

Fondamenti di Intelligenza Artificiale

Appunti di Niccolò Della Rocca

Università degli studi di Firenze

21 Settembre 2020 \rightarrow 3 Dicembre 2020

Indice

21 Settembre 2020	1
1.1 Introduzione all'Intelligenza Artificiale	1
1.2 Problemi difficili e problemi intrattabili	1
1.2.1 Problemi intrattabili	2
24 Settembre 2020	5
2.1 Agenti ed Ambienti	5
2.1.1 Agenti razionali	6
2.1.2 Ambienti di lavoro	6
2.1.3 Struttura di un agente	8
2.2 Introduzione ai problemi di ricerca	10
28 Settembre 2020	13
3.1 Risolvere problemi di ricerca	13
3.2 Questioni implementative	14
3.3 Confronto tra algoritmi	15
3.4 Ricerca non informata	16
3.4.1 Ricerca in ampiezza (BFS)	16
3.4.2 Ricerca in profondità (DFS)	17
3.4.3 Uniform Cost Search	20
3.4.4 Ricerca bidirezionale	21
3.4.5 Confronto tra strategie di ricerca non informate	22
1 Ottobre 2020	23
4.1 Algoritmi di ricerca informata	23
4.1.1 Greedy best-first search	23
4.1.2 Ricerca con A*	24
4.2 Progettare buone euristiche	27
5 Ottobre 2020	29
5.1 Progettare buone euristiche per A*	29
5.1.1 Dominanza tra euristiche	29
5.1.2 Pattern databases: sottoproblemi ed euristiche	30
5.1.3 Effective branching factor: confrontare due euristiche	31
5.2 Ricerca locale	32
5.2.1 Hill Climbing	33
5.2.2 Beam search locale	34
5.2.3 Simulated Annealing	35
15 Ottobre 2020	39
6.1 Problemi di soddisfacimento di vincoli	39
6.1.1 Inferenza in problemi CSP	39

19 Ottobre 2020	43
7.1 Programmazione a vincoli: Backtracking	43
7.1.1 Scelta della variabile da assegnare	44
7.1.2 Scelta dell'ordine dei valori	44
7.1.3 Inferenza	45
7.2 Sfruttare struttura del problema	45
7.2.1 Cutset conditioning	47
7.2.2 Problema duale	47
22 Ottobre 2020	51
8.1 Triangolazione di un grafo	51
8.2 Ricerca locale per programmazione a vincoli	53
8.3 Introduzione agli agenti logici	54
8.3.1 Generico programma agente logico	55
26 Ottobre 2020	57
9.1 Mondo dello Wumpus	57
9.2 Elementi di logica	58
9.2.1 Entailment	58
9.2.2 Logica proposizionale	59
9.2.3 Theorem Proving	62
29 Ottobre 2020	67
10.1 Completezza di TT-RESOLUTION	67
10.2 Forward e backward chaining	69
10.2.1 Clausole definite e clausole di Horn	69
10.2.2 Algoritmi di inferenza per clausole di Horn	70
10.3 DPLL: Un algoritmo per SAT	71
10.4 Ricerca locale stocastica per SAT	73
2 Novembre 2020	77
11.1 Probabilità e assiomi di Kolmogorov	77
11.2 Beliefs, probabilità soggettive	78
11.2.1 Assiomi di Cox e Jaynes	78
11.2.2 Credenze secondo De Finetti	79
11.3 Ragionamento probabilistico secondo Bayes	80
11.3.1 Indipendenze marginali e condizionali	81
5 Novembre 2020	85
12.1 Modelli grafici orientati: Reti di Bayes	85
12.2 D-separazione	89
9 Novembre 2020	95
13.1 Limiti della D-separazione e Reti di Markov	95
13.1.1 Modelli grafici non orientati: reti di Markov	95
13.2 Inferenza in reti di Bayes	97
13.2.1 Algebra su tabelle di probabilità	98
13.2.2 Junction tree e inizializzazione	100
13.2.3 Scambio di messaggi su junction tree	103

13.2.4 Costruzione di junction trees su grafi orientati	105
12 Novembre 2020	109
14.1 Apprendimento dei parametri in Reti di Bayes	109
14.2 Apprendimento bayesiano su reti di Bayes	111
19 Novembre 2020	115
15.1 Generalizzazione apprendimento dei parametri	115
15.2 Apprendimento della struttura	118
15.2.1 Campionamento di un dataset	120
15.3 Classificatore Naive Bayes	121
23 Novembre 2020	125
16.1 Laplace smoothing per Naive Bayes	125
16.2 Classificatore Naive Bayes multinomiale	126
16.2.1 Punto di break-even	127
16.3 Apprendimento supervisionato	129
16.3.1 Formalizzazione di Naive Bayes	130
26 Novembre 2020	133
17.1 Perceptron	133
17.1.1 Convergenza	135
17.1.2 Limiti e miglioramenti a Perceptron	138
30 Novembre 2020	141
18.1 Precisazioni su Perceptron	141
18.2 Alberi di decisione	142
18.2.1 Algoritmo basato su albero di decisione	143
3 Dicembre 2020	149
19.1 Potatura di alberi di decisione	149
19.1.1 Potatura di regole	150
19.2 Convalida incrociata K-fold	151
19.3 Bagging e Random Forest	152
2 Dicembre 2021	155
20.1 Multilayered Perceptron	155
20.2 Apprendimento in Feedforward Neural Networks	156
9 Dicembre 2021	161
21.1 Backpropagation per Multilayered Perceptron	161
21.2 Combinazione di classificatori semplici	163

21/09/2020

1.1 Introduzione all'Intelligenza Artificiale

In questa sezione si cerca di introdurre il lettore al concetto di intelligenza artificiale. Per rispondere alla domanda di cosa questa sia uno è chiamato anzitutto a cercare di definire il concetto di intelligenza, che di per sé è piuttosto complicato; noi cercheremo di dare un'impostazione più pratica che filosofica, ed una prima definizione operativa, anche storicamente, del concetto di intelligenza è stata la seguente:

Definizione 1.1 (Test di Turing). Si considera il seguente test: sono assegnati due interlocutori, A e B, di cui uno è un umano ed uno una macchina, con cui si possono scambiare informazioni (e.g. tramite chat di testo). Se non siamo in grado di distinguere chi tra A e B sia l'umano diciamo che la macchina è in grado di esibire un comportamento **intelligente**.

Al di là delle possibili altre definizioni, rimarchiamo che l'obiettivo scientifico dell'intelligenza artificiale è quello di comprendere i principi che rendono possibili comportamenti intelligenti, sia in sistemi naturali che artificiali. Uno degli aspetti formali che vedremo fin da subito è il cosiddetto **agente**, che per il momento possiamo immaginare come qualcosa che svolge un ruolo in qualche ambiente. In base a questa definizione il concetto di agente è estremamente ampio, e può comprendere sia agenti umani che software o di varia natura; ovviamente tra tutti quelli esistenti ci interesseremo ad un sottogruppo, che sono gli *agenti computazionali* e quelli *intelligenti*. Affinché un agente si possa considerare come intelligente occorre che questo esegua le azioni appropriate in base agli obiettivi posti ed alle circostanze ambientali, tenendo conto delle conseguenze delle proprie azioni a lungo e breve termine; in pratica si definisce una misura di prestazioni che quantifica la bontà delle decisioni dell'agente, e spesso i problemi in intelligenza artificiale si riconducono alla minimizzazione o massimizzazione di questa metrica. Invece per **agente computazionale** intendiamo un agente le cui azioni possono essere suddivise in operazioni primitive che siano implementabili su un dispositivo controllato da software. Come modelli astratti di sistemi computazionali spesso si considerano la macchina di Turing oppure il λ -calcolo (*Alonso Church*).

1.2 Problemi difficili e problemi intrattabili

Due dei problemi che ci si deve porre quando si vuole risolvere un nuovo problema sono se taluno ammette una soluzione algoritmica ed in tal caso se ne ammette una efficiente. Rispondere a questa domanda è un problema dell'informatica teorica, qui ci limitiamo a riportare dei passaggi fondamentali senza approfondire eccessivamente. Il primo risultato interessante è il seguente:

Congettura 1.2.1 (Turing-Church). Ogni funzione calcolabile $f: \mathbb{N} \mapsto \{0, 1\}$ è anche calcolabile da una macchina di Turing.

Chiaramente il numero di funzioni fatte in questo modo è infinito in quanto sono infiniti gli argomenti che può prendere la funzione. Al di là di questa intuizione si può far vedere che l'insieme di queste funzioni non è numerabile, bensì ha la stessa cardinalità di \mathbb{R} . Di seguito esploriamo le conseguenze di questa semplice osservazione, che come vedremo sono tutt'altro che marginali.

Immaginiamo di aver assegnato un alfabeto Σ , ad esempio

$$\Sigma = \{a, b, \dots, z, A, B, \dots, Z, 0, \dots, 9, +, -, \dots\}$$

Allora un particolare programma non è altro che una stringa su questo alfabeto; ciascuna di queste può essere espressa come una sequenza di bit che a sua volta possiamo interpretare come un numero naturale, per cui i diversi programmi che possono essere scritti sono in quantità numerabile. Poiché le funzioni $f: \mathbb{N} \mapsto \{0, 1\}$, dette “di **decisione**”, per l'osservazione precedente sono un'infinità non numerabile, esistono dei problemi che non sono decidibili, ossia tali per cui non sia possibile calcolare la funzione corrispondente alla sua soluzione.

1.2.1 Problemi intrattabili

Non solo vogliamo dei problemi che siano calcolabili, ma vorremmo anche che lo fossero in maniera efficiente. Poiché la totalità dei problemi che verranno affrontati durante il corso appartengono a questa categoria, qui spieghiamo in breve cosa si intende quando si parla di problema *intrattabile*.

Si indica con P l'insieme dei problemi **risolvibili** in tempo polinomiale; assegnato un problema in questa classe e detta n la sua dimensione, il tempo di esecuzione è un $T(n) = O(n^k)$ con k fissato. Ad esempio il problema di ordinare un vettore di dimensione n tramite confronti ha un costo minimo che è un $\Omega(n \log n)$ e quindi sicuramente ne fa parte. Enfatizziamo inoltre che l'appartenenza o meno a questa classe è una proprietà del problema e non del particolare algoritmo. A fianco di questa troviamo l'insieme NP dei problemi **verificabili** in tempo polinomiale: assegnata una candidata soluzione per uno di questi deve esistere un algoritmo che in tempo polinomiale è in grado di controllare se questa è effettivamente (o meno) una soluzione. Un'annosa questione nell'informatica teorica è se $P=NP$; la convinzione più diffusa è che in effetti tale uguaglianza non valga, ma nessuno è mai riuscito a dimostrare né in uno né nell'altro senso.

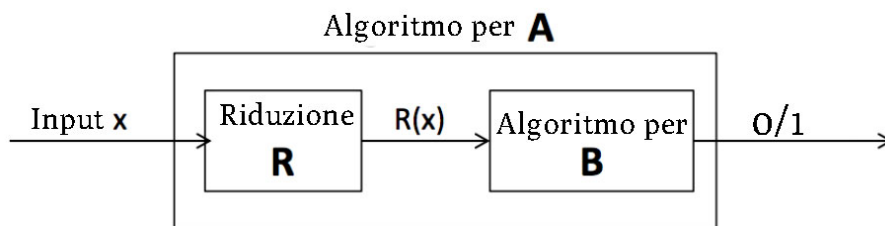
Problemi NP-Completi

Esiste un sottoinsieme di NP che c'interessa particolarmente, che è quello dei cosiddetti problemi NP-Completi, che introduciamo a seguire. Esiste una relazione tra problemi detta “riducibilità di Karp”:

Definizione 1.2 (Riducibilità Di Karp). Siano $A, B: \mathbb{N} \mapsto \{0, 1\}$ due problemi di decisione, allora diciamo che A è **Karp-riducibile** a B , e scriviamo che $A \leq_K B$, se esiste una funzione $R: \mathbb{N} \mapsto \mathbb{N}$ che trasforma un input per A in un input per B in modo tale che $\forall x \in \mathbb{N} \quad A(x) = B(R(x))$.

Per fare un parallelo, è come se si sviluppasse un algoritmo B da utilizzare come sotto-programma per risolvere il problema A . La figura seguente illustra il concetto. In pratica, se tra due problemi sussiste tale relazione esiste un modo per trasformare un'istanza del

primo in una del secondo. In questo contesto se è noto un algoritmo che risolve il problema B ed è stato sviluppato un modo per passare da un'istanza di A ad una di B allora si è ottenuto un risolutore del problema A .



In pratica molti dei problemi che vediamo possono essere ridotti rispetto alla precedente definizione a quello della soddisfacibilità booleana, che tratteremo più avanti nel corso. Se la funzione R che effettua la trasformazione delle istanze si può calcolare in tempo polinomiale allora si scrive che $A \leq_p B$. In questo contesto diamo la seguente definizione:

Definizione 1.3 (Problema NP-Completo). Sia B un problema assegnato, allora si dice che esso è **NP-Completo** se valgono entrambe le seguenti condizioni:

- (i) $B \in NP$
- (ii) B è NP-hard, ossia $A \leq_p B \quad \forall A \in NP$

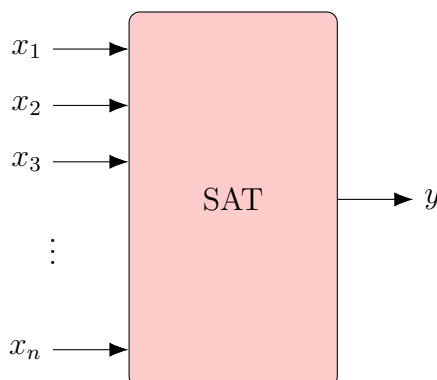
Un problema NP-hard è tale se qualsiasi problema in NP può essere ridotto ad esso in tempo polinomiale. Legate a queste definizioni possiamo far vedere che valgono i risultati che riportiamo a seguire:

Teorema 1.2.1. Se esiste un problema B NP-completo tale che $B \in P$ allora $P=NP$

Il risultato è piuttosto intuitivo: se esistesse un B che verifica le ipotesi qui sopra allora tutti gli altri potrebbero essere risolti in tempo polinomiale tramite la riduzione a B . Questo teorema giustifica anche il perché della scetticità della comunità informatica sul fatto che $P=NP$, infatti in molti anni che problemi NP-completi sono oggetti di studio nessuno è ancora riuscito ad esibire un algoritmo che risolva uno di questi in tempo polinomiale; se qualcuno riuscisse allora in un colpo solo per riduzione riuscirebbe a risolvere in tempo polinomiale tutti i problemi NP.

