettata con R_i :

- In $[1,1]$ non ci sono pozzi:

$$R_1 := \neg P_{1,1}$$

- In una stanza si percepisce una brezza se e solo se c'è un pozzo in una stanza adiacente. Si indicano solo le due formule rilevanti:

$$R_2 := B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

$$R_3 := B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$$

- Ora aggiungiamo le percezioni relative alla  raccolte nelle prime due stanze:

$$R_4 := \neg B_{1,1}$$

$$R_5 := B_{2,1}$$

Lo scopo ora è decidere se, data una formula α , vale la conseguenza logica $KB \models \alpha$, ovvero, ricordando la definizione, se $M(KB) \subseteq M(\alpha)$.

Si presenta un algoritmo detto **model checking** che è un'implementazione diretta della definizione di conseguenza logica: dovremo enumerare esplicitamente i modelli e verificare che α sia vera in ogni modello in cui la KB lo è. Questo algoritmo è completo e corretto.

Costo temporale $O(2^n)$, con n numero di simboli in KB e α e 2^n il numero di modelli.

```
function TT-ENTAILS?(KB,  $\alpha$ ) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
           $\alpha$ , the query, a sentence in propositional logic
symbols  $\leftarrow$  a list of the proposition symbols in KB and  $\alpha$ 
return TT-CHECK-ALL(KB,  $\alpha$ , symbols, { })
```

```
function TT-CHECK-ALL(KB,  $\alpha$ , symbols, model) returns true or false
if EMPTY?(symbols) then
  if PL-TRUE?(KB, model) then return PL-TRUE?( $\alpha$ , model)
  else return true // when KB is false, always return true
else do
   $P \leftarrow$  FIRST(symbols)
  rest  $\leftarrow$  REST(symbols)
  return (TT-CHECK-ALL(KB,  $\alpha$ , rest, model  $\cup$  { $P = \text{true}$ })
         and
         TT-CHECK-ALL(KB,  $\alpha$ , rest, model  $\cup$  { $P = \text{false}$ }))
```

Dimostrazione di Teoremi nella Logica Proposizionale

Fin qui abbiamo mostrato come determinare la conseguenza logica attraverso il *model checking*, enumerando i modelli e mostrando che la formula deve valere in tutti.

Vediamo adesso come la conseguenza logica può essere ottenuta attraverso la **dimostrazione di teoremi (Theorem Proving)**: applico le regole di inferenza direttamente alle formule della KB per costruire una dimostrazione della formula desiderata senza consultare alcun modello.

Introduciamo qualche concetto necessario alla comprensione del paragrafo.

Il primo è quello di **equivalenza logica**: $\alpha \equiv \beta \Leftrightarrow \alpha \models \beta \wedge \beta \models \alpha$, quindi α e β sono logicamente equivalenti se sono vere nello stesso insieme di modelli.

Il secondo concetto è quello di **validità**: una formula è valida se è vera in tutti i modelli (in tal caso viene detta **tautologia**).

L'ultimo concetto è quello della **soddisfacibilità**: una formula è soddisfacibile se è vera in, o soddisfatta da, qualche modello.

Il problema di determinare la soddisfacibilità delle formule nella logica proposizionale (problema **SAT**) è stato il primo problema per cui è stata dimostrata la NP-completezza.

Proposizione

- α è valida $\Leftrightarrow \neg\alpha$ è insoddisfacibile
- $\alpha \models \beta \Leftrightarrow (\alpha \wedge \neg\beta)$ è insoddisfacibile
- $\alpha \models \beta \Leftrightarrow (\alpha \Rightarrow \beta)$ è valida (Teorema di Deduzione)

Trattiamo adesso le **regole di inferenza** che possono essere usate per in una dimostrazione:

- La regola più nota è quella del **Modus Ponens**, che si scrive: $\frac{(\alpha \Rightarrow \beta), \quad \alpha}{\beta}$.

Significa che, ogni volta che sono date le formule $\alpha \Rightarrow \beta$ e α , allora si può inferire β .

- Un'altra utile formula è l'**eliminazione degli and**: $\frac{\alpha \wedge \beta}{\alpha}$, che si può applicare su entrambi i congiunti.

- **Eliminazione del bicondizionale**: $\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}$

Ritorniamo all'esempio del Wumpus per vedere come si applicano queste regole di inferenza.

La KB iniziale contiene le formule da R_1 a R_5 (vedi sopra), vediamo come ricavare, a partire da queste, la formula $\neg P_{1,2}$ (non c'è alcun pozzo in [1,2]).

Applico l'eliminazione del bicondizionale a R_2 .

$$R_6 := (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$$

Applico l'eliminazione degli and.

$$R_7 := ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$$

Per l'equivalenza logica con la contrapposizione.

$$R_8 := \neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1})$$

Applico il Modus Ponens su R_8 e sfrutto R_4 .

$$R_9 := \neg(P_{1,2} \vee P_{2,1})$$

Applico De Morgan

$$R_{10} := \neg P_{1,2} \wedge \neg P_{2,1} \text{ (ovvero non vi sono pozzi in [1,2])}$$

La strada del Theorem Proving rappresenta quindi una valida alternativa al Model Checking, risultando spesso anche più vantaggiosa: attraverso la dimostrazione è possibile ignorare le formule irrilevanti.

Dimostrazione per Risoluzione

Le regole di inferenza viste fin qui sono tutte *corrette*, tuttavia non garantiscono la *completezza* per tutti gli algoritmi di ricerca che le utilizzano.

Introduciamo adesso una singola regola di inferenza, la **risoluzione**, che unita a qualsiasi algoritmo di ricerca completo dà luogo ad un algoritmo di inferenza completo.

Riprendiamo l'esempio del Wumpus e supponiamo l'applicazione di ulteriori regole inferenziali (che non sono specificate) che hanno portato a dedurre la seguente formula:

$$R_{15} := P_{1,1} \vee P_{2,2} \vee P_{3,1}$$

Si può applicare ora la prima regola della risoluzione: sappiamo che vale $\neg P_{2,2}$ e $\neg P_{1,1}$, diciamo che queste *risolvono* rispettivamente $P_{2,2}$ e $P_{1,1}$ in R_{15} . Quindi, da R_{15} otteniamo:

$$R_{17} := P_{3,1}$$

Questi ultimi passi inferenziali sono esempi della regola di inferenza di **risoluzione unitaria**:

$$\frac{\ell_1 \vee \dots \vee \ell_k, \quad m}{\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k}$$

Dove ℓ è un letterale, ℓ_i e m sono **letterali complementari** (uno è la negazione dell'altro, ovvero $\neg \ell_i = m$).

La regola di risoluzione unitaria prende quindi una **clausola** -cioè una disgiunzione di letterali- e un letterale (m) e produce una nuova clausola.

In generale, la risoluzione prende due clausole e ne produce una nuova che contiene tutti i letterali delle due clausole originali *tranne* i due complementari.

$$\text{Ad esempio: } \frac{P_{1,1} \vee P_{3,1}, \quad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}$$

L'operazione di eliminazione di eventuali copie di letterali è detta **fattorizzazione**.

Forma Normale Congiuntiva (CNF)

È importante notare che la risoluzione unitaria è utilizzabile solo a clausole, ovvero disgiunzioni di letterali.

In realtà *ogni formula della logica proposizionale è logicamente equivalente a una congiunzione di clausole*. Una formula così espressa viene detta in **CNF** (*Conjunctive Normal Form*).

La procedura per ottenere la forma CNF è la seguente:

- Elimino \Leftrightarrow , sostituendo quindi $\alpha \Leftrightarrow \beta$ con $\alpha \Rightarrow \beta \wedge \beta \Rightarrow \alpha$
- Elimino \Rightarrow , sostituendo $\alpha \Rightarrow \beta$ con $\neg \alpha \vee \beta$
- Il \neg deve essere applicato solo ai letterali, quindi lo si "muove all'interno"
- Si hanno ora \wedge e \vee nidificati, si usa la distributiva per ottenere una congiunzione di clausole.

Le procedure di inferenza basate sulla risoluzione sfruttano il principio di dimostrazione per assurdo. Quindi se vogliamo dimostrare che $KB \models \alpha$, dimostriamo che $(KB \wedge \neg\alpha)$ è insoddisfacibile (vedi Proposizione sopra).

Si ha un algoritmo che fa questo:

```

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

  clauses  $\leftarrow$  the set of clauses in the CNF representation of  $KB \wedge \neg\alpha$ 
  new  $\leftarrow \{ \}$ 
  loop do
    for each pair of clauses  $C_i, C_j$  in clauses do
      resolvents  $\leftarrow$  PL-RESOLVE( $C_i, C_j$ )
      if resolvents contains the empty clause then return true
      new  $\leftarrow$  new  $\cup$  resolvents
    if new  $\subseteq$  clauses then return false
    clauses  $\leftarrow$  clauses  $\cup$  new
  
```

Converte $(KB \wedge \neg\alpha)$ in CNF.
 Applica la regola di risoluzione alle clausole risultanti.
 Ogni coppia che contiene coppie letterali complementari è risolta per produrre una nuova clausola che viene aggiunta all'insieme.
 Il processo continua finché non si verifica una delle due possibilità:

- Non è più possibile aggiungere alcuna clausola, quindi α non è conseguenza logica di KB .
- La risoluzione applicata a due clausole dà come risultato la clausola vuota, allora α è conseguenza logica della KB .

Per concludere la discussione sulla risoluzione, dimostriamo che PL-RESOLUTION() è completo. Per farlo si introduce la nozione di **chiusura della risoluzione** $RC(S)$ di un insieme di clausole S , che è l'insieme di tutte le clausole derivabili dall'applicazione ripetuta della regola di risoluzione alle clausole in S .

Si enuncia e dimostra il **Teorema di Risoluzione Ground**:

“Se un insieme di clausole è insoddisfacibile, la sua chiusura della risoluzione contiene la clausola vuota”

Dim: per assurdo, se la chiusura $RC(S)$ non contiene la clausola vuota, S è soddisfacibile.

La completezza della risoluzione la rende una procedura di inferenza molto importante. Tuttavia in molti casi non è necessaria tutta questa potenza. Alcune KB soddisfano certe restrizioni sulle clausole tali da rendere gli algoritmi dedicati più efficienti. Alcuni casi di classificazione delle clausole:

- La **clausola definita** è una disgiunzione di letterali di cui esattamente uno è positivo.
- La **clausola di Horn** è una disgiunzione di letterali in cui al massimo uno dei letterali è positivo. La clausola definita è un sottocaso della clausola di Horn, cos' come la **clausola obiettivo**, nella quale nessun letterale è positivo.

Le clausole di Horn sono chiuse rispetto alla risoluzione: se si risolvono due clausole di Horn se ne ottiene ancora una di Horn.

Le KB contenenti esclusivamente clausole di Horn sono interessanti per tre ragioni:

1. Ogni clausola definita può essere scritta come un'implicazione la cui premessa è una congiunzione di letterali positivi e la cui conclusione è un letterale positivo.
2. L'inferenza sulle clausole di Horn può essere svolta mediante algoritmi di **concatenazione in avanti** e **concatenazione all'indietro**, che vedremo tra poco.
3. Con le clausole di Horn possiamo determinare una conseguenza logica in un tempo che cresce *linearmente* con la dimensione della KB.

L'algoritmo di **concatenazione in avanti** PL-FC-Entails? (KB, q) determina se un singolo simbolo proposizionale q è conseguenza logica della KB composta da sole clausole definite. L'algoritmo comincia dai fatti conosciuti nella KB. Poi prende le implicazioni e se tutti i fatti a sinistra di ciascuna di esse è verificato, allora aggiunge la conclusione alle conoscenze della KB. Questo processo continua finché non viene aggiunta q alla KB o non è più possibile effettuare alcuna inferenza.
Un dettaglio fondamentale di questo algoritmo è la sua complessità lineare.

```

function PL-FC-ENTAILS?( $KB, q$ ) returns true or false
  inputs:  $KB$ , the knowledge base, a set of propositional definite clauses
            $q$ , the query, a proposition symbol
   $count \leftarrow$  a table, where  $count[c]$  is the number of symbols in  $c$ 's premise
   $inferred \leftarrow$  a table, where  $inferred[s]$  is initially false for all symbols
   $agenda \leftarrow$  a queue of symbols, initially symbols known to be true in  $KB$ 

  while  $agenda$  is not empty do
     $p \leftarrow \text{POP}(agenda)$ 
    if  $p = q$  then return true
    if  $inferred[p] = \text{false}$  then
       $inferred[p] \leftarrow \text{true}$ 
      for each clause  $c$  in  $KB$  where  $p$  is in  $c.\text{PREMISE}$  do
        decrement  $count[c]$ 
        if  $count[c] = 0$  then add  $c.\text{CONCLUSION}$  to  $agenda$ 
  return false

```

L'algoritmo è **corretto**: ogni inferenza è sostanzialmente l'applicazione del modus ponens. È anche **completo**.