Teorema 1.2.2. Il problema della soddisfacibilità booleana, detto SAT, è NP-Completo

Il teorema precedente garantisce l'esistenza di almeno un problema NP-completo. A seguire illustriamo in breve l'idea alla base di SAT:



Assegnato un insieme di simboli booleani x_1, x_2, \dots, x_n si cerca un'assegnazione di questi tale da rendere y vero, i.e. con valore 1.

24/09/2020

2.1 Agenti ed Ambienti

Quando parliamo di **agenti** ci riferiamo a qualsiasi cosa si possa immaginare che sia in grado di percepire l'ambiente in cui agisce attraverso dei **sensori** e modificare lo stesso tramite degli **attuatori**. Un agente ad ogni istante di tempo può ricevere delle **percezioni**: chiamiamo così tutto ciò che l'agente può acquisire attraverso i propri sensori; oltretutto al generico tempo t l'agente può scegliere di eseguire un'azione sulla base di tutta la sequenza di percezioni ricevuta fino a quell'istante. Un possibile modo di rappresentare un agente quindi potrebbe essere di assegnare una funzione che associa ad ogni possibile sequenza di percezioni l'azione corrispondente, e si parla allora di **funzione agente**; tuttavia è complicato farlo in quanto se volessimo tabulare questa funzione avremmo bisogno di infinite righe, a meno che non si pongano dei limiti alla lunghezza della sequenza di percezioni che si vuole considerare. La funzione agente è soltanto una caratterizzazione esterna dell'agente, cioè un'astrazione matematica che ne descrive il comportamento; in pratica la funzione agente sarà implementata internamente sotto forma di un **programma agente**.

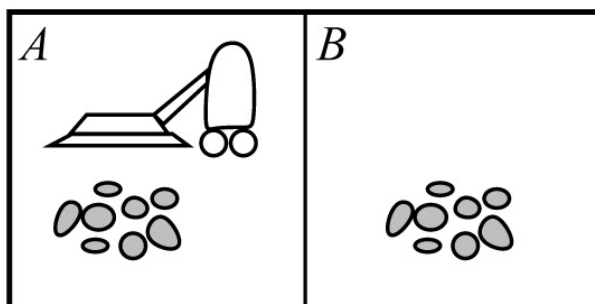


Figura 2.1: Agente ed ambiente del mondo dell'aspirapolvere

Per avere un esempio pratico a cui fare riferimento, introduciamo di seguito il “mondo dell'aspirapolvere automatizzato”: questo consiste in due stanze rettangolari, nominate rispettivamente A e B. L'aspirapolvere è in grado di percepire in quale stanza si trova e se è pulita o sporca, inoltre può decidere di spostarsi a destra, a sinistra, aspirare oppure non fare niente. La sua funzione agente potrebbe apparire come illustrato di seguito. Si noti che una funzione agente è appunto una funzione, nel senso matematico del termine: per assegnarla dobbiamo specificare l'uscita per ogni ingresso immaginabile, al di là del fatto che questo si possa o meno veramente presentare; è il caso ad esempio di $[A, pulito]$, $[A, pulito]$, che dev'essere presente in tabella nonostante se A sia pulito si vada in B e quindi non si riceverà mai questa sequenza.

Una domanda naturale da porsi è come riempire questa tabella di modo da avere il comportamento “migliore” possibile, e dare una risposta sarà obiettivo della prossima sezione.

Sequenza di percezioni	Azione
$[A, pulito]$	<i>Destra</i>
$[A, sporco]$	<i>Aspira</i>
$[B, pulito]$	<i>Sinistra</i>
$[B, sporco]$	<i>Aspira</i>
$[A, pulito], [A, pulito]$	<i>Destra</i>
$[A, pulito], [A, sporco]$	<i>Aspira</i>
\vdots	\vdots
$[A, pulito], [A, pulito], [A, pulito]$	<i>Destra</i>
$[A, pulito], [A, pulito], [A, sporco]$	<i>Aspira</i>
\vdots	\vdots

2.1.1 Agenti razionali

Un agente si dice essere razionale se fa la cosa giusta: nel momento in cui questo viene inserito in un ambiente riceve delle percezioni ed effettua in loro conseguenza delle azioni che fanno in modo di far attraversare all'ambiente una serie di stati; se tale sequenza è vantaggiosa allora l'agente si è comportato bene. La nozione di vantaggiosità in pratica viene tradotta in una **misura di prestazioni**, che è una funzione in grado di valutare ogni possibile sequenza di stati dell'ambiente. Una linea guida da seguire è calibrare questa misura non sulla base di come si pensa che l'agente dovrebbe operare, bensì su ciò che si vuole che accada all'ambiente. Per dare un'idea del perché è importante seguire questa regola si pensi all'esempio dell'aspirapolvere: potremmo definire la sua misura di prestazioni come la quantità di sporco pulito in un determinato arco di tempo, ma a questo punto l'agente potrebbe massimizzare la propria prestazione pulendo una stanza, rovesciando lo sporco sul pavimento, pulendo di nuovo e così via. Un'idea migliore è quella di assegnare un punteggio basato su quante stanze sono pulite ad ogni istante di tempo.

Più formalmente, la razionalità di un agente dipende da fattori come la misura di prestazioni usata per definire la vantaggiosità dei vari stati, la conoscenza pregressa che l'agente ha dell'ambiente, le azioni che esso può compiere ed anche sulla sequenza di percezioni ricevuta. Diamo quindi una definizione che tenga conto di tutto questo:

Definizione 2.1 (Agente Razionale). Un **agente razionale** è un agente che per ogni possibile sequenza di percezioni seleziona l'azione che ci si aspetta massimizzi la misura di prestazioni, in base alla propria conoscenza pregressa ed alle informazioni portate dalla stessa sequenza di percezioni.

2.1.2 Ambienti di lavoro

Quando si parla di **ambiente di lavoro** (*task environment*) si fa riferimento a quattro fattori che entrano in gioco nella descrizione di un agente logico, che sono l'insieme dei suoi sensori ed attuatori, la misura di prestazioni ed una descrizione dell'ambiente. Nel progettare un agente il primo passo dev'essere proprio quello di specificare l'ambiente di

lavoro nel modo più dettagliato possibile. A seguire elencheremo alcune proprietà, non mutuamente esclusive, di cui un ambiente di lavoro potrebbe godere o meno.

Completa o parziale osservabilità

Se i sensori di un agente consentono di accedere completamente allo stato dell'ambiente ad ogni istante di tempo si dice che l'ambiente di lavoro è **completamente osservabile**; all'atto pratico per godere di questa proprietà è sufficiente che i sensori diano accesso a tutto ciò che sia rilevante per la scelta dell'azione da compiere. Questo tipo di ambiente è vantaggioso perché permette all'agente di non dover mantenere un proprio stato interno per tener traccia del mondo esterno. Se non si ha la completa osservabilità ma l'agente è comunque dotato di sensori si parla di parziale osservabilità; questa situazione può derivare da rumore nei sensori, dalla loro imprecisione e non ultimo alcuni elementi dello stato potrebbero essere non disponibili all'agente: nel caso dell'aspirapolvere ad esempio non è possibile sapere se ci sia o meno sporco nella stanza accanto perché i sensori riescono a percepire lo sporco solo nelle vicinanze.

Agente singolo o multiplo

Un ambiente si dice essere ad agente singolo se vi opera un solo agente, altrimenti si dice essere ad agente multiplo. La distinzione è netta e non richiede ulteriori precisazioni, in ogni caso è legittimo il dubbio su cosa si debba considerare come agente e cosa no. Se nell'ambiente oltre all'agente è presente un secondo oggetto, potremmo chiederci se conviene trattarlo come un mero oggetto o a sua volta come un agente; in questo caso ciò che fa la differenza è se il comportamento di quest'ultimo è descritto in maniera migliore come qualcosa che cerca di migliorare una misura di prestazioni che dipende dal comportamento del primo agente. Nel caso di un ambiente di lavoro multiagente si opera una distinzione ulteriore tra ambienti **competitivi**, in cui per massimizzare la propria misura di prestazioni si deve cercare di minimizzare quella degli altri agenti, oppure **collaborativi**, in cui gli agenti collaborano per raggiungere un fine comune.

Ambiente deterministico o stocastico

Un ambiente di lavoro si dice essere **deterministico** se lo stato futuro dell'ambiente è determinato unicamente dallo stato presente e dall'azione eseguita dall'agente in sua risposta, mentre se questo non accade si parla di ambiente di lavoro **stocastico**. Spesso nella pratica si possono presentare situazioni in cui gli aspetti non osservati di cui dover tener traccia sono talmente tanti che conviene modellare l'ambiente come stocastico. Se l'ambiente non è completamente osservato o non è deterministico si dice che esso è **incerto**. Infine, quando parliamo di un ambiente "stocastico" si assume che l'incertezza sui possibili esiti delle azioni è accompagnato da delle probabilità, mentre se per ogni azione è nota solo una lista di possibili esiti, ma senza una probabilità associata, si parla più in generale di ambiente **non deterministico**.

Ambiente episodico o sequenziale

Un ambiente di lavoro si dice essere **episodico** se l'esperienza dell'agente è divisa in episodi, in ciascuno dei quali l'agente riceve una percezione ed esegue una singola azione. La caratteristica fondamentale di tali episodi è che quanto accade in quello attuale non

dipende dalle azioni intraprese in quelli precedenti, ad esempio nel gioco degli scacchi partite diverse possono essere considerate come episodi. Invece gli ambienti di lavoro **sequenziali** sono caratterizzati da una esperienza continuativa in cui le decisioni prese adesso possono avere influenza nelle decisioni future, ad esempio una singola partita di scacchi è un ambiente di lavoro deterministico in quanto una mossa eseguita adesso può avere conseguenze anche a lungo termine.

Ambiente statico o dinamico

Se l'ambiente può cambiare mentre l'agente sta prendendo una decisione allora si parla di ambiente di lavoro **dinamico**, mentre in caso contrario si chiama **statico**. Il secondo tipo di ambiente è vantaggioso perché l'agente non deve tenere traccia del mondo circostante mentre decide e può ignorare il passaggio del tempo. Al contrario in un ambiente dinamico è come se all'agente venisse continuamente chiesto che azione intraprendere, e se una decisione non è ancora stata presa si considera come se l'agente avesse deciso di non fare niente.

2.1.3 Struttura di un agente

In questa sezione descriviamo vari possibili approcci alla progettazione della struttura di agenti. Precisiamo anzitutto che tutti questi hanno un'ossatura comune, infatti prendono tutti quanti come unico ingresso la percezione attuale e restituiscono un'azione agli attuatori; se per prendere una decisione un'agente ha necessità dell'intera sequenza di percezioni dovrà salvarli da qualche parte via via che arrivano.

Una prima banale struttura possibile è quella che chiamiamo **agente guidato da una tabella**: l'agente in questione mantiene traccia della sequenza di percezioni ricevuta e la usa per indicizzare una tabella, che rappresenta per esplicito la funzione agente che l'agente implementa. Uno pseudocodice di riferimento è il seguente:

Algoritmo 1 Agente basato su una tabella

```

1: function AGENTE-GUIDATO-DA-TABELLA(percezione)
2:   Aggiungi percezione alla lista di percezioni
3:   azione ← RICERCA(tabella, percezioni)
4:   return azione

```

Qui sopra, la lista di percezioni cui si fa riferimento è inizialmente vuota, mentre la tabella *tabella* è già completamente precompilata.

È evidente che questo tipo di agente sia destinato a rimanere soltanto qualcosa di teorico: è sufficiente un'applicazione pratica relativamente comune come il gioco degli scacchi per ottenere tabelle con un numero di righe dell'ordine di 10^{150} , talmente grande da essere impossibile da memorizzare su qualsiasi dispositivo pratico considerando che il numero stimato di atomi dell'universo osservabile è dell'ordine di "appena" 10^{80} . In ogni caso, anche se esistesse un dispositivo con tanta memoria nessun essere umano riuscirebbe a riempirla, l'agente stesso non sarebbe comunque in grado di apprendere tutte le righe in modo corretto tramite tecniche di apprendimento ed infine anche se tutti questi non fossero problemi e la tabella rimanesse di dimensioni contenute, comunque l'esperto umano chiamato a riempire una simile tabella non avrebbe nessuna linea guida

su come farlo. Nonostante tutte queste difficoltà, un agente basato su tabelle fa ciò che deve.

Agenti semplice a riflesso

Il tipo più semplice di agente implementabile nella pratica è quello che viene chiamato **agente semplice a riflesso** (*simple reflex agent*). Questo prende una decisione sull'azione da eseguire unicamente sulla base dell'ultima percezione ricevuta, ignorando tutta la storia pregressa. Sebbene possa apparire come una semplificazione estremamente brutale, anche in ambienti complessi se ne possono immaginare delle applicazioni: se si pensa ad un taxi a guida autonoma, quest'ultimo dovrebbe avviare la frenata non appena viene percepito che la macchina di fronte sta frenando, indipendente da cosa abbiamo percepito in precedenza.

Nella pratica possiamo immaginare che questo agente mantenga una serie di regole nella forma **if condizione then azione**, e di aver costruito un interprete generale per regole espresse in questa forma e poi creare insiemi di regole specifici per il particolare ambiente. Uno pseudocodice di riferimento per il generico agente di questo tipo lo troviamo a seguire:

Algoritmo 2 Agente che decide solo in base all'ultima percezione

```

1: function AGENTE-SEMPLICE-A-RIFLESSO(percezione)
2:   stato ← INTERPRETA-INGRESSO(percezione)
3:   regola ← REGOLA-CORRISPONDENTE(stato, regole)
4:   azione ← regola.AZIONE
5:   return azione

```

Qui sopra INTERPRETA-INGRESSO produce una descrizione astratta dello stato dell'ambiente a partire dall'ultima percezione ricevuta, mentre REGOLA-CORRISPONDENTE restituisce la prima regola tra tutte quelle presenti nell'oggetto *regole* che corrisponde alla descrizione dello stato. Questo tipo di agenti può funzionare solamente se la decisione corretta può essere presa in base alla sola ultima percezione ricevuta, ossia soltanto in ambienti completamente osservabili. Oltretutto questi semplici agenti sono soggetti ad un'ulteriore seria problematica: immaginiamo di implementare l'aspirapolvere automatizzato dell'esempio precedente in questo modo, allora ci sono due sole possibili percezioni che sono [*Sporco*], [*Pulito*]; ovviamente in risposta alla prima delle due l'azione sarà quella di aspirare, ma cosa si dovrebbe fare quando la stanza è pulita? Se si decide di andare a destra allora l'azione fallisce perpetuamente non appena ci si trova nella stanza B e si rileva che è pulita, mentre se avessimo deciso di andare a sinistra si avrebbe una situazione analoga ma per la stanza A. Per questo tipo di agenti in effetti è spesso inevitabile che si formino dei loop, tuttavia esiste una semplice soluzione: scegliere un'azione a caso; così facendo si risolve il problema e potenzialmente si ottiene un agente più performante di uno deterministico.