Model Checking Proposizionale ed Efficiente

Si descrive un algoritmo efficiente per l'inferenza proposizionale che si rifà al model checking e si basa sulla ricerca backtracking.

Questo algoritmo serve a verificare la soddisfacibilità: il problema SAT.

Il primo algoritmo che si considera viene spesso chiamato **algoritmo Davis-Putnam** (o DPLL). DPLL prende in input una formula in CNF. Così come TT-ENTAILS? e RICERCA-BACKTRACKING si tratta essenzialmente di un'enumerazione ricorsiva in profondità dei modelli possibili. Rispetto a TT-ENTAILS?, apporta tre migliorie:

- **Terminazione anticipata**: l'algoritmo è in grado di determinare se la formula è vera o falsa anche con un modello parzialmente incompleto. Una clausola è vera se qualsiasi suo letterale è vero, anche se agli altri non è stato assegnato un valore di verità. Ad esempio $(A \vee B) \wedge (A \vee C)$ è vera se A è vera, indipendentemente da B e C .
- **Euristica del simbolo puro**: un **simbolo puro** è un simbolo che compare sempre con lo stesso segno in tutte le clausole. È facile vedere che se una formula ha un modello, i valori di verità dei simboli puri saranno assegnati in modo che i corrispondenti letterali valgano *true*, in modo tale che la clausola diventi sempre *true*.
- **Euristica della clausola unitaria**: una clausola unitaria è una clausola che contiene solo un letterale. Nel contesto del DPLL, il termine indica anche clausole in cui è già stato assegnato dal modello il valore *false* a tutti i letterali tranne uno.

function DPLL-SATISFIABLE?(*s*) **returns** true or false

inputs: *s*, a sentence in propositional logic

clauses \leftarrow the set of clauses in the CNF representation of *s*

symbols \leftarrow a list of the proposition symbols in *s*

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* \leftarrow FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)

if *P* is non-null **then return** DPLL(*clauses*, *symbols* - *P*, *model* \cup {*P*=*value*})

P, *value* \leftarrow FIND-UNIT-CLAUSE(*clauses*, *model*)

if *P* is non-null **then return** DPLL(*clauses*, *symbols* - *P*, *model* \cup {*P*=*value*})

P \leftarrow FIRST(*symbols*); *rest* \leftarrow REST(*symbols*)

return DPLL(*clauses*, *rest*, *model* \cup {*P*=true}) **or**

DPLL(*clauses*, *rest*, *model* \cup {*P*=false}))

Possiamo trovare un'alternativa a DPLL, utilizzando un algoritmo di ricerca locale.

Come funzione di valutazione utilizziamo quella che minimizza il numero di clausole non soddisfatte, un po' come MIN-CONFLICTS().

Il più efficiente algoritmo studiato è WALK-SAT(): ad ogni iterazione, l'algoritmo seleziona una clausola non soddisfatta e cambia il valore di verità di uno dei suoi simboli. Per scegliere il simbolo ha due alternative: un passo che minimizzi il numero di clausole non soddisfatte o una camminata casuale che lo scelga casualmente.

Se l'algoritmo restituisce un modello, la formula è soddisfacibile, altrimenti o la formula è insoddisfacibile o dovremmo concedergli più tempo.

Tuttavia la ricerca locale non è in grado di determinare l'insoddisfacibilità.

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 flips allowed before giving up

model \leftarrow a random assignment of true/false to the symbols in *clauses*

for *i* = 1 **to** *max_flips* **do**

if *model* satisfies *clauses* **then return** *model*

clause \leftarrow a randomly selected clause from *clauses* that is false in *model*

with probability *p* 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

Il Panorama dei problemi SAT casuali

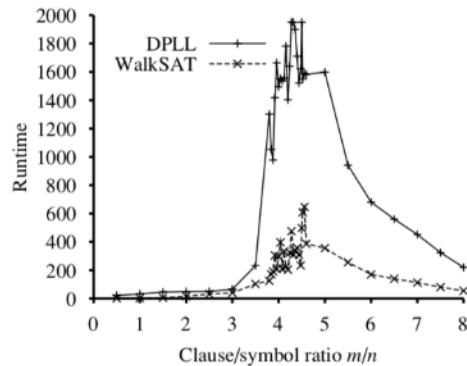
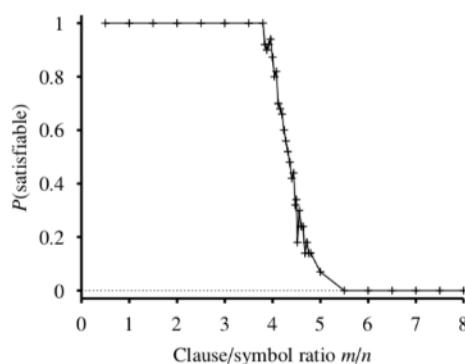
I problemi SAT, essendo NP-completi, avranno almeno alcune istanze che richiederanno un tempo esponenziale.

Quando si considerano i problemi di soddisfabilità in CNF, diciamo che un problema è **sotto-vincolato** quando ci sono relativamente *poche* clausole che vincolano le variabili.

Si definisce con esattezza il modo in cui sono generate clausole casuali:

La notazione $CNF_k(m, n)$ denota una formula $k - CNF$ con m clausole e n simboli, in cui le clausole sono scelte in modo uniforme, indipendente e senza sostituzione tra tutte le clausole con k letterali diversi.

Data una sorgente di clausole casuali, possiamo misurare la probabilità della soddisfabilità. Il grafico a sx traccia la probabilità (di soddisfabilità) di $CNF_3(m, 50)$ in funzione del rapporto clausole/simboli m/n . Per m/n piccoli, la probabilità è 1, ma al crescere di m/n tende a 0. Si può notare come la probabilità tenda bruscamente verso il basso intorno a $m/n = 4,3$. Da cui la congettura sotto.



Teorema (Congettura della soglia di soddisfabilità)

Se $k \geq 3$ ogni algoritmo DPLL impiega un tempo esponenziale su $CNF_k(n, 2n)$ per

$$\frac{m}{n} \geq 2^k \log 2 \text{ (con alta probabilità).}$$

Oppure: per ogni $k \geq 3$, esiste un rapporto di soglia r_k tale che, al tendere di n a infinito, la probabilità che $CNF_k(n, rn)$ sia soddisfacibile diventa 1 per tutti i valori di r al di sotto di tale soglia, e 0 per tutti i valori al di sopra.



Incertezza

Gli agenti logici visti fin ora si assumono l'impegno epistemologico che le preposizioni siano vere, false o sconosciute. Quindi se un agente conosce abbastanza fatti sull'ambiente, l'approccio logico gli permette di derivare piani di sicuro funzionamento. Tuttavia un agente che agisce nel mondo reale non ha quasi mai accesso a tutte le informazioni necessarie per poter eseguire un'inferenza esatta: un agente deve quindi essere in grado di agire in condizioni di **incertezza**.

Per gestire l'incertezza l'agente può, nel caso migliore, fornire un certo *grado di credenza* per una certa formula logica. Lo strumento principale che viene utilizzato per la gestione dei gradi di credenza consiste nella **teoria della probabilità**, che assegna ad ogni formula un valore compreso tra 0 e 1. Questo valore viene anche detto **belief**, e lo si può definire in due modi:

- **De Finetti** (Bet system) utilizza un *approccio pragmatico* e lo definisce $bel(x) = \frac{1}{1+m}$, con $m : 1$ bets. Secondo lui un sistema di scommesse è coerente sse i numeri soddisfano gli assiomi di probabilità.
- **Cox-James Axioms**, viene utilizzato un *approccio assiomatico*.
 - Se vale $bel(X|I) > bel(Y|I) \wedge bel(Y|I) > bel(Z|I)$, allora $bel(X|I) > bel(Z|I)$
 - \exists funzione F t.c. $bel(X|I) = F(bel(\neg X|I))$
 - \exists funzione G t.c. $bel(X \wedge Y|I) = G(bel(X|I), bel(Y|X \wedge I))$

Nella teoria delle probabilità una formula come “*la probabilità che un paziente abbia una carie è 0,8*” riguarda esclusivamente le credenze dell'agente e non direttamente il mondo, come invece avveniva per una data formula nel caso della logica. Le credenze sono frutto delle percezioni ricevute all'istante corrente. Quando la probabilità è calcolata prima della percezione viene detta **a priori**, quando viene calcolata dopo una percezione è detta **condizionata** (o a posteriori).

La presenza dell'incertezza modifica radicalmente le modalità in cui un agente prende decisioni: questo deve prima di tutto avere alcune preferenze su tutti i possibili esiti a sua disposizione (un esito è la descrizione completa di uno stato). Per rappresentare e ragionare con le preferenze si ricorre alla **teoria delle utilità**, secondo la quale ogni stato ha per l'agente una determinata utilità.

Le decisioni saranno quindi prese da parte dell'agente facendo riferimento alla **teoria delle decisioni** = (teoria delle probabilità) + (teoria delle utilità).

Sulla base di queste definizioni possiamo assumere che *un agente è razionale se e solo se sceglie l'azione che porta alla più alta utilità attesa, calcolata sulla media di tutti i possibili esiti dell'azione*.

Notazione base della Teoria delle Probabilità

Il linguaggio che verrà introdotto rappresenta una naturale estensione di quello visto nella logica proposizionale.

I gradi di credenza sono sempre attribuiti ad una **proposizione**, che sono descritte come enunciati che affermano che qualcosa è verificato.

L'elemento base del linguaggio è la **variabile casuale** che si può immaginare come “una parte” del mondo. Le variabili casuali possono essere suddivise in tre categorie in base alla struttura del proprio dominio: **booleans**, **discrete**, o **continue**.

Un altro concetto molto utile è quello di **evento atomico**: consiste in una specifica completa dello stato del mondo di cui l'agente è incerto (ovvero un assegnamento di

particolari valori a tutte le variabili casuali del dominio). Gli eventi atomici hanno alcune proprietà:

- Sono *mutuamente esclusivi*: in ogni istante di tempo se ne può verificare soltanto uno.
- L'insieme di tutti gli eventi atomici è *esaustivo*: uno di essi sarà sempre verificato.
- Da ogni particolare evento atomico si può derivare la verità o falsità di ogni proposizione.
- Ogni proposizione è logicamente equivalente alla disgiunzione di tutti gli eventi atomici che implicano la verità della proposizione stessa.

La **probabilità a priori** associata con una proposizione a è il grado di credenza che le viene accordato *in assenza di ogni altra informazione* e viene indicata con $P(a)$.

La **distribuzione di probabilità** di una variabile casuale rappresenta l'insieme (o vettore) delle sue probabilità per ogni valore del suo dominio (es: $P(CondAtm) = \langle 0.7, 0.2, 0.08, 0.02 \rangle$).

Spesso vogliamo considerare le probabilità di tutte le combinazioni di valori di un insieme di variabili casuali (es. $P(CondAtm, Carie)$) e prende il nome di **distribuzione di probabilità congiunta**. Se si considera la distribuzione congiunta di tutte le variabili casuali presenti nel nostro mondo, questa prende il nome di **distribuzione di probabilità congiunta completa**.

Una volta che l'agente ottiene informazioni attraverso delle prove che precedentemente erano sconosciute, si devono utilizzare le **probabilità condizionate**, indicate con $P(a | b)$. Un'equazione molto importante per la probabilità condizionata:

$$P(a | b) = \frac{P(a, b)}{P(b)}.$$

Questa formula può essere riscritta $P(a, b) = P(a | b)P(b)$ e prende il nome di **regola del prodotto**.