Agente a riflesso basato su modello

Il modo migliore per compensare la parziale osservabilità dell'ambiente di lavoro è mantenere uno stato interno costruito e mantenuto tramite le percezioni e simulare attraverso

esso gli aspetti del mondo che non sono osservati. Per poter aggiornare lo stato è richiesto che l'agente possieda due tipi di conoscenza: come evolve il mondo in sua assenza e come le azioni che esso esegue impattano sul mondo stesso, e questa conoscenza viene detta **modello**, da cui il nome dell'agente. Di seguito riportiamo uno pseudocodice di riferimento:

Algoritmo 3 Agente che decide in base all'ultima percezione e ad un modello

```

1: function AGENTE-A-RIFLESSO-BASATO-SU-MODELLO(percezione)
2:   stato ← AGGIORNA-STATO(stato, azione, percezione, modello)
3:   regola ← REGOLA-CORRISPONDENTE(stato, regole)
4:   azione ← regola.AZIONE
5:   return azione

```

Qui sopra *azione* rappresenta l'azione intrapresa più di recente; per il resto si può notare come lo pseudocodice somiglia fortemente a quello di un agente semplice a riflesso, differendo da esso formalmente soltanto nella prima riga. Ovviamente è difficile che la rappresentazione interna che l'agente si tiene del mondo esterno corrisponda perfettamente a ciò che è la realtà, più spesso si tratta della migliore supposizione a riguardo.

Agente guidato da un obiettivo

La conoscenza sullo stato attuale dell'ambiente talvolta non è sufficiente da sola per prendere la decisione sull'azione da svolgere, ad esempio un taxi a guida autonoma che si trova ad un incrocio a quattro vie potrebbe andare davanti, a destra o a sinistra ma senza conoscere la propria destinazione non può decidere l'azione corretta. Più in generale oltre alla descrizione dello stato dell'ambiente può esserci bisogno di un **obiettivo** (*goal*), ovvero di uno stato dell'ambiente considerato come vantaggioso. Per raggiungere un obiettivo un'agente potrebbe dover compiere una singola azione, altre volte invece dovrà prendere in considerazione lunghe sequenze di azioni. Il ramo dell'intelligenza artificiale che si occupa di trovare sequenze di azioni per raggiungere degli obiettivi si chiama **ricerca**.

2.2 Introduzione ai problemi di ricerca

Abbiamo appena accennato al fatto che un agente che deve massimizzare una misura di prestazioni può ottenere importanti semplificazioni se l'agente può assumere un obiettivo e cercare di soddisfarlo; questo aiuta a non considerare sequenze di azioni che portano verso stati dell'ambiente non vantaggiosi. Chiameremo "obiettivi" un insieme di stati del mondo, per la precisione quelli in cui l'obiettivo è soddisfatto; l'agente dovrà scegliere che azioni eseguire adesso ed in futuro per raggiungere uno di questi stati. Prima di questo, però, è necessario definire quali azioni e quali stati prendere in considerazione: sempre nel contesto della guida autonoma, se le azioni fossero del tipo "ruota il volante di α gradi" allora anche obiettivi semplici come uscire da un parcheggio potrebbero diventare impossibili per via dell'eccessivo numero di azioni richieste per riuscire. Il processo di decisione su quali stati e quali azioni considerare assegnato un obiettivo da raggiungere si chiama **formulazione del problema**.

In termini dell'ambiente, a meno di specificare diversamente faremo le seguenti assunzioni: l'ambiente di lavoro si considera come osservabile, di modo che lo stato dell'ambiente sia noto all'agente ad ogni istante, inoltre si assume che sia anche discreto, nel senso che il numero di azioni tra cui scegliere in ciascuno stato sia in quantità finita. Si supporrà inoltre che l'ambiente sia "noto", inteso con ciò che si conosca lo stato che si raggiunge in conseguenza all'applicazione di ciascuna azione, ed infine considereremo l'ambiente di lavoro come deterministico di modo che ciascuna azione abbia un solo possibile esito.

Il processo di individuare una sequenza di azioni che porta ad uno stato goal è chiamato ricerca, ed un algoritmo di ricerca prende in ingresso un problema e restituisce come soluzione una sequenza di azioni. Un agente guidato da un obiettivo, dopo aver formulato quest'ultimo, chiamerà una procedura per risolverlo e solo una volta ottenuta la soluzione potrà mettere in atto la sequenza di azioni suggerita; nel fare questo può ignorare la sequenza di percezioni che gli si presentano in ingresso. Vediamo di seguito uno pseudocodice per un generico agente guidato da un obiettivo che risolve un problema di ricerca:

Algoritmo 4 Agente che decide in base all'ultima percezione e ad un modello

```

1: function AGENTE-SEMPLICE-RISOLUTORE-DI-PROBLEMI(percezione)
2:   stato ← AGGIORNA-STATO(stato, percezione)
3:   if sequenza è vuota then
4:     obiettivo ← FORMULA-OBIETTIVO(stato)
5:     problema ← FORMULA-PROBLEMA(stato, obiettivo)
6:     sequenza ← RISOLVI-PROBLEMA(problema)
7:     if sequenza = fallimento then
8:       return un'azione non valida
9:   azione ← ESTRAI-PRIMO(sequenza)
10:  restanti ← RIMANENTI(sequenza)
11:  return azione

```

Qui sopra *sequenza* è una lista di azioni inizialmente vuota. Adesso ci occupiamo di come definire opportunamente un problema e le sue soluzioni. Per assegnare un problema dobbiamo definire queste componenti:

- (i) Uno **stato iniziale**, che è quello da cui l'agente inizia ad operare.
- (ii) Una descrizione delle azioni disponibili all'agente. Se siamo in uno stato s assumeremo che $AZIONI(s)$ restituisca l'insieme delle azioni che possono essere eseguite dall'agente quando l'ambiente si trova nello stato s . Le azioni in questa lista si dicono essere **applicabili** in s .
- (iii) Un **modello di transizione**, cioè una descrizione degli esiti delle varie azioni quando applicate nei vari stati a cui sono applicabili. Assumeremo che $RISULTATO(s, a)$ specifichi lo stato che si raggiunge se si applica l'azione a nello stato s .
- (iv) Un **goal test**, cioè una funzione che permette di stabilire se uno stato assegnato sia un'obiettivo o meno. Se è disponibile una lista esplicita degli stati obiettivo allora il goal test semplicemente verificherà se uno stato s fa parte della lista o meno, ma più in generale potrebbe valutare qualche proprietà astratta per determinare se lo stato è vantaggioso.

- (v) Una funzione di **costo dei cammini** (*path cost*) che ad ogni cammino associa un costo numerico. Questa funzione riflette la misura di prestazioni dell'agente, ad esempio nella guida autonoma i cammini corrispondono fisicamente a delle strade ed i loro costi potrebbero essere espressi in termini della loro lunghezza in *km*.

L'insieme dello stato iniziale, le azioni disponibili ed un modello di transizione definiscono implicitamente uno **spazio degli stati**, che è l'insieme di stati raggiungibili applicando diverse sequenze di azioni a partire dallo stato iniziale. Tale spazio forma un grafo orientato i cui nodi sono associati a degli stati e gli archi corrispondono ad azioni che hanno prodotto la transizione tra i due nodi collegati. Un cammino nel grafo è una sequenza di stati connessi da una sequenza di azioni. È questo il senso con cui si deve intendere “cammino” quando abbiamo introdotto il concetto di costo dei cammini.

Una soluzione di un problema di ricerca è una sequenza di azioni che porta dallo stato iniziale ad uno degli stati obiettivo, e la qualità di una soluzione è misurata tramite la funzione di costo dei cammini. Una **soluzione ottima** è una che ha il costo di cammino più basso possibile tra le varie possibili soluzioni.

28/09/2020

3.1 Risolvere problemi di ricerca

Abbiamo espresso alcune delle idee fondamentali sui problemi di ricerca, adesso vediamo come risolverli. Una soluzione è una sequenza di azioni e gli algoritmi di ricerca prendono in considerazione varie di queste sequenze. Le possibili sequenze di azioni che partono dallo stato iniziale s_0 definiscono un **albero di ricerca** che ha s_0 come radice, i cui nodi corrispondono a stati dello spazio degli stati del problema e i cui rami sono associati ad azioni. I vari algoritmi non fanno altro che costruire tale albero: assegnato un certo stato si considerano tutte le possibili sequenze di azioni applicabili in esso e in questo modo si genera un nuovo insieme di stati che nell'albero di ricerca corrisponderanno a figli del nodo corrente; questo processo si chiama **espansione**. L'insieme dei nodi candidati all'espansione, tipicamente corrispondenti a foglie nella porzione di albero di ricerca generato fino a questo momento, viene detto **frontiera**, ed il processo di espansione dei nodi ivi contenuti prosegue fintantoché non viene individuata una soluzione o non restano più nodi da espandere. Diverse implementazioni pratiche della frontiera corrispondono a modi diversi di scegliere il prossimo nodo da espandere e perciò a strategie di ricerca diverse.

Osserviamo a questo punto che nel caso in cui siano presenti delle azioni reversibili si possono presentare situazioni indesiderate, ad esempio se c'è un'azione a che porta da s_i ad s_j e nello stato s_j è applicabile un'azione a^{-1} che ci riporta in s_i si ottiene un **cammino ciclico** (*loopy path*) e questi possono far fallire algoritmi di ricerca in problemi che sarebbero altrimenti risolvibili. In ogni caso è facile intuire che uno di questi cammini non porta mai a soluzioni ottime: si può infatti percorrere lo stesso ciclo un numero arbitrario di volte e se i costi dei cammini sono non negativi non si ottiene mai un costo complessivo migliore rispetto a quello dello stesso cammino privato del ciclo. Anche in assenza di cicli esiste un'ulteriore problematica importante che è quella dei **cammini ridondanti**: assegnato un certo stato non è ovvio che lo si possa raggiungere in un solo modo, i.e. tramite una unica sequenza di azioni, e talvolta il proliferare di cammini ridondanti può provocare l'intrattabilità di un problema: immaginiamo per esempio una griglia rettangolare in cui ciascun nodo ha quattro vicini, allora ad una profondità d l'albero di ricerca avrà 4^d foglie, mentre si può dimostrare che l'insieme di stati diversi raggiungibili entro d azioni è soltanto $2d^2$, e.g. per $d = 20$ le foglie sono dell'ordine di 1×10^{12} mentre gli stati distinti raggiungibili sono solo 800.

La soluzione per queste due problematiche è di tenere traccia degli stati già esplorati, e a tal fine istituire una struttura dati che chiameremo **insieme degli esplorati** (*explored set*). A questo punto prima di aggiungere un nuovo nodo nella frontiera si verifica se esso è già presente nella frontiera stessa oppure in questo insieme degli esplorati ed in tal caso lo si scarta. Si ottiene così un'ossatura per gli algoritmi di ricerca conosciuta come GRAPH-SEARCH, di cui è riportato sotto un possibile pseudocodice. Nel frattempo osserviamo che in questa versione la frontiera gode di una proprietà interessante:

Proprietà 3.1.1 (Separazione della frontiera). Ogni cammino dallo stato iniziale ad uno

stato inesplorato deve necessariamente attraversare la frontiera.

Questo illustra anche come operano gli algoritmi di ricerca basati su questo schema: di volta in volta che vengono espansi nuovi nodi la frontiera avanza all'interno dello spazio degli stati finché quest'ultimo non viene sistematicamente esplorato tutto o viene trovata una soluzione.

Algoritmo 5 Algoritmo di ricerca che usa un'insieme degli esplorati

```

1: function GRAPH-SEARCH(problema)
2:   frontiera  $\leftarrow$  {problema.STATO-INIZIALE}
3:   esplorati  $\leftarrow$   $\emptyset$ 
4:   loop
5:     if frontiera è vuota then
6:       return fallimento
7:     else
8:       nodo  $\leftarrow$  frontiera.ESTRAI()
9:       if nodo contiene uno stato che è un obiettivo then
10:        return la sequenza di azioni corrispondente
11:       esplorati.AGGIUNGI(nodo)
12:       espandi nodo ed aggiungi i nodi generati alla frontiera, se nodo  $\notin$  frontiera
       e nodo  $\notin$  esplorati

```

3.2 Questioni implementative

Diversamente da quanto si potrebbe pensare l'albero di ricerca non è mai completamente memorizzato in memoria, ma al contrario lo è solamente in maniera parziale. La struttura dati responsabile del tenere traccia dell'albero in costruzione è il nodo, che sarà dotato almeno dei seguenti attributi:

- STATO: Lo stato nello spazio degli stati a cui il nodo è associato;
- GENITORE: Il nodo la cui espansione ha generato questo nodo;
- AZIONE: L'azione che ha portato dallo stato associato al genitore a questo stato;
- COSTO-CAMMINO: Il costo dell'intero cammino dal nodo associato allo stato iniziale a questo nodo.

Tutte queste informazioni possono essere facilmente compilate a partire da quelle del genitore, ad esempio il costo del cammino del nodo generato sarà pari alla somma tra quello del genitore ed il costo dell'azione che ha portato alla generazione di quel nodo. Inoltre precisiamo a questo punto che non c'è una identificazione tra nodo e stato: uno stesso stato potrebbe essere raggiungibile attraverso sequenze di azioni diverse e queste producono nodi associati entrambi a quello stato ma diversi tra di loro in tutti gli altri attributi (almeno a priori).

Un'altra questione implementativa rilevante riguarda l'implementazione della struttura utilizzata per la frontiera e le operazioni minime necessarie. Per quanto riguarda queste ultime prevediamo almeno le seguenti tre operazioni:

- **isEmpty:** Questa funzione accetta come unico argomento la frontiera, restituisce *true* se questa è vuota e *false* altrimenti.
- **Estrai:** Restituisce un elemento dalla frontiera in base a qualche criterio fissato e lo elimina dalla stessa.
- **Inserisci:** Questa funzione accetta due argomenti, un elemento da inserire e la frontiera, ed il risultato della sua invocazione è l'aggiunta dell'elemento specificato come parametro nella frontiera, anch'essa indicata come parametro.