Assiomi della Teoria della Probabilità

1. $0 \leq P(a) \leq 1$
2. Le proposizioni necessariamente vere hanno probabilità 1, quelle false hanno probabilità 0.
3. $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$ (principio di inclusione/esclusione)

Inferenza basata su distribuzioni congiunte complete

Vediamo un metodo per l'**inferenza probabilistica**, ovvero il calcolo delle probabilità a posteriori di proposizioni partendo dalle prove osservate. Come "base di conoscenza" si utilizzerà una distribuzione congiunta completa, da cui sarà possibile derivare le risposte a tutte le domande.

Un'attività particolarmente comune è l'estrazione della distribuzione su un sottoinsieme particolare di variabili o una variabile singola. Ad esempio, sommare tutte le probabilità della prima riga della tabella ci fornisce la **probabilità marginale** di *Carie*:

$$P(Carie) = 0,108 + 0,012 + 0,072 + 0,008 = 0,2$$

	<i>toothache</i>		\neg <i>toothache</i>	
	<i>catch</i>	\neg <i>catch</i>	<i>catch</i>	\neg <i>catch</i>
<i>cavity</i>	0.108	0.012	0.072	0.008
\neg <i>cavity</i>	0.016	0.064	0.144	0.576

Figure 13.3 A full joint distribution for the *Toothache*, *Cavity*, *Catch* world.

Questo processo è detto **marginalizzazione**, e può essere espresso attraverso la regola generale per qualsiasi coppia di insiemi di variabili Y e Z:

$$P(Y) = \sum_z P(Y, z)$$

Una variante sfrutta la regola del prodotto e esplicita le probabilità condizionate:

$$P(Y) = \sum_z P(Y | z)P(z) \text{ e prende il nome di } \mathbf{condizionamento}.$$

A partire dalla marginalizzazione possiamo estrapolare una procedura di inferenza generale. Prendiamo in esame il caso in cui la query riguardi una sola variabile.

Siano: X la variabile di interrogazione, e le evidenze osservate e Y le restanti variabili non

$$P(X | e) = \alpha P(X | e) = \alpha \sum_y P(X, e, y) \quad [31.1]$$

osservate. La query sarà quindi del tipo $P(X | e)$:

Esempio del Wumpus rivisitato

Brevemente si riprende l'esempio del Wumpus per vedere un'applicazione della formula [31.1] utilizzata per rispondere alle query.

La situazione è quella rappresentata in foto e valgono le formule:

$$B = \neg b_{1,1} \wedge b_{1,2} \wedge b_{2,1}$$

$$\text{conosciute} = \neg p_{1,1} \wedge \neg p_{1,2} \wedge \neg p_{2,1}$$

La query che ci interessa è

$$P(p_{1,3} | \text{conosciute}, B) = ?$$

Applicando la [31.1] ottengo:

$$P(p_{1,3} | \text{conosciute}, B) = \alpha \sum_{\text{sconosciute}} P(p_{1,3}, \text{sconosciute}, \text{conosciute}, B)$$

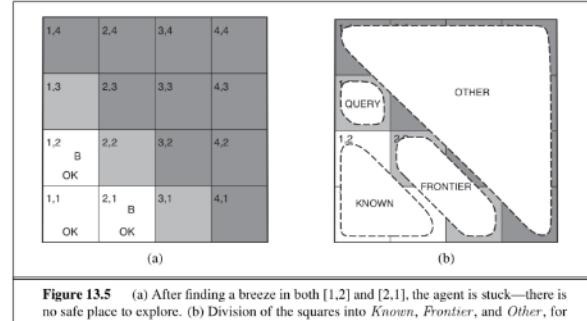


Figure 13.5 (a) After finding a breeze in both [1,2] and [2,1], the agent is stuck—there is no safe place to explore. (b) Division of the squares into *Known*, *Frontier*, and *Other*, for a query about [1,3].

Indipendenza

L'**indipendenza** è una proprietà delle variabili casuali che si verifica quando, date a e b :

$$P(a | b) = P(a) \text{ oppure } P(a \wedge b) = P(a)P(b).$$

Si tratta di una proprietà importante perché se l'insieme di variabili può essere suddiviso in sottoinsiemi indipendenti, la distribuzione completa può essere fattorizzata in distribuzioni congiunte separate sui vari sottoinsiemi.

La regola di Bayes e il suo utilizzo

Sfruttando la regola del prodotto, possiamo arrivare a scrivere $P(b | a) = \frac{P(a | b)P(b)}{P(a)}$

che prende il nome di **regola di Bayes**.

La regola di Bayes risulta molto utile nella pratica perché per ciascuno dei tre termini richiesti in molti casi si ha una buona stima. In attività come la diagnosi medica molto spesso si hanno a disposizione le probabilità condizionate delle relazioni causali e vogliamo derivare da esse una diagnosi. Facendo un esempio per chiarire il concetto, possiamo dire che un medico sa che la *meningite* causa il *torcicollo* nel 50% dei casi, così come conosce la probabilità che un paziente abbia la meningite e quella che abbia il torcicollo. Con questi tre dati è possibile calcolare $P(m | s) = \frac{P(s | m)P(m)}{P(s)} = 0,0002$.

A partire quindi dalla conoscenza dell'effetto causato dalla malattia, possiamo conoscere la malattia partendo dall'effetto.

Possiamo denominare i fattori all'interno della regola di Bayes nel seguente modo:

$$P(A | B) = \frac{P(B | A) P(A)}{P(B)}$$

Likelihood:
Probabilità di osservare il sintomo sapendo la malattia

Posterior:
Probabilità della malattia dopo le osservazioni (sintomi)

Prior:
Probabilità della malattia a priori, senza osservazioni

Possiamo notare la relazione di proporzionalità diretta tra posterior e prior:

$$\text{Posterior} \propto \text{Likelihood} \cdot \text{Prior}$$

Per ottenere risultati più consistenti spesso è utile sfruttare la **normalizzazione**, ovvero moltiplichiamo ogni risultato per un fattore α , in modo tale che la somma totale degli elementi di $P(A | B)$ sia 1:

$$P(A | B) = \alpha P(B | A) P(A)$$

Abbiamo visto come possiamo sfruttare un elemento di prova del tipo $P(\text{effetto} | \text{causa})$ per ottenere una probabilità del tipo $P(\text{causa} | \text{effetto})$. Ma se dispongo di più di un elemento di prova?

$$\text{In tal caso la formula diventa: } P(A | C, B) = \frac{P(C | A, B) P(A | B)}{P(C | B)}.$$

Osserviamo che cambiano il posterior, la likelihood e il prior, ma la relazione di

$$\text{Posterior}(t) \propto \text{Likelihood}(t) \cdot \text{Posterior}(t - 1)$$

proporzionalità possiamo scriverla a partire da quella precedente come:

Esempio

Generalizziamo l'esempio di un agente con sensori immerso in un ambiente che è in grado di registrare un'evidenza e compiere un'azione.

In particolare consideriamo:

- Azione a
- Evidenza e
- $s = \text{Result}(a)$
- U : funzione di probabilità che ha come dominio le azioni e rappresenta la loro utilità

$$\text{Allora vale } \sum_s P(\text{Result}(a) = s | a, e) U(s) = E[U(a | e)]$$

Molto spesso si ha che una singola causa influenza direttamente più effetti, tutti condizionalmente indipendenti data la causa. La distribuzione congiunta completa la si può scrivere:

$$P(\text{causa}, \text{effetto}_1, \dots, \text{effetto}_n) = P(\text{causa}) \prod_i P(\text{effetto}_i | \text{causa})$$

Una simile distribuzione di probabilità prende il nome di modello **naive Bayes**.

Ragionamento Probabilistico

Vediamo come sfruttare le relazioni probabilistiche viste rappresentandole sotto forma di **reti Bayesiane**.

Una rete bayesiana rappresenta le dipendenze tra le variabili e fornisce una specifica concisa di qualsiasi distribuzione di probabilità congiunta completa. Consiste in un grafo orientato aciclico in cui ogni nodo è etichettato con un'informazione probabilistica quantitativa.

La specifica completa è la seguente:

- I nodi della rete sono costituiti da un insieme di variabili casuali
- Un insieme di archi orientati collega coppie di nodi. Se X è collegato a Y diciamo che X è genitore di Y .
- Ogni nodo X_i ha una distribuzione di probabilità condizionata $P(X_i | \text{parents}(X_i))$ che quantifica gli effetti dei genitori sul figlio.
- Il grafo non ha cicli (è un DAG)

Ogni elemento della distribuzione di probabilità congiunta può essere calcolato a partire dall'informazione contenuta nella rete. Un elemento generico della distribuzione congiunta indica la probabilità di una congiunzione di assegnamenti. Il valore di quest'elemento è dato da:

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i | \text{parents}(x_i)) \quad [33.1]$$

Nel caso discreto una distribuzione è una tabella.

Esempio 1

$D \rightarrow A \leftarrow B \leftarrow C$ lo scrivo come $P(A, B, C, D) = P(A | B, D) P(B | C) P(C) P(D)$

L'equazione [33.1] definisce il significato di una rete bayesiana. Vediamo però come arrivarci in modo tale che rappresenti bene il dominio di riferimento.

A partire dalla distribuzione congiunta, utilizzando la regola del prodotto otteniamo che

$$P(x_1, \dots, x_n) = P(x_n | x_{n-1}, \dots, x_1) P(x_{n-1}, \dots, x_1)$$

Continuando ad applicare in modo iterativo la regola del prodotto otteniamo

$$P(x_1, \dots, x_n) = P(x_n | x_{n-1}, \dots, x_1) P(x_{n-1}, \dots, x_1) \dots P(x_2 | x_1) P(x_1) = \prod_{i=1}^n P(x_i | x_{i-1}, \dots, x_1)$$

Questa identità è detta **chain rule**.

Confrontandola con la [33.1] notiamo che è equivalente all'asserzione generale sulla rete per cui per ogni variabile X_i , si ha

$$P(X_i | X_{i-1}, \dots, X_1) = P(X_i | \text{Parents}(X_i)), \quad [33.2]$$

a patto che $\text{Parents}(X_i) \subseteq \{X_i, \dots, X_1\}$. Quest'ultima condizione è soddisfatta etichettando i nodi in qualsiasi ordine consistente con l'ordinamento parziale implicito nella struttura del grafo.

Il senso dell'equazione [33.2] è che la rete bayesiana è una rappresentazione corretta del dominio solo se ogni nodo è condizionalmente indipendente dai suoi predecessori, dati i suoi genitori.

Per costruire correttamente una rete bayesiana dobbiamo per ogni nodo X_i scegliere opportunamente i genitori tra i nodi X_{i-1}, \dots, X_1 .

Dati $X, Y, Z \subset U$, diciamo che X è **condizionalmente indipendente** da Y dato Z e scriviamo $X \perp Y | Z$ se $P(X | Y, Z) = P(X | Z)$. In questo caso si tratta di una relazione ternaria tra variabili casuali.

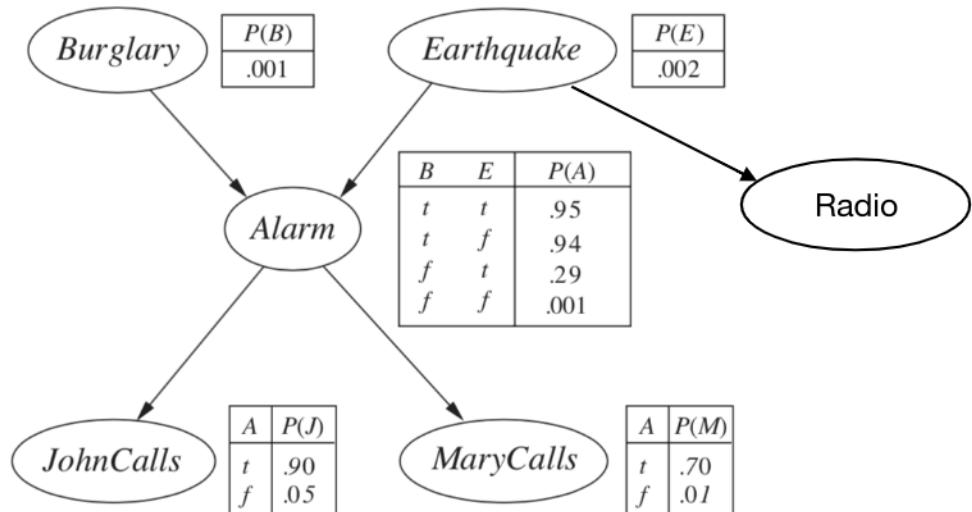
Se invece abbiamo che $P(A | B) = P(A)$, allora scriviamo $A \perp B | \emptyset$ e diciamo che A e B sono **marginalmente indipendenti**.