Detto ciò normalmente la scelta sulla struttura dati da utilizzare ricade su una **coda**, di cui possiamo avere diverse implementazioni: in una **coda FIFO** viene estratto sempre l'elemento presente in coda da più tempo, in una **coda LIFO** l'elemento ad essere rimosso è sempre quello più recentemente inserito, infine in una **coda con priorità** viene estratto l'elemento che massimizza o minimizza una funzione di priorità assegnata.

L'ultima questione da affrontare riguarda l'implementazione dell'insieme degli esplorati. Le operazioni più frequenti che si effettuano con questa sono l'inserimento di nuovi stati e la ricerca, pertanto conviene che queste siano veloci; la scelta tipica è quella di implementarla come una tabella hash, che se costruita opportunamente permette di eseguire ambo le due operazioni in tempo pressoché costante indipendentemente dal numero di nodi presenti. Una questione a cui prestare attenzione è come viene deciso se due stati sono uguali: ci possono essere degli stati rappresentati da insiemi, come $\{A, B, C\}$, ma in cui l'ordine è irrilevante, ad esempio $\{B, A, C\}$ da considerare come uguale al precedente; in questo caso conviene far sì che lo stato sia rappresentato in una forma canonica, ad esempio come mappa di bit, di modo da ottenere una rappresentazione che renda agevole il test.

3.3 Confronto tra algoritmi

Nel corso di Algoritmi e strutture dati abbiamo visto vari algoritmi che lavorano su grafi ed effettuato l'analisi, che in quel caso molto spesso veniva formulata, per un grafo $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in termini di $|\mathcal{V}| + |\mathcal{E}|$; questo era possibile perché effettivamente i grafi su cui si lavorava erano presenti in memoria, ma adesso un'analisi del genere non è più adeguata perché come già sottolineato l'albero di ricerca è soltanto rappresentato nella memoria (e costruito gradualmente) ma mai del tutto memorizzato. Per effettuare l'analisi degli algoritmi di ricerca e quindi per confrontarli introduciamo quattro parametri:

- **Completezza:** Un algoritmo di ricerca si dice essere completo se garantisce di trovare una soluzione qualora questa esista;
- **Ottimalità:** Un algoritmo di ricerca si dice essere ottimo se tra tutte le soluzioni restituisce quella ottima, ovvero quella con costo di cammino più basso;
- **Complessità temporale:** È una stima di quanto l'algoritmo impiega a trovare una soluzione;
- **Complessità spaziale:** È una stima di quanta memoria viene richiesta dall'algoritmo per trovare una soluzione;

I primi due parametri in realtà sono delle proprietà desiderate, mentre le due complessità dovranno essere formulate in termini di tre quantità che ci accingiamo ad introdurre:

- **Fattore di ramificazione:** Indicato con b , dall'inglese *branching factor*, rappresenta il massimo numero di successori che un nodo dell'albero può avere;
- **Profondità:** Indicato con d , dall'inglese *depth*, rappresenta la profondità, contata come numero di azioni da eseguire a partire dallo stato iniziale, della soluzione più superficiale.
- **Massima lunghezza di un cammino:** Indicata con m , come suggerisce il nome è la massima lunghezza di un cammino nell'albero di ricerca.

3.4 Ricerca non informata

Quando si parla di ricerca non informata si fa riferimento ad una classe di algoritmi che cercano di risolvere un problema di ricerca visitando sistematicamente tutti gli stati possibili a partire dallo stato iniziale, senza avere nessuna conoscenza specifica sul problema che sono chiamati a risolvere.

3.4.1 Ricerca in ampiezza (BFS)

Questo algoritmo espande tutti i nodi che si trovano ad una data profondità prima di scendere al livello successivo. Il comportamento viene ottenuto semplicemente utilizzando l'ossatura GRAPH-SEARCH ed implementando la frontiera come una coda FIFO, o equivalentemente una coda di max-priorità ordinata secondo $f(n) = n.depth$, di modo che ad essere estratto sia sempre quello che è in coda da più tempo, ovvero quello più superficiale non ancora espanso. Come variazione rispetto allo schema generale il goal test viene eseguito prima di inserire il nodo nella frontiera consentendo di risparmiare in termini di complessità temporale. Prima di passare all'analisi osserviamo che quando uno

Algoritmo 6 Algoritmo di ricerca non informata che espande prima in ampiezza

```

1: function BREADTH-FIRST-SEARCH(problema)
2:   nodo  $\leftarrow$  Un nodo con lo stato iniziale e costo nullo
3:   if problema.GOAL-TEST( nodo.STATO ) then
4:     return SOLUZIONE( nodo )
5:   frontiera  $\leftarrow$  una coda FIFO contenente solo nodo
6:   esplorati  $\leftarrow \emptyset$ 
7:   loop
8:     if ISEMPY( frontiera ) then
9:       return fallimento
10:    nodo  $\leftarrow$  ESTRAI( frontiera )
11:    aggiungi nodo.STATO ad esplorati
12:    for all azione in problema.AZIONI(nodo.STATO) do
13:      figlio  $\leftarrow$  GENERA-FIGLIO( problema, nodo, azione )
14:      if figlio.STATO  $\notin$  frontiera  $\cup$  esplorati then
15:        if problema.GOAL-TEST( figlio.STATO ) then
16:          return SOLUZIONE( figlio )
17:        frontiera  $\leftarrow$  INSERISCI( figlio, frontiera )
```

stato viene esplorato lo si è raggiunto sempre attraverso il cammino più piccolo possibile:

se nell'espandere nuovi nodi viene generato uno stato che è già presente in frontiera o nell'insieme degli esplorati lo si scarta e questo ci garantisce la proprietà descritta. Di seguito riportiamo anche uno pseudocodice a cui fare riferimento.

Per quanto riguarda la completezza, se il fattore di ramificazione b è finito e c'è uno stato obiettivo ad una profondità $d < +\infty$ allora ne abbiamo la garanzia in quanto prima o poi vengono generati tutti gli stati raggiungibili, mentre per quanto riguarda l'ottimalità non possiamo garantirla nel caso generale: il primo stato obiettivo che incontriamo per quanto detto è garantito che sia il più superficiale di tutti ma non è ovvio che il più superficiale sia anche quello con il costo di cammino più basso; in ogni caso questo certamente accade se la funzione di costo dei cammini è una funzione non decrescente della profondità, ad esempio se tutti gli archi hanno costo pari ad una certa costante c . Con riferimento alla complessità, quella temporale è dominata dalla generazione delle foglie che a profondità d sono un $O(b^d)$; osserviamo che se avessimo eseguito il goal test solamente all'estrazione del nodo dalla frontiera piuttosto che prima di inserirlo si sarebbe scoperto che il nodo in questione è un goal soltanto una volta raggiunta la profondità $d+1$ e questo avrebbe portato ad una complessità di $O(b^{d+1})$, i.e. sarebbe più grande di un fattore b rispetto al meglio che si riesce ad ottenere. Infine, per quanto riguarda la complessità spaziale, ci sono due posti in cui vengono memorizzati dei nodi: la frontiera e l'insieme degli esplorati. La prima delle due contiene le foglie dell'albero, che a profondità d sono un $O(b^d)$, mentre il secondo contiene tutti i nodi precedenti, che sono

$$N_{\text{nodi}} = 1 + b + b^2 + \dots + b^{d-1} = O(b^{d-1})$$

Pertanto il costo maggiore in termini di memoria è dovuto alla memorizzazione delle foglie nella frontiera, per un totale anche in questo caso di $O(b^d)$. In entrambi i casi la complessità non è buona, ma il fatto che quella spaziale sia esponenziale in b è particolarmente critico perché la memoria è limitata ed in fretta si raggiungono requisiti di memoria che non possono essere soddisfatti dai computer attualmente in commercio. Ciò ci spinge a cercare algoritmi che almeno sotto questo aspetto facciano di meglio.

3.4.2 Ricerca in profondità (DFS)

Di volta in volta si estrae dalla frontiera il nodo più profondo ivi contenuto al momento; questo fa sì che l'algoritmo inizi sin da subito a scendere in profondità in un cammino piuttosto di aspettare di aver generato prima tutti i nodi allo stesso livello. Il modo in cui questa strategia viene realizzata è implementando la frontiera come una coda LIFO, di modo da estrarre sempre il nodo generato più recentemente, o equivalentemente con una coda di max-priorità ordinata secondo $f(n) = -n.\text{depth}$: così facendo quando si estrae un nodo dalla struttura esso sarà a profondità pari ad uno più quella del genitore, che a sua volta era il più profondo quando era stato estratto.

Le proprietà di questo algoritmo dipendono strettamente dal tipo di schema adottato tra GRAPH-SEARCH e TREE-SEARCH¹: nel primo caso si riescono ad evitare cicli e cammini ridondanti e pertanto prima o poi si genera l'intero spazio degli stati se questo è finito, quindi si ha garanzia di completezza. D'altro canto se si usa la versione senza lista degli esplorati l'algoritmo può rimanere intrappolato indefinitamente in un ciclo e quindi non si ha completezza; questa versione può essere modificata per fare in modo di controllare se uno stato generato era già presente nel cammino dalla radice al nodo corrente e

¹Simile a GRAPH-SEARCH ma senza insieme degli esplorati

quindi evitare la formazione di cicli, ma resta il problema dei cammini ridondanti. Se lo spazio degli stati è infinito non si può avere completezza indipendentemente dalla scelta sullo schema, infatti può succedere che l'algoritmo intraprenda un cammino di lunghezza infinita che non contenga stati obiettivo e quindi non se ne trova mai uno.

Per quanto riguarda l'ottimalità è facile convincersi che in nessuna delle due implementazioni goda di questa proprietà, infatti possiamo immaginare una situazione in cui i costi degli archi siano costanti e quindi il costo di una soluzione sia pari alla sua profondità: in questo caso la soluzione ottima viene trovata soltanto se l'obiettivo in questione si trova nel primo cammino espanso contenente uno stato obiettivo.

In termini di complessità temporale, questa è dominata dalla generazione dei vari nodi, e DEPTH-FIRST-SEARCH può arrivare a generare l'intero spazio degli stati che ha dimensione $O(b^m)$; si noti che potenzialmente $m \gg d$ e quindi almeno fino a questo punto non si sono incontrati vantaggi evidenti nell'utilizzare questa implementazione piuttosto di una ricerca in ampiezza. Il vero vantaggio risiede nella complessità spaziale dell'algoritmo: nel percorrere un cammino con la ricerca in profondità dobbiamo tenere traccia di tutti i nodi nel cammino insieme ai suoi successori, che sono al massimo b ; poiché un cammino può arrivare ad avere lunghezza massima m la memoria richiesta è un $O(bm)$, che è molto migliore rispetto all' $O(b^d)$ richiesto dalla BREADTH-FIRST-SEARCH. L'occupazione può essere ulteriormente migliorata se invece di memorizzare tutti i successori di ogni nodo del cammino se ne memorizza uno solo e si fa in modo che ciascun nodo tenga traccia internamente di quale successore generare come prossimo, portando ad un $O(m)$.

Ricerca a profondità limitata (DLS)

Algoritmo 7 Ricerca prima in profondità con limitazione sulla profondità massima

```

1: function DEPTH-LIMITED-SEARCH(problema, limite)
2:   return DLS(CREA-NODO(problema.STATO-INIZIALE), problema, limite)
3: function DLS(nodo, problema, limite)
4:   if problema.GOAL-TEST(nodo) then
5:     return SOLUZIONE(nodo)
6:   else if limite = 0 then
7:     return interrotto
8:   else
9:     avvenuta_interruzione  $\leftarrow$  false
10:    for all azione in problema.AZIONI( nodo.STATO ) do
11:      figlio  $\leftarrow$  GENERA-FIGLIO( problema, nodo, azione )
12:      risultato  $\leftarrow$  DLS( figlio, problema, limite-1 )
13:      if risultato = interrotto then
14:        avvenuta_interruzione  $\leftarrow$  true
15:      else if risultato  $\neq$  fallimento then
16:        return risultato
17:    if avvenuta_interruzione = true then
18:      return interrotto
19:    else
20:      return fallimento

```

Si tratta di un algoritmo che nasce per porre rimedio alla incompletezza di DLS in spazi di stati infiniti: l'idea è quella di passare come ulteriore parametro un certo ℓ che rappresenta la massima profondità che vogliamo consentire all'algoritmo di raggiungere lungo un cammino. Benché effettivamente si riesca così a porre rimedio al problema che ci proponevamo di risolvere, l'applicare questa idea scaturisce una nuova potenziale fonte di incompletezza: può infatti succedere che, non conoscendo il valore di d , si scelga accidentalmente $\ell < d$ e questo fa sì che nessuno stato obiettivo venga mai raggiunto. Anche se scegliessimo $\ell \geq d$ non avremmo comunque garanzie di ottimalità, anche con questa miglioria. In quanto all'analisi, l'algoritmo ha una complessità temporale di $O(b^\ell)$ e spaziale di $O(b\ell)$ essendo una semplice variante di DLS in cui si limita la lunghezza massima di un cammino ad $m = \ell$. Sopra è riportato uno pseudocodice per l'algoritmo.

Approfondimenti successivi (iterative deepening)

L'algoritmo di ricerca prima in profondità con approfondimenti successivi una strategia utilizzata in combinazione con la versione priva di insieme degli esplorati di DEPTH-FIRST-SEARCH per individuare il miglior limite ℓ con cui lanciare la ricerca a profondità limitata. L'idea è quella di chiamare la DLS con un limite che, partendo da 0, viene incrementato ogni volta di 1 finché non viene trovato uno stato obiettivo, ossia finché il valore del limite non raggiunge proprio d , la profondità dello stato obiettivo più superficiale.

Questa strategia consente di simulare il comportamento di una ricerca prima in ampiezza (BFS) mantenendo i benefici in termini di requisiti di memoria di una ricerca prima in profondità, infatti la memoria richiesta è un $O(bd)$ mentre il tempo è nuovamente un $O(b^d)$. Come nella BFS quando il fattore di ramificazione b è finito ed esiste un nodo obiettivo ad una profondità finita si ha garanzia di completezza, e si ha anche ottimalità se la funzione di costo è non decrescente rispetto alla profondità. Vale la pena approfondire la questione della complessità temporale: uno potrebbe pensare che il fatto di generare più volte da capo lo stesso albero possa far aumentare il costo, tuttavia se il fattore di ramificazione è pressoché costante lungo l'albero la maggior parte dei nodi si trova all'ultimo livello ed è questo che ne fissa il costo. Più in particolare il primo livello viene generato ogni volta che si riparte per un totale di d volte, il secondo $d - 1$ volte, il terzo $d - 2$ e via dicendo, pertanto

$$N_{\text{nodi generati}} = db + (d - 1)b^2 + (d - 2)b^3 + \dots, 1b^d = O(b^d)$$

Che è quanto abbiamo suggerito in precedenza.