Riprendendo l'esempio 1 diciamo che $P(A | B, C) = P(A | B)$, quindi vale $A \perp C | B$.

Dati $X, Y, Z \subset U$, diciamo che X è **d-separato** da Y dato Z e scriviamo $X \perp_D Y | Z$. In questo caso si tratta invece di una relazione ternaria tra vertici (in un grafo).

Grazie alla struttura delle rete bayesiana è possibile partire da una semantica topologica che specifica le relazioni di indipendenza condizionale codificate nella struttura del grafo, e da esse ricavarne una semantica "numerica". Vediamo come ricavare le indipendenze condizionali a partire dal grafo sfruttando un esempio.

Esempio 2



- **Connessione consecutiva** si ha nel caso $B \rightarrow A \rightarrow J : B \perp_D J | A$.

Infatti:

$$P(J | B, A) = \frac{P(J, B, A)}{P(B, A)} = \frac{P(J | A)P(A | B)P(B)}{P(A | B)P(B)} = P(J | A)$$

- **Connessione divergente** si ha nel caso $A \leftarrow E \rightarrow R : A \perp_D R | E$.

- **Connessione convergente** si ha nel caso $B \rightarrow A \leftarrow E : B \perp_D E | \emptyset$ e si ottiene marginalizzando $P(B, A, E)$ rispetto ad A .

In generale, dati X, Y e Z insiemi di nodi:

$X \perp_D Y | Z$ si dice **d-separated** se e solo se ogni cammino tra X ed Y è bloccato da Z .

Un cammino π si dice **bloccato** da Z se e solo se almeno una delle seguenti condizioni è vera:

1. π ha un vertice in Z che non rappresenta un nodo collisione.
2. \exists un vertice C per π che è un nodo collisione per π , tale che $C \notin Z$ e nessun discendente di C è in Z

Teorema

Se G è un DAG per una distribuzione P , si ha $P(x_1, \dots, x_n) = \prod_i P(x_i | Parents(x_i))$.

Dati $X, Y, Z \subset U$, con $U = \{x_1, \dots, x_n\}$, allora $X \perp_D Y | Z \Rightarrow P(X | Y, Z) = P(X | Z)$.

All'interno di una rete bayesiana gli elementi della produttoria sopra sono rappresentati attraverso le **CPT** (Conditional Probability Table).

Inferenza nelle reti bayesiane

Il compito di ogni sistema di inferenza probabilistica è calcolare la distribuzione di probabilità a posteriori di un insieme di variabili, dette **variabili di query**, dato qualche evento osservato.

Tipicamente un'interrogazione riguarderà una distribuzione di probabilità a posteriori $P(X | e)$, con X variabile di query ed e evento osservato, come già visto in precedenza.

Possiamo definire due tipi di inferenza:

1. Approssimata

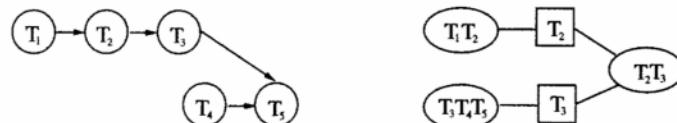
2. Esatta, che è NP-COMPLETE:

- Attraverso l'uso di alberi, quindi grafi aciclici non orientati, in un tempo polinomiale.
- Attraverso DAG, con l'uso di un algoritmo che sia in grado di far comunicare i nodi di un Junction Tree (NB: i nodi di un Junction Tree sono cluster).

Esempio (Junction Tree)

Un Junction Tree, come già visto, è un albero particolare che rispetta le proprietà:

- I nodi sono sottoinsiemi di U , dove U è l'insieme delle variabili.
- I collegamenti (gli archi) sono etichettati attraverso i *separators*.
- \forall variabile A in U è necessario almeno un cluster che contenga A e $Pa(A)$.
- Proprietà *running intersection*: $\forall v, w$ ogni nodo o separator nel cammino che connette v e w deve contenere $v \cap w$



Grafo (sx) e Junction Tree corrispondente (dx)

Vediamo alcuni algoritmi per il calcolo dell'inferenza esatta ed esaminiamo la complessità di questo compito.

Algebra sulle tavole

Si introduce brevemente l'operazione di moltiplicazione sulle tavole, già implicitamente trattata.

Siano t e t' due tavole sulle stesse variabili. Allora il prodotto delle due, che copre tutte le configurazioni, indicate con c^* , sarà dato da $t \cdot t'(c^*) = t(c^*) \cdot t'(c^*)$.

Esempio:

a_1	a_2	a_3	a_1	a_2	a_3	a_1	a_2	a_3
b_1	x_1	x_2	x_3	b_1	x'_1	x'_2	x'_3	b_1
b_2	y_1	y_2	y_3	b_2	y'_1	y'_2	y'_3	b_2
b_3	z_1	z_2	z_3	b_3	z'_1	z'_2	z'_3	b_3
t			t'			$t \cdot t'$		

Supponendo invece che si abbiano due tavole su insiemi di variabili diverse, quali $t_{A,B}$ su $\{A, B\}$ e $t_{A,C}$ su $\{A, C\}$, la tabella risultante da una loro moltiplicazione sarà $t_{A,B,C}$.

Esempio:

A	B	$f_1(A, B)$	B	C	$f_2(B, C)$	A	B	C	$f_3(A, B, C)$
T	T	.3	T	T	.2	T	T	T	.3 × .2 = .06
T	F	.7	T	F	.8	T	T	F	.3 × .8 = .24
F	T	.9	F	T	.6	T	F	T	.7 × .6 = .42
F	F	.1	F	F	.4	T	F	F	.7 × .4 = .28
						F	T	T	.9 × .2 = .18
						F	T	F	.9 × .8 = .72
						F	F	T	.1 × .6 = .06
						F	F	F	.1 × .4 = .04

Figure 14.10 Illustrating pointwise multiplication: $f_1(A, B) \times f_2(B, C) = f_3(A, B, C)$.