Algoritmo 8 Variante DLS con aumento progressivo del limite sulla profondità massima

```

1: function ITERATIVE-DEEPENING(problema)
2:   for profondità = 0, 1, ..., +∞ do
3:     risultato ← DEPTH-LIMITED-SEARCH( problema, limite )
4:     if risultato ≠ interrotto then
5:       return risultato

```

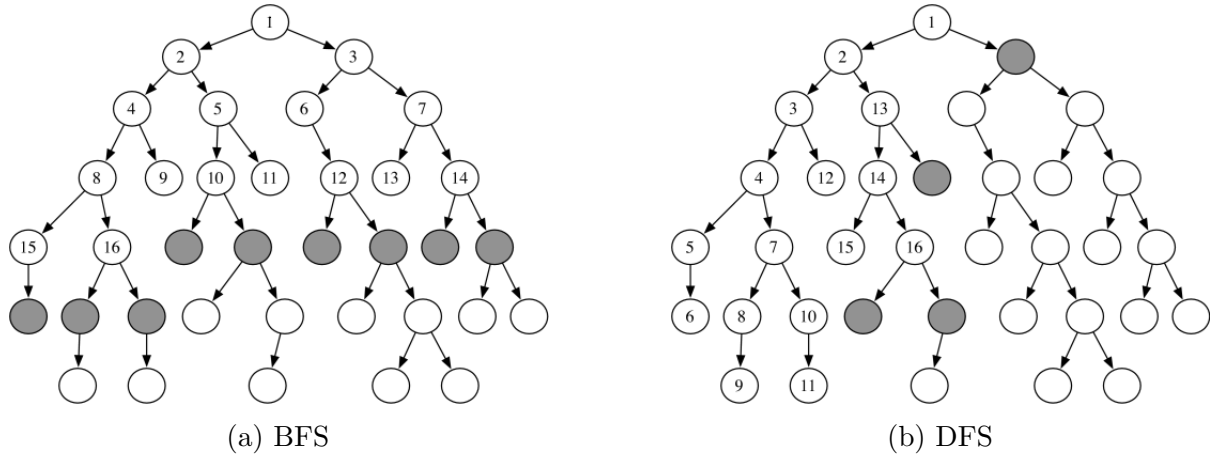


Figura 3.1: Un'illustrazione del funzionamento di BFS e DFS; l'etichetta di ciascun nodo corrisponde al numero d'ordine con cui il nodo viene espanso

3.4.3 Uniform Cost Search

Abbiamo già visto un algoritmo che è ottimale quando i costi dei cammini sono tutti costanti, ed adesso vediamo un algoritmo che è in grado di garantire l'ottimalità in generale, che si chiama Uniform Cost. L'obiettivo viene raggiunto implementando la frontiera come una coda con priorità, e di volta in volta il nodo estratto per l'espansione tra quelli ivi contenuti è quello che ha il costo minore del cammino dalla radice ed esso. Dato un nodo n indichiamo tale costo con $g(n)$, che rappresenta anche la funzione di priorità usata dalla coda.

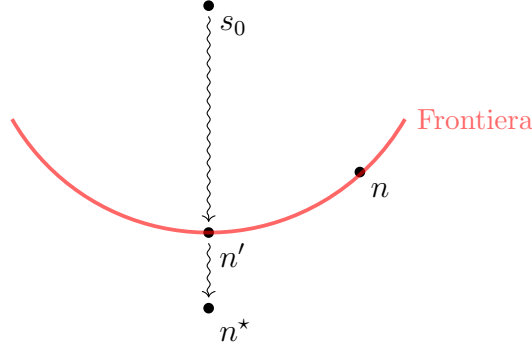
Rispetto a BFS sono apportati due cambiamenti: il goal test viene riportato al momento dell'estrazione dalla frontiera come avviene nella GRAPH-SEARCH piuttosto che prima dell'inserimento come nella ricerca prima in ampiezza, questo perché nel momento in cui uno stato obiettivo viene generato, considerato che possono esserci più cammini che portano a lui, potremmo trovarci in un cammino subottimo. Allo stesso fine viene aggiunto un ulteriore controllo: ogni volta che si genera un nuovo nodo si guarda se esso è presente in frontiera, ed in tal caso se si proviene da un cammino migliore rispetto al vecchio, il nodo in frontiera viene sostituito.

Poiché Uniform Cost Search non tiene in considerazione la lunghezza dei cammini ma solamente i costi, se c'è un ciclo in cui è presente una sequenza infinita di azioni con costo nullo l'algoritmo non è completo. Si tratta comunque di una situazione piuttosto rara, e per esprimerla in termini formali dovremo richiedere che esista una costante $\epsilon > 0$ tale per cui i costi di ogni azione sono maggiori di questa; se questo accade abbiamo completezza. Per quanto riguarda l'analisi della complessità, poiché l'algoritmo è guidato dai costi dei cammini piuttosto che dalla loro profondità è difficile effettuarla in termini dei parametri canonici introdotti in precedenza e conviene invece effettuarla in termini del costo della soluzione ottima, che indichiamo con C^* . Poiché ogni azione ha un costo non più piccolo di ϵ possiamo dare un limite superiore alla lunghezza del cammino dallo stato iniziale al nodo che rappresenta la soluzione ottima, che è $\lceil C^*/\epsilon \rceil$. Se i costi fossero costanti allora tale valore coinciderebbe con d e sarebbe esatto, non più una limitazione superiore. Il costo di Uniform Cost nel caso peggiore, sia in termini di tempo che di memoria, è un $O(b^{\lceil C^*/\epsilon \rceil})$, e se i costi dei cammini sono costanti si ritorna ad un $O(b^{d+1})$, con il +1 che deriva dal fatto che il goal test è stato spostato al momento dell'estrazione

del nodo dalla frontiera.

Proposizione 3.4.1 (Ottimalità di Uniform Cost). La ricerca con UNIFORM-COST è ottimale.

Dimostrazione.



Con riferimento alla figura, si prende in considerazione il seguente scenario: n^* è lo stato obiettivo ottimo mentre n è uno subottimo presente nella frontiera, ossia

$$g(n^*) < g(n) \quad (3.1)$$

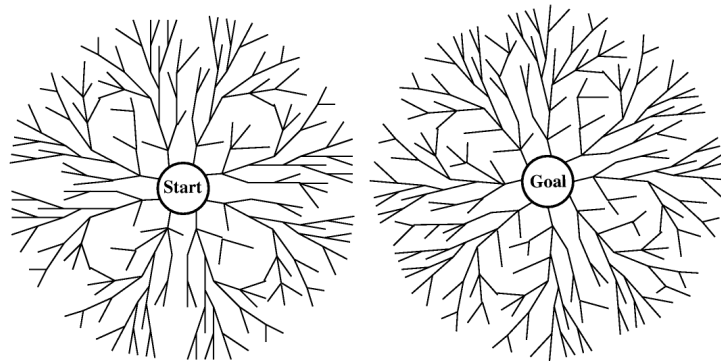
In questo contesto vogliamo dimostrare che è assurdo estrarre n dalla frontiera.

Notiamo che n^* non è ancora stato esplorato, altrimenti l'algoritmo sarebbe terminato. Allora, per la proprietà di separazione della frontiera il cammino dallo stato iniziale s_0 ad n^* deve intersecare quest'ultima in un certo nodo che chiamiamo n' . Ovviamente il costo di n^* può essere espresso in termini di quello di n' come $g(n^*) = g(n') + C$, se C è il costo del cammino da n' ad n^* . Poiché i costi sono quantità positive segue che

$$g(n') < g(n^*) \quad (3.2)$$

Mettendo insieme le relazioni 3.1 e 3.2 segue che $g(n) > g(n')$ e quindi non si può aver estratto dalla frontiera prima n rispetto ad n' in quanto è il secondo ad avere il costo minore, pertanto questo scenario, l'unico che avrebbe portato alla non ottimalità dell'algoritmo, è assurdo. ■

3.4.4 Ricerca bidirezionale



Se lo stato obiettivo è noto e le azioni disponibili sono tutte reversibili allora si può pensare di eseguire simultaneamente due ricerche, una che parte dallo stato iniziale ed

una che parte dallo stato obiettivo, con la speranza che le due ricerche si intersechino a metà cammino portando così a costi dell'ordine di $2b^{d/2}$, che è un risparmio considerevole rispetto al b^d di BFS ad esempio. In questo caso il controllo di terminazione non consiste più in un goal test ma nella verifica se le due frontiere si intersecano, nel qual caso si è trovata una soluzione. Se la ricerca in entrambe le direzioni è effettuata usando BFS e il fattore di ramificazione è finito allora l'algoritmo di ricerca bidirezionale è completo, ma in generale non è garantita l'ottimalità. Per avere quest'ultima è necessario fare delle assunzioni in più, ad esempio l'abbiamo se i costi delle azioni sono tutti uguali.

Per quanto già discusso in precedenza, la complessità temporale e spaziale è un $O(b^{d/2})$, ma l'occupazione di memoria può essere approssimativamente dimezzata se una delle due ricerche viene effettuata con ITERATIVE-DEEPENING (algoritmo 8).

Per poter impiegare questa tecnica dobbiamo essere in grado di cercare all'indietro. Se n è un nodo dell'albero chiamiamo **predecessori** di n tutti i nodi che hanno n tra i loro figli, e qualora le azioni fossero reversibili tali predecessori coinciderebbero coi successori del nodo, quindi almeno in questo caso è facile. Inoltre è difficile applicare la tecnica quando piuttosto di avere uno stato obiettivo ben definito conosciamo soltanto una descrizione astratta di una sua caratteristica desiderata, per esempio nel problema delle 8 regine non si conosce esplicitamente una soluzione, si sa solo che nessuna regina deve attaccarne un'altra.

3.4.5 Confronto tra strategie di ricerca non informate

Criterio	BFS	DFS	DLS	ID-DLS	Uniform Cost	Bidirectional Search
Completo	Sì ¹	No	No	Sì ¹	Sì ¹²	Sì ¹⁴
Ottimo	Sì ³	No	No	Sì ³	Sì	Sì ³⁴
Tempo	$O(b^d)$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^{d/2})$
Memoria	$O(b^d)$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^{d/2})$

¹ Completo se il fattore di ramificazione è finito.

² Completo se i costi delle azioni sono tutti $\geq \epsilon$.

³ Ottimo se i costi delle azioni sono tutti uguali.

⁴ Se entrambe le direzioni usano BFS.

1/10/2020

4.1 Algoritmi di ricerca informata

Questa classe di algoritmi, in contrapposizione a quelli di ricerca “alla cieca” (o non informata), utilizzano conoscenze specifiche sul problema che vanno al di là della sua sola formulazione. La strategia dominante è nota come **best-first search**, che consiste in una generalizzazione di quanto fatto da UNIFORM-COST-SEARCH: anche in questo caso la frontiera viene implementata come una coda con priorità, però la funzione con cui vengono ordinati i nodi qui dentro è una generica $f(n)$ detta **funzione di valutazione**. Spesso questa impiega una cosiddetta **euristica**:

Definizione 4.1 (Euristica). Chiamiamo **euristica** una funzione $h(n)$ che per ogni nodo n stima la sua distanza da uno stato obiettivo. Questa è soggetta al vincolo di non negatività e al dover assumere valore nullo in corrispondenza di nodi n il cui stato associato sia un obiettivo.

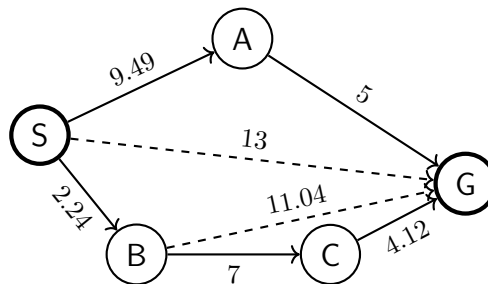
Tale $h(n)$ è dove viene inserita la conoscenza aggiuntiva sul problema, e spesso è una componente della funzione di valutazione. A seconda di come viene scelta $f(n)$ si ottengono diversi algoritmi best-first. Notiamo inoltre che nonostante l'euristica prenda come argomento un nodo n , il suo valore dipende unicamente dallo stato associato.

4.1.1 Greedy best-first search

Se la funzione di valutazione coincide con l'euristica, i.e. $f(n) = h(n)$, si ottiene un algoritmo conosciuto come **greedy best-first**: di volta in volta dalla frontiera viene estratto il nodo che sembra più promettente, cioè la cui distanza stimata da una soluzione è più piccola.

Nonostante l'idea possa sembrare interessante, l'algoritmo corrispondente è sia incompleto che subottimo, infatti se nel problema sono presenti azioni reversibili si possono formare dei cicli infiniti che impediscono all'algoritmo anche solo di terminare. Anche in assenza di cicli tuttavia non possiamo dare garanzie di ottimalità, sebbene si riesca a recuperare la completezza. Ce ne convinciamo con un controesempio.

Esempio 4.1: Non ottimalità di Greedy best-first

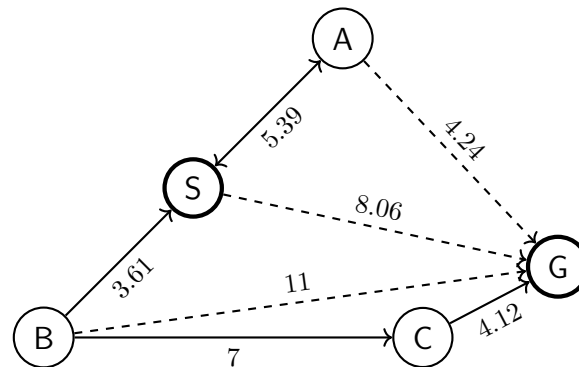


Si considera il grafo in figura con i suoi costi. Si assume come stato iniziale quello associato al nodo S e come stato obiettivo quello associato al nodo G. Gli archi tratteggiati

riportano come etichetta il valore dell'euristica in corrispondenza del nodo in cui arriva la freccia.

L'algoritmo dapprima espande il nodo S generando $\{B, A\}$, in cui l'euristica vale rispettivamente 11.04 e 5, pertanto dalla frontiera si estrae A . Espandendo quest'ultimo viene generato G che contiene uno stato obiettivo e quindi si termina. Il costo del cammino $S - A - G$ è di 14.49, ma non è il cammino ottimo, infatti $S - B - C - G$ ha un costo complessivo di 13.36 che è più basso. ■

Esempio 4.2: Incompletezza di Greedy best-first



Come nell'esempio di prima si assume S come stato iniziale e G come stato obiettivo.

Espandendo S vengono generati i nodi $\{A, B\}$ con rispettivi valori di priorità 5.39 ed 11. Per questa ragione il candidato all'espansione scelto da Greedy best-first è A ; la sua espansione genera il nodo G e quindi a questo punto la frontiera sarà composta da $\{S, B\}$ che hanno valori di priorità rispettivamente 8.06 ed 11, pertanto il nodo selezionato dall'algoritmo è S . In questa situazione si genera un ciclo infinito $S - A - G$ a causa della presenza di un'azione reversibile e l'algoritmo non termina, causandone l'incompletezza. ■

In termini del costo si può far vedere che se si usa la versione senza insieme degli esplorati sia il costo temporale che la richiesta di memoria sono un $O(b^m)$, ma si possono ottenere sostanziali miglioramenti se si progetta adeguatamente l'euristica; di quanto si può riuscire a migliorare dipende dalla stessa euristica e dal problema in questione.

4.1.2 Ricerca con A^*

L'algoritmo più famoso tra quelli della classe best-first è certamente A^* : esso utilizza come funzione di valutazione $f(n) = g(n) + h(n)$, ovvero una stima del costo della soluzione migliore che passa per n . In quest'ottica è ragionevole estrarre dalla frontiera il nodo che minimizza questa quantità, infatti sarà quello che tra tutti ha un costo stimato della soluzione migliore possibile; per il resto il comportamento è analogo a quello di UNIFORM-COST. In realtà si fa vedere che l'algoritmo è ben più che semplicemente ragionevole: sotto alcune condizioni dimostreremo che è anche ottimale.

Alcune proprietà delle euristiche

In questa sottosezione vediamo alcune proprietà di cui possono godere o meno delle euristiche, utili al fine di garantire completezza per A^* .

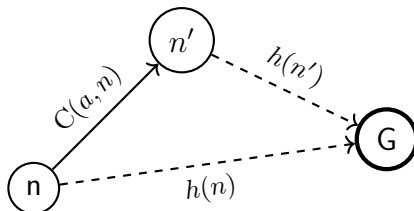
Definizione 4.2 (Euristica Ammissibile). Una euristica $h(n)$ si dice essere **ammissibile** se il vero costo ottimo da n .STATO allo stato obiettivo non è mai sovrastimato.

Per esempio se immaginiamo che il problema da risolvere sia di passare da un paese dato ad un altro impostato come destinazione, potremmo scegliere come euristica $h(n)$ la distanza in linea d'aria dal paese associato al nodo n a quello di destinazione; poiché quella distanza è la più piccola possibile l'euristica corrispondente è certamente ammissibile. Più in generale se intendiamo usare $h(n)$ come in A^* sovrastimare non è un bene, infatti un'euristica troppo grande fa sì che $f(n) = g(n) + h(n)$ sia troppo influenzata da quest'ultima e quindi intuitivamente ci si sposta verso una Greedy best-first.

Adesso invece introduciamo una seconda proprietà e vediamo com'è legata alla precedente:

Definizione 4.3 (Euristica Consistente). Un'euristica $h(n)$ si dice essere **consistente** se per ogni nodo n , per ogni suo successore n' ed ogni azione a applicabile in n vale la seguente disuguaglianza:

$$h(n) \leq \text{COSTO}(a, n) + h(n')$$



A parole, la distanza stimata dello stato n .STATO dall'obiettivo dev'essere non più grande della somma del costo per raggiungere n' partendo da n e la distanza stimata da n' da G ; all'atto pratico è una sorta di disuguaglianza triangolare. Il legame tra le due proprietà è espresso dal seguente risultato che non dimostriamo:

Teorema 4.1.1 (Legame ammissibilità–consistenza). Ogni euristica consistente è anche ammissibile, mentre il viceversa non vale in generale sebbene i controesempi siano rari.

Lemma 4.1.2 (Monotonicità di euristiche consistenti). Ogni euristica $h(n)$ consistente è anche **monotona**, ossia assegnato un albero ed un cammino su di esso il valore della funzione di valutazione $f(n)$ sul cammino è non decrescente: più si scende lungo esso più aumenta (o resta uguale).

Dimostrazione. Se n' è un successore di n per dimostrare dobbiamo far vedere che $f(n') \geq f(n)$ assumendo che l'euristica adottata sia consistente. Poiché n' è un successore di n ci dovrà essere un arco nell'albero $n \rightarrow n'$ con un costo associato $C(a, n)$, da cui segue che

$$g(n') = g(n) + C(a, n) \tag{4.1}$$

Con questo risultato siamo pronti a concludere, infatti:

$$\begin{aligned} f(n') &\stackrel{\text{def}}{=} g(n') + h(n') \stackrel{(a)}{=} g(n) + C(a, n) + h(n') \\ &\stackrel{(b)}{\geq} g(n) + h(n) \stackrel{\text{def}}{=} f(n) \end{aligned}$$

L'uguaglianza (a) segue dall'equazione 4.1 mentre la disuguaglianza (b) dalla definizione di euristica consistente. Prendendo il primo e l'ultimo membro della catena si ottiene che in effetti

$$f(n') \geq f(n) \quad \forall n, n' \text{ t.c. } n' \text{ è successore di } n$$

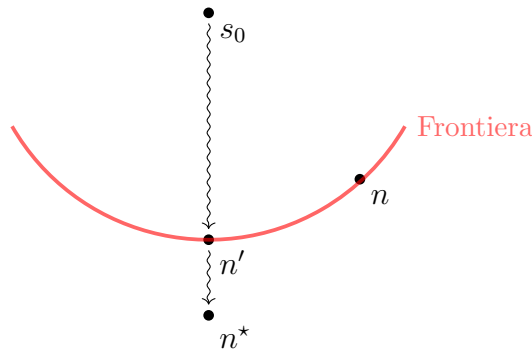
■

Ottimalità di A^*

Teorema 4.1.3 (A^* con euristica ammissibile). La versione senza insieme degli esplorati di A^* con un'euristica ammissibile è ottima

Teorema 4.1.4 (A^* con euristica consistente). La versione GRAPH-SEARCH di A^* con un'euristica consistente è ottima.

Dimostrazione. Poiché A^* ha un comportamento simile ad UNIFORM-COST anche in questo caso l'unica situazione che può indurre la non ottimalità dell'algoritmo è quella in cui in frontiera è presente un nodo associato ad uno stato subottimo e taluno viene scelto per l'estrazione:



Chiamiamo s_0 lo stato iniziale, n^* l'obiettivo ottimo ed n quello subottimo presente in frontiera. Per non ottimalità si dovrà avere $g(n) > g(n^*)$ ed inoltre n^* non è ancora stato esplorato, altrimenti l'algoritmo sarebbe terminato. Per la proprietà di separazione della frontiera della GRAPH-SEARCH il cammino $s_0 \rightsquigarrow n^*$ dovrà intersecare quest'ultima e chiamiamo n' il nodo che realizza tale intersezione. Inoltre si sta impiegando un'euristica consistente e quindi per il lemma 4.1.2 si avrà che

$$f(n') \leq f(n^*) \tag{4.2}$$

Mettendo insieme tutto questo si ricava la seguente catena di relazioni:

$$\begin{aligned} f(n') &\leq f(n^*) \stackrel{def}{=} g(n^*) + h(n^*) \\ &\stackrel{(a)}{=} g(n^*) < g(n) \\ &\stackrel{(b)}{=} g(n) + h(n) = f(n) \end{aligned}$$

La relazione (a) deve valere perché si sta usando un'euristica consistente e quindi si deve stimare consistentemente la distanza da n^* ad uno stato obiettivo (sé stesso), per cui $h(n^*) = 0$ necessariamente; la disuguaglianza seguente deriva dal fatto che n^* è ottimo

ed n no, mentre la (b) deriva da una considerazione analoga a quella della relazione (a). Se si prende il primo e l'ultimo membro della catena otteniamo che

$$f(n') \leq f(n)$$

Pertanto lo scenario proposto è assurdo: n' ha una priorità maggiore rispetto ad n e quindi il secondo non potrà essere estratto dalla frontiera prima del primo. ■

4.2 Progettare buone euristiche

Esempio 4.3: Euristiche per il puzzle dell'8

Stato obiettivo

	1	2
3	4	5
6	7	8

Generico n.STATO

2		3
4	1	7
6	5	8

Il problema giocattolo consiste nello spostare ciascun tassello numerato da una data configurazione assegnata in maniera tale da riordinarli come illustrato nella figura di destra. Ogni tassello può essere spostato nelle quattro direzioni ma non può essere né fisicamente staccato e riattaccato da un'altra parte né spostato in diagonale.

Un'euristica sensata per questo problema potrebbe essere la cosiddetta **misplaced tiles** (*tasselli fuori posto*), che assegna $h_1(n)$ come il numero di mattoncini che non sono nella posizione corretta, per esempio nello stato riportato nella figura di destra $h_1(n) = 6$. Supponiamo di riuscire a generare un'ulteriore euristica per questo problema che in corrispondenza dello stesso stato prenda valore $h_2(n) = 13$, allora quale delle due si preferisce? Intuitivamente la seconda, perché un'euristica con un valore più elevato produce un'esecuzione che si avvicina di più alla greedy best-first, che va dritta all'obiettivo e normalmente richiede un numero piccolo di passi. ■

5/10/2020

5.1 Progettare buone euristiche per A^*

5.1.1 Dominanza tra euristiche

Lemma 5.1.1. Sia assegnato un problema di ricerca con costo della soluzione ottima pari a C^* , allora l'algoritmo A^* :

- (a) Espande tutti i nodi n per cui $f(n) < C^*$
- (b) Può espandere alcuni nodi per cui $f(n) = C^*$
- (c) Non espande alcun nodo per cui $f(n) > C^*$ ■

Questo lemma può guidarci quando siamo chiamati ad operare una scelta tra due euristiche o più: se sono assegnate $h_1(n), h_2(n)$ tali per cui per ciascun nodo n $h_2(n) \geq h_1(n)$ allora tutti i nodi espansi da A^* usando $h_2(n)$ vengono espansi anche con $h_1(n)$ mentre il viceversa non vale, ossia con $h_2(n)$ vengono espansi meno nodi. In questo caso si dice che la seconda euristica è **più informata** della prima, o talvolta anche che **domina** la prima. Per vederlo formalmente:

- (i) $E_1 = \{n: h_1(n) < C^* - g(n)\} \cup X_1 := S_1 \cup X_1$
- (ii) $E_2 = \{n: h_2(n) < C^* - g(n)\} \cup X_2 := S_2 \cup X_2$

Qui sopra X_1 ed X_2 sono sottoinsiemi di nodi che hanno un costo pari a quello dell'obiettivo ottimo ma non sono a loro volta degli obiettivi, con il pedice che è riferito all'euristica. Assegnato un problema possiamo assumere che C^* sia fissato, e così anche $g(n)$ se si stabilisce un particolare nodo n , e quindi, a meno che il problema non sia "sfortunato" nel senso che ci siano molti nodi con costo vicino a quello dell'obiettivo ottimo, la maggior parte dei nodi si colloca in S_1 ed S_2 rispettivamente. È facile convincersi che $S_2 \subset S_1$ se $h_2(n)$ è più informata di $h_1(n)$, infatti più è grande il valore dell'euristica e minore sarà il numero di nodi per cui questa è più piccola di $C^* - g(n)$ ($g(n)$ è uguale sia in S_1 che S_2). Segue quindi che euristiche più informate sono più vantaggiose perché fanno espandere meno nodi.

Se sono assegnate due euristiche $h_1(n)$ ed $h_2(n)$ per le quali non si riesce a dimostrare che proprio per ogni nodo n valga $h_2(n) \geq h_1(n)$ si può definire una nuova euristica $h_3(n)$ nel modo seguente:

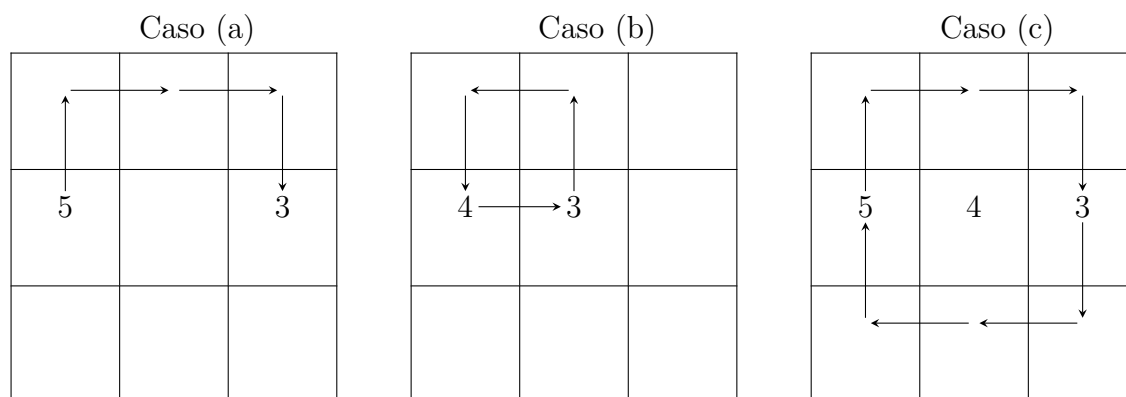
$$h_3(n) = \text{MAX} \{h_1(n), h_2(n)\}$$

Si dimostra che quella ottenuta è nuovamente un'euristica ammissibile se lo sono quelle da cui siamo partiti, ed inoltre $h_3(n)$ siffatta è più informata sia di $h_1(n)$ che di $h_2(n)$. La tecnica può essere estesa ad un numero arbitrario, purché finito, di euristiche consistenti.

Esempio 5.1: Euristiche più informate di Manhattan

Sempre riferendoci al problema del puzzle dell'8 chiamiamo **Manhattan** l'euristica $h_2(n)$ il cui valore è dato dalla somma delle distanze di ciascun tassello dalla sua posizione corretta. In realtà vedremo adesso che si riescono ad ottenere euristiche

migliori rispetto a questa.