È poi possibile **marginalizzare** i valori della tabella risultante rispetto ad una o più variabili $S \subset V$ sommando le righe al variare di $V - S$, ovvero:

$$t_S = \sum_{V-S} t_v$$

Riprendendo l'esempio sopra, se $V = \{A, B, C\}$ e $S = \{A, B\}$, allora $V - S = \{C\}$,

$$\text{quindi } t_S = t_{\{A,B\}} = \sum_C t_v = <.06 + .42 + .18 + .06, .24 + .28 + .72 + .04>$$

Inferenza per Enumerazione

Come abbiamo già detto, una rete bayesiana fornisce una rappresentazione di una distribuzione congiunta completa. L'equazione [31.1] mostra che i fattori della distribuzione possono essere scritti in termini di probabilità condizionate note alla rete.

Un algoritmo può fare questo enumerando gli elementi della distribuzione congiunta completa. Prende in input una rete bayesiana ed una query.

Prendiamo ad esempio una query relativa alla rete dell'Esempio 2:

$$P(\text{Intrusione} \mid \text{JohnTel} = \text{True}, \text{MaryTel} = \text{True})$$

In tal caso le variabili nascoste sono Terremoto e Allarme.

$$\text{Ottengo } P(I \mid j, m) = \alpha P(I, j, m) = \alpha \sum_t \sum_a P(I, j, m, t, a)$$

E attraverso la semantica delle reti bayesiane sfrutto le CPT attraverso l'equazione [31.1] :

$$P(i|j, m) = \alpha \sum_t \sum_a P(i)P(t)P(a|t, i)P(j|a)P(m|a)$$

Dato che ciascuna variabile è booleana, le somme su t e a produrranno due addendi ciascuna (si deve considerare entrambi i valori: Terremoto=True e Terremoto=False). Nel caso pessimo avrà da sommare, anche raccogliendo i termini non coinvolti nelle somme, per una rete a n variabili $O(n2^n)$ valori.

Quando si vuole calcolare la probabilità a posteriori di tutte le variabili di una rete, l'uso dei Junction Tree può migliorare la complessità da $O(n^2)$ a $O(n)$.

Esempio di trasformazione da DAG a Junction Tree:

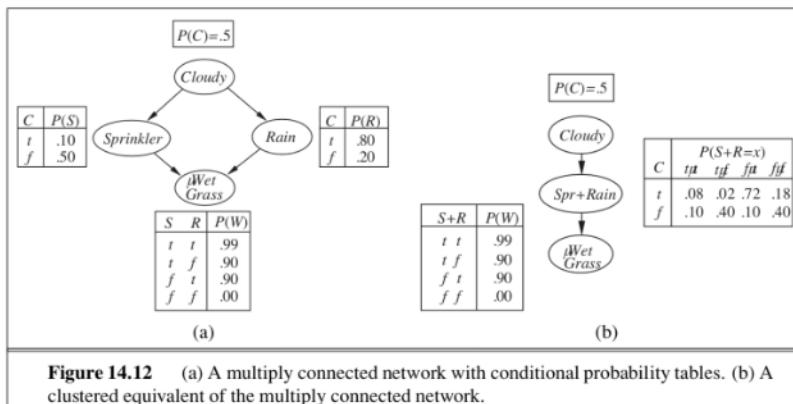


Figure 14.12 (a) A multiply connected network with conditional probability tables. (b) A clustered equivalent of the multiply connected network.

Una volta trasformata la rete in modo adeguato si applica un'algoritmo di inferenza. Viene accennato sotto e si tratta di un algoritmo di propagazione delle costanti i cui vicoli assicurano che cluster adiacenti concordino sulle probabilità a posteriori di tutte le variabili in comune.

Si possono sfruttare le tabelle in corrispondenza di ogni nodo del Junction Tree che hanno almeno una colonna in comune ed eseguire un prodotto tra le due, in cui ogni riga sarà il prodotto di due valori, uno per ciascuna delle due tabelle.

I passi dell'algoritmo sono i seguenti:

1. Inizializzazione

- 1) Per ogni V , assegno $t_v = 1$
- 2) Per ogni S , assegno $t_s = 1$
- 3) Per ogni $A \subset U$ scelgo un (e uno solo) clauster $V = \{A\} \cup Par(A)$, ovvero eseguo

$$t_v = t_v \cdot P(A | Par(A))$$

Esempio:



Per il passo 3), dato il Junction Tree in foto, ottengo:

$$t_v = 1 \cdot P(X) \cdot P(Y|X)$$

$$t_w = 1 \cdot P(Z|X)$$

$$P(U) = P(X, Y, Z) = P(X)P(Y|Z)P(Z|X) = \frac{t_v \cdot t_w}{t_s}$$

2. Assorbimento

Scambio messaggi tra i nodi.

Calcolo:

$$1) \quad t_s^* = \sum_{V-S} t_v$$

$$2) \quad t_w^* = t_w \frac{t_s^*}{t_s}$$

$$3) \quad t_s = t_s^*$$

Quindi nel passo 1) t_s diventa t_s^* creando il messaggio marginalizzando t_v , nel passo

2) t_w assorbe il messaggio, ovvero t_v

Un collegamento $v \rightarrow w$ si dice consistente (concetto di **consistenza locale**) se e solo se $\sum_{V-S} t_v = t_s = \sum_{W-S} t_w$; in altre parole se la marginalizzazione rispetto ad una stessa variabile è uguale.

Un Cluster Tree si dice consistente (concetto di **consistenza globale**) se

$$\sum_{V-S} t_v = \sum_{W-S} t_w \quad \forall S = V \cap W.$$

Quindi (consistenza locale)+(running intersection) \Rightarrow Consistenza Globale

3. Algoritmo

Si sceglie un nodo come radice, poi si hanno due passaggi:

1. **Collect Evidence**: eseguo l'assorbimento dalle foglie alle radici

2. **Distribute Evidence**: dato che ora la radice ha tutte le informazioni, le distribuisce alle foglie

Adesso si ha la consistenza globale