Nel caso (a) il quadro raffigurato darebbe un contributo di 4 al calcolo di $h_2(n)$, ma per scambiare di posto i tasselli 3 e 5 occorre far passare il secondo da sopra o da sotto portando ad un costo effettivo di almeno 6, i.e. due in più di quanto stimato con Manhattan; per la stessa ragione nel caso (b) usando $h_2(n)$ i tasselli 4 e 3 porterebbero un contributo di +2 mentre si nota dalla figura che le mosse necessarie sono quantomeno 4, i.e. due in più rispetto a quelle stimate da Manhattan. Infine nel caso (c) il contributo dei tasselli 5, 4, 3 all'euristica h_2 sarebbe di 4 mentre il numero effettivo di mosse è almeno 8, cioè quattro in più a quanto stimato. ■

5.1.2 Pattern databases: sottoproblemi ed euristiche

Riprendendo l'ultimo esempio visto, dovrebbe essere chiaro che l'euristica migliore di tutte è quella che per ogni configurazione stima la sua vera distanza da uno stato obiettivo, e quindi uno potrebbe pensare di utilizzare proprio quest'ultima: tutte le volte che si deve calcolare $h(n)$ si lancia un problema di ricerca e si conta la lunghezza del cammino trovato assumendo n come stato iniziale. Il problema di questo approccio è che così facendo il calcolo dell'euristica inizia ad assorbire quasi tutto il tempo di esecuzione dell'algoritmo che così aumenta senza controllo: idealmente si vorrebbe poter calcolare $h(n)$ in un tempo costante, $O(1)$.

L'idea non è comunque completamente da buttare: si potrebbe ad esempio pensare di memorizzare il valore dell'euristica per tutte le possibili configurazioni del problema, precalcolato come detto sopra, e all'occorrenza quando si va a risolvere un'istanza del problema riuscire a recuperare la vera distanza dallo stato obiettivo in tempo $O(1)$ come desiderato. Ovviamente il precalcolo di queste distanze richiede potenzialmente molto tempo e quindi fare così ha senso solo quando si vogliono risolvere molte istanze di uno stesso problema, ad esempio qualche centinaio di puzzle dell'8. Anche in questo caso però c'è da precisare che il numero di configurazioni diverse può essere estremamente grande, e.g. è fattoriale nel caso del puzzle dell'8, e quindi può tranquillamente succedere che la tabella così ottenuta non ci stia in memoria.

Per ovviare a questo inconveniente si può pensare di scomporre il problema di ricerca in sottoproblemi più piccoli ed applicare l'idea a questi piuttosto che al problema di partenza. Per fissare le idee si consideri il puzzle del 15 con il seguente schema:

Stato obiettivo				Tipico sottoproblema			
	1	2	3		✖	✖	
4	5	6	7	✖	✖	✖	
8	9	10	11	✖	✖	✖	
12	13	14	15				

Nelle figure la cella in giallo rappresenta il tassello vuoto mentre le celle in azzurro sono una selezione di un sottoinsieme di possibili tasselli che si vogliono prendere in considerazione. Le celle contenenti un carattere ✖ invece vengono ignorate, i.e. non viene preso in considerazione il valore che assumono. Se si usano dei sottoproblemi fatti così per calcolare le euristiche è chiaro che si ottiene un valore che non può sottostimare la vera distanza dall'obiettivo: le mosse per posizionare correttamente i numeri scelti vanno eseguite comunque, ed eventualmente ne saranno richieste delle altre per correggere la posizione dei tasselli ignorati. Più in generale delle euristiche per problemi rilassati sono ammissibili per quello originale.

Il vantaggio di mettere in atto questa idea è, inoltre, che i valori dell'euristica per tutte le configurazioni possono essere mantenute in memoria: infatti è facile contare quante sono queste: occorre scegliere 8 posizioni da un insieme di 16 e poi considerare tutte le loro possibili permutazioni, pertanto

$$N_{\text{Conf. distinte}} = \binom{16}{8} \cdot 8! \simeq 0.5 \times 10^9$$

Ossia per memorizzarle è sufficiente “solo” mezzo gigabyte, che è una richiesta ragionevole. La tecnica appena descritta è nota come **pattern database**.

5.1.3 Effective branching factor: confrontare due euristiche

Un'euristica è vantaggiosa perché permette di espandere un numero minore di nodi rispetto al caso della ricerca non informata, ed un modo possibile per confrontare due euristiche è, oltre a cercare di dimostrare la dominanza di una sull'altra, guardare quanti nodi vengono espansi con queste. Una metrica di prestazione utile in questa ottica è la seguente:

Definizione 5.1 (Effective Branching Factor). Sia assegnato un problema di ricerca e sia N il numero di nodi interni espansi da un algoritmo di ricerca; detta d la profondità della soluzione trovata, chiamiamo **effective branching factor** il valore b^* tale che

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

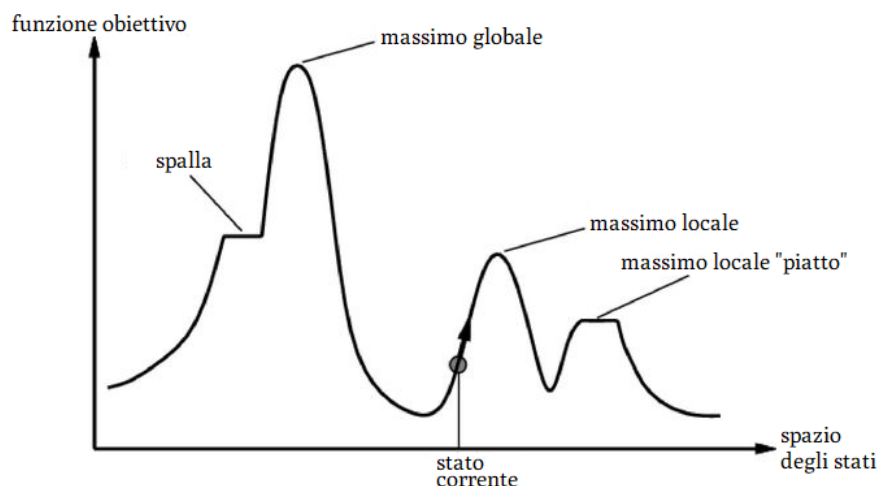
L'idea qui è che l'euristica in media faccia espandere b^* nodi piuttosto di b , ed il caso ideale è quello in cui prende sempre il cammino che porta all'obiettivo, i.e. di volta in volta un solo nodo viene generato, nel qual caso $b^* = 1$. Più tipicamente verranno espansi anche altri nodi, e la bontà di un'euristica si misura in base alla vicinanza dell'effective branching factor ad 1.

Controintuitivamente b^* in generale non è un numero intero ma un numero reale in quanto deve soddisfare l'equazione riportata nella definizione. Inoltre è indicato usare questa metrica soltanto nel caso in cui le euristiche da confrontare si calcolino in tempo $O(1)$, infatti con l'effective branching factor non si riesce a catturare il tempo speso per calcolare l'euristica e quindi non è adatto ad un confronto in tal senso.

5.2 Ricerca locale

In questa sezione presentiamo una classe di algoritmi, detti “di **ricerca locale**”, che opera in maniera profondamente diversa da quelli visti sino ad ora. Fino a questo momento gli algoritmi visti esploravano in maniera sistematica cammini nell’albero di ricerca, e lo facevano tenendo traccia in memoria dei nodi visitati lungo il cammino e quelli candidati per l’espansione. Gli algoritmi di ricerca locale, invece, abbandonano questa strada e come suggerisce il nome stesso si concentrano unicamente su un **nodo corrente** ed i suoi vicini, senza tener traccia dei cammini percorsi, motivo per il quale questi sono adatti a risolvere problemi di ricerca in cui si è interessati solo ad uno stato obiettivo e non alla sequenza di azioni che ci porta a raggiungerlo. Per questo motivo con questa nuova classe di algoritmi sarà necessario tenere in memoria soltanto il nodo corrente, i suoi vicini ed il costo corrispondente, portando ad un’esigenza di memoria spesso costante; questo è uno dei fondamentali vantaggi rispetto agli algoritmi visti fino ad ora, che vedevano proprio nell’elevata richiesta di memoria uno dei loro punti più deboli. Per ragioni simili, questi algoritmi sono applicabili anche in problemi con spazio degli stati molto grande o addirittura continuo, laddove quelli visti fino ad ora sarebbero inadeguati.

Gli algoritmi di ricerca locale comunque rientrano tra quelli di ricerca informata, in quanto la decisione su quale nodo visitare come prossimo continua ad essere guidata da un’euristica; inoltre si possono utilizzare anche per risolvere dei problemi di ottimizzazione, in cui piuttosto di minimizzare un’euristica si vuole massimizzare una funzione obiettivo assegnata.



Un buono strumento per capire come operano questi algoritmi è il **panorama dello spazio degli stati**: un grafico in cui riportiamo sull’asse verticale il valore dell’euristica o della funzione obiettivo a seconda del tipo di problema e sull’asse orizzontale lo spazio degli stati. Se si vuole risolvere un problema di ricerca il nostro obiettivo sarà quello di trovare la gola più profonda, mentre in caso contrario si vorrà trovare il picco più alto.

Per quanto riguarda le proprietà desiderate di un algoritmo, continuano a vigere le definizioni di completezza ed ottimalità invariate rispetto a come le abbiamo definite per i problemi di ricerca tradizionali.

Algoritmo 9 Algoritmo di ricerca locale

```

1: function HILL-CLIMBING(problema)
2:   nodo_corrente  $\leftarrow$  GENERA-NODO( problema.STATO-INIZIALE )
3:   loop
4:     vicino  $\leftarrow$  Il vicino di nodo_corrente con valore più alto
5:     if vicino.VALORE  $\leq$  nodo_corrente.VALORE then
6:       return nodo_corrente.STATO
7:     else
8:       nodo_corrente  $\leftarrow$  vicino

```

5.2.1 Hill Climbing

Abbiamo riportato qui sopra la versione di HILL-CLIMBING che viene utilizzata per risolvere problemi di ottimizzazione, i.e che cerca di massimizzare una funzione obiettivo assegnata, e così faremo d'ora in poi in quanto è più vicino all'intuizione. D'altra parte per passare da un problema di minimo ad uno di massimo è sufficiente cambiare il segno alla funzione da ottimizzare e quindi la discussione che segue non è limitante; questa versione è nota come **scalata più ripida** (*steepest ascent*).

Ciò detto, questo primo esponente degli algoritmi di ricerca locale che vediamo opera in maniera molto semplice e vicina all'intuizione: a partire da un nodo assegnato valuta tutti i vicini e se nessuno di questi porta un miglioramento nella funzione da ottimizzare termina, altrimenti passa dal nodo corrente al suo vicino che porta il maggior miglioramento tra tutti e ripete da capo. In questo, Hill Climbing è molto simile agli algoritmi greedy in quanto si sceglie sempre l'alternativa localmente più promettente, e come questi è tipicamente veloce nel trovare una soluzione. Ciononostante l'algoritmo in questione tende a rimanere bloccato, cosa che può avvenire in questi scenari, con riferimento al panorama del problema:

- **Massimo locale:** Se l'algoritmo seleziona come nodo corrente un massimo locale non è in grado di andare avanti in quanto nessun vicino porta a dei miglioramenti, quindi si arresta senza trovare la soluzione migliore.
- **Cresta:** Il panorama può presentare una cresta, cioè una sequenza di massimi locali che gli algoritmi greedy faticano a navigare e così anche Hill Climbing.
- **Altopiano:** Talvolta può succedere in un panorama che ci sia una regione piatta della funzione obiettivo, ossia in cui questa non varia localmente di valore. Ne distinguiamo di due tipi: una **spalla**, in cui a seguito della regione piatta la funzione obiettivo riprende a migliorare, od un **massimo locale piatto**, che è una regione in cui il valore massimo locale non è assunto da un singolo punto ma lungo un intero intervallo.

Sebbene le prime due situazioni siano critiche e non risolvibili, per quanto riguarda la terza una soluzione, almeno parziale, può essere trovata: possiamo infatti immaginare di permettere all'algoritmo di effettuare anche delle mosse laterali piuttosto che sole mosse che portano a miglioramenti (in pratica passando da \leq a $>$ nella riga 5 dell'algoritmo 9). Questo però apre un'ulteriore problematica: l'algoritmo può rimbalzare all'infinito tra due stati che si trovano sullo stesso altopiano, e per evitare che questo accada si fissa un tetto massimo al numero di mosse laterali consentite all'algoritmo. Con questa miglio-

si aumenta il tasso di successo dell'algoritmo ma anche il numero di mosse richieste in media per terminare.

Varianti stocastiche

L'algoritmo di Hill Climbing è stato implementato in numerose varianti, alcune delle quali impiegano delle tecniche stocastiche al fine di migliorarne qualche aspetto. In questa sottosezione scorriamo quelle principali. La prima che incontriamo è **Hill Climbing stocastico**, che usa un'idea molto semplice: piuttosto di scegliere sempre il vicino che porta il miglioramento più grande si associa a ciascuno di questi una probabilità di essere selezionato che è direttamente proporzionale al vantaggio che porta. Invece, **First-choice Hill Climbing** prevede di generare successori a caso finché non se ne produce uno che ha un valore della funzione obiettivo migliore rispetto a quella del nodo corrente, ed è particolarmente utile quando ciascun nodo ha un elevato numero di vicini, e.g. qualche migliaio, in quanto permette di non doverli espandere tutti.

Infine menzione onorevole per Hill climbing con **random restarts**. L'idea è che se l'algoritmo sta impiegando troppo per trovare una soluzione può convenire di più farlo ripartire da uno stato iniziale generato a caso, infatti è probabile che si fosse partiti da uno stato iniziale sfortunato che ha portato ad una delle problematiche sopra descritte. Oltretutto questa versione dà garanzie di completezza in quanto banalmente prima o poi si genererà un nodo che è associato ad uno stato obiettivo come punto da cui ripartire.

Per quanto riguarda il numero di volte che ci si aspetta di dover ripartire, detta p la probabilità di successo con un'esecuzione del metodo, possiamo modellare il numero di ripartenze prima di trovare soluzione come una variabile aleatoria con distribuzione geometrica, che ha valore atteso $1/p$. Allo stesso modo dato che $(1 - p)/p$ è il numero atteso di fallimenti, possiamo calcolare il costo totale dell'algoritmo come $1/p$ per il costo di un successo più $(1 - p)/p$ per il costo di un fallimento.

Il buon esito di Hill Climbing in ogni caso è fortemente legato alla forma del panorama; problemi NP-Hard sono noti per avere panorami con un numero esponenziale di minimi locali e quindi sono più soggetti a fallimento.

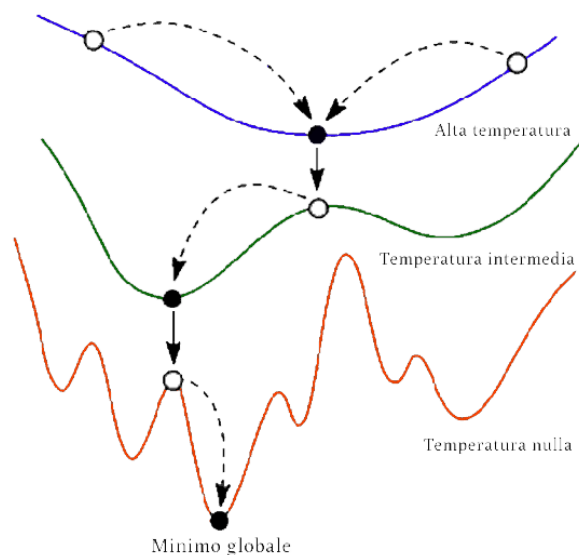
5.2.2 Beam search locale

Questa tecnica di ricerca locale è caratterizzata da un parametro k , il cui valore viene fissato prima di avviare l'algoritmo. Il modo in cui procede è il seguente: si parte con k stati iniziali scelti in maniera casuale, e per ciascuno di questi vengono generati tutti i suoi successori. Se nel processo si è generato uno stato obiettivo l'algoritmo termina, altrimenti si riparte considerando i k nodi migliori generati all'ultimo passaggio fino a trovare uno stato obiettivo.

Si potrebbe pensare che l'algoritmo sia poco diverso dall'eseguire k ricerche con HILL-CLIMBING ciascuna che parte da uno stato iniziale scelto a caso. In realtà c'è una sostanziale differenza tra fare così ed eseguire una Beam Search, infatti nel primo caso tutti i problemi sono indipendenti e l'esecuzione di uno non influenza quella degli altri, mentre nel secondo caso le ricerche sono in comunicazione tra di loro: non è detto che quando si riparte ciascuno dei k nodi provenga da genitori diversi, al contrario, spesso accade che l'algoritmo tenda a concentrarsi in solo alcune regioni dello spazio degli stati. Proprio per questa ragione anche per Beam Search esistono delle varianti stocastiche: una in particolare, detta **beam search stocastica**, prevede di scegliere i k nodi da cui

ripartire in maniera casuale, dove ogni nodo ha probabilità di essere scelto proporzionale al valore ad esso associato.

5.2.3 Simulated Annealing



Negli algoritmi di ricerca locale visti fino a questo momento c'è una problematica sostanziale che è la seguente: nessuno di questi è in grado di garantire di trovare la soluzione ottima, infatti è possibile rimanere intrappolati in altopiani o massimi locali. Più in generale si può far vedere che non c'è modo di garantire l'ottimalità per tutti quegli algoritmi locali che permettono soltanto mosse favorevoli, i.e. mirate all'ottimizzazione della funzione obiettivo. In questa sottosezione presentiamo un algoritmo che garantisce di avere almeno asintoticamente una forma di ottimalità, che si chiama **simulated annealing**.

Cerchiamo anzitutto di provare a dare un'intuizione di come funziona con l'aiuto della figura precedente: la curva rossa (in basso) rappresenta la funzione di cui si vuole trovare il minimo locale. Immaginiamo di riuscire a levigare il grafico fino ad ottenerne una versione più semplice, come potrebbero essere le due curve soprastanti, per le quali è più difficile intrappolarsi in minimi locali e quindi più facile operarci sopra dell'ottimizzazione. Estremizzando l'idea si potrebbe trovare un'approssimazione come la curva blu (in alto) in cui c'è soltanto un minimo: da ovunque si parta, anche con Hill Climbing, si troverebbe la soluzione. Più in generale si può pensare di generare una successione di curve da ottimizzare progressivamente meno "lisce" e più vicine a quella della funzione da ottimizzare fino al limite posarsi sopra; così facendo ogni volta che si risolve il problema di ottimizzazione su una di queste si inizia a concentrarsi sulla regione giusta con la speranza che questa successione porti a trovare la soluzione ottima. In particolare i ogni volta che si passa da una curva meno precisa, come quella blu, ad una più precisa, come quella verde, si fa partire la ricerca sulla nuova curva utilizzando come punto di partenza la soluzione trovata in precedenza.

Questa è chiaramente solo una descrizione informale, mentre adesso vediamo come si fa a passare da un problema di ottimizzazione all'altro di modo da poterci scrivere sopra un algoritmo. Simulated annealing prende ispirazione da un meccanismo diffuso in metallurgia noto come **ricottura** (**annealing**, appunto), utilizzato per far ottenere ad

alcuni materiali delle proprietà desiderabili, come renderli più duttili e facili da lavorare. Nello specifico, questo processo consiste nel portare il materiale ad elevata temperatura, mantenercelo per un breve intervallo di tempo per poi successivamente raffreddarlo in maniera graduale.

Quest'idea viene riprodotta per ottenere il comportamento descritto sopra: è come se ad “alta temperatura” si vedesse il problema in una forma imprecisa, e via via che la temperatura decresce si tende a far diventare il problema di ottimizzazione sempre più vicino a quello reale, con caso limite in cui la temperatura è zero e si recupera proprio il problema originario. Se la diminuzione della temperatura avviene opportunamente si può garantire di riuscire quasi certamente, ossia con probabilità pari ad 1, a trovare l'ottimo globale indipendentemente dal punto di partenza. Simulated annealing è un algoritmo randomizzato: come sottolineato in apertura di sottosezione un algoritmo di ricerca locale che non permette mosse controproducenti non ha speranza di garantire di trovare l'ottimo globale. In particolare vogliamo permettere di eseguire mosse localmente sfavorevoli, ovvero selezionare un successore a caso piuttosto di quello migliore, ma con una probabilità che è tanto più alta quanto più lo è la temperatura: quando questa è elevata si eseguono frequentemente mosse sfavorevoli mentre a temperature più basse non lo si fa quasi mai.

A questo punto diventa rilevante scegliere una buona legge per assegnare le probabilità con cui si sceglie ciascun successore di uno stato, ed anche in questo caso Simulated annealing prende ispirazione da una proprietà della materia conosciuta come **equazione di Boltzmann**. Questa afferma che la frazione P di particelle che occupano uno stato con energia E_i è dato da

$$P = \frac{1}{z} \cdot e^{-E_i/k_B T}$$

Dove il fattore $1/z$ è una *funzione di partizione*, cioè un termine utile a garantire che la somma su tutti i livelli energetici E_i faccia 1, di fatto permettendo di interpretare quel valore come una probabilità. Inoltre $k_B \simeq 8.617 \times 10^{-5}$ è la costante di Boltzmann e T è la temperatura.

In termini algoritmici ricalcheremo esattamente questa legge a patto di cambiare E_i con ΔE definita nel modo seguente, che rappresenta il peggioramento dovuto al passare dal nodo corrente al successore in questione:

$$\Delta E = \text{COSTO}(\text{nodo successore}) - \text{COSTO}(\text{nodo corrente})$$

Anche il calo della temperatura deve avvenire secondo una tabella di marcia opportuna, tipicamente rappresentata da una funzione decrescente e non negativa del tempo che converge a zero (di solito abbastanza lentamente); una scelta possibile è una funzione di questo tipo:

$$T(t) = \frac{T_0}{\log(t + \alpha)}$$

Con T_0 ed α dei parametri che controllano la condizione iniziale e la velocità con cui la funzione converge a zero. Di seguito è riportato un plausibile pseudocodice per l'algoritmo.

Algoritmo 10 Algoritmo di ricerca locale asintoticamente ottimo

```

1: function SIMULATED-ANNEALING(problema, tabella_di_marcia)
2:   nodo_corrente  $\leftarrow$  GENERA-NODO( problema.STATO-INIZIALE )
3:   for  $t = 1, 2, \dots, +\infty$  do
4:      $T \leftarrow$  tabella_di_marcia( $t$ )
5:     if  $T = 0$  then
6:       return nodo_corrente
7:     else
8:       prossimo  $\leftarrow$  successore a caso di nodo_corrente
9:        $\Delta E \leftarrow$  prossimo.COSTO - nodo_corrente.COSTO
10:      if  $\Delta E < 0$  then ▷ Funzione obiettivo migliorata
11:        nodo_corrente  $\leftarrow$  prossimo
12:      else ▷ Funzione obiettivo peggiorata
13:        nodo_corrente  $\leftarrow$  prossimo con probabilità  $p \propto e^{-\Delta E/T}$ 

```

15/10/2020

6.1 Problemi di soddisfacimento di vincoli

Un problema di soddisfacimento di vincoli, abbreviato spesso in CSP, è assegnato una volta che sono dati i tre elementi seguenti:

- Un insieme $X = \{X_1, X_2, \dots, X_n\}$ di variabili;
- Un insieme $\mathcal{D} = \{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_n\}$ di domini finiti, uno per ciascuna variabile;
- Un insieme C di vincoli, cioè regole che stabiliscono quali configurazioni delle variabili in X sono ammissibili.

Ogni dominio è l'insieme dei valori che possono essere assunti dalla variabile corrispondente, mentre ogni vincolo è una coppia $\langle \text{ambito}, \text{relazione} \rangle$, dove con ambito indichiamo la tupla delle variabili che sono coinvolte nel vincolo e con relazione si possono intendere le due seguenti cose: o una lista di tutte le tuple che la soddisfano tale relazione, oppure una rappresentazione astratta che consente due operazioni: verificare se una configurazione delle variabili nell'ambito la soddisfa, e generare tutte le tuple che la verificano. Ad esempio dato come ambito $\{X_1, X_2\}$ con domini $\mathcal{D}_1, \mathcal{D}_2 = \{A, B\}$ il vincolo di diversità tra queste può essere espresso come segue:

- $\langle \{A, B\}, \{(A, B), (B, A)\} \rangle$
- $\langle \{A, B\}, X_1 \neq X_2 \rangle$

Adesso definiamo altri concetti utili nella formulazione di un problema di soddisfacimento di vincoli. Il primo che incontriamo è lo **stato**: diversamente dai problemi di ricerca, adesso uno stato non è più una descrizione astratta di alcuni aspetti rilevanti dell'ambiente ma è un'assegnazione, parziale o completa, delle variabili in X . Un'assegnazione si dice essere **completa** se fissa un valore a tutte le variabili in X , e si dice essere **consistente** se soddisfa tutti i vincoli. Una **soluzione** è un'assegnazione per X che sia completa e consistente. Come per i problemi di ricerca, infine, può essere comoda una rappresentazione di questi problemi tramite grafo, in questo caso non orientato: chiamiamo **grafo dei vincoli** una rete che ha un nodo per ciascuna variabile in X ed un arco tra due qualsiasi variabili che sono coinvolte in uno stesso vincolo.

6.1.1 Inferenza in problemi CSP

Ci affacciamo per la prima volta ad un concetto inedito fino ad ora che è quello dell'inferenza; in particolar modo ne vediamo una forma conosciuta come **propagazione dei vincoli**: possiamo utilizzare i vincoli per ridurre il numero di valori consentiti per una variabile, i.e. cancellarne qualcuno dal suo dominio, e a sua volta questo può portare a cancellare valori da domini di altre variabili. Se effettuiamo l'inferenza prima di lanciare un'algoritmo di risoluzione vero e proprio può succedere che questa da sola riesca a trovare una soluzione o a scoprire che il problema non ne ammette nessuna, se ad un certo punto del processo qualcuno dei domini rimane vuoto. L'inferenza fa uso del concetto di **consistenza locale**, di cui ne vediamo adesso alcune forme:

Consistenza di nodo

Definizione 6.1 (Node Consistency). Una variabile $X_i \in X$ si dice essere **consistente di nodo** se tutti i valori nel suo dominio sono consistenti coi vincoli unari su quella variabile.

Il concetto può essere anche esteso ad un'intera rete: in quel caso si parla di consistenza di nodo quando la proprietà sopra citata vale per ciascun suo nodo, e l'imporre questa forma di consistenza locale permette di eliminare da C tutti i vincoli unari. Più in generale, però, i vincoli non riguardano una sola variabile ma più di queste; in realtà si può dimostrare che tutti i vincoli su un numero $k \geq 2$ di variabili possono essere scomposti in vincoli binari, motivo per il quale molti risolutori di CSP sono impostati proprio per lavorare con problemi i cui vincoli sono tutti binari, ed anche motivo per cui ci spingiamo a cercare altre forme di consistenza.

Esempio 6.1: Da vincolo ternario a binario

Siano A, B, C tre variabili numeriche e si consideri il vincolo $A + B = C$, allora si può scomporre taluno in un vincolo binario e due unari introducendo una nuova variabile X con dominio $\mathcal{D}_X = \mathcal{D}_A \times \mathcal{D}_B$ richiedendo che $X[0] = A$, $X[1] = B$. In questo modo le coppie che soddisfano il nuovo vincolo sono le seguenti se assumiamo $\mathcal{D}_A, \mathcal{D}_B = \{0, 1, \dots, 9\}$:

X	(0,0)	(0,1)	...	(9,9)
C	0	1	...	18



Consistenza d'arco

Definizione 6.2 (Arc Consistency). Assegnate due variabili $X_i, X_j \in X$ unite da un vincolo binario, diciamo che X_i è **consistente d'arco** rispetto ad X_j se

$$\forall a \in \mathcal{D}_i \quad \exists b \in \mathcal{D}_j \quad \text{t.c. } (a, b) \text{ soddisfa il vincolo binario sull'arco } (X_i, X_j)$$

Come prima anche in questo caso la nozione può essere generalizzata: un'intera rete si dice essere consistente d'arco se ogni sua variabile lo è rispetto a tutte le altre. Un'idea interessante può essere di cercare di rendere il grafo dei vincoli consistente d'arco prima di lanciare l'algoritmo di risoluzione vero e proprio di modo da ottenere un problema più semplice con dei domini più piccoli, ed in qualche caso ottenere anche la stessa soluzione senza fare altro o scoprire che questa non esiste.

L'algoritmo più noto per imporre la consistenza d'arco ad un grafo dei vincoli è noto come AC-3: l'elemento centrale su cui esso si appoggia è una lista di archi (vincoli) *coda* che inizialmente contiene tutti gli archi del grafo. Ad ogni iterazione viene estratto un arco (X_i, X_j) a caso e si effettua un **aggiornamento** (*revise*) del dominio della X_i : se questa è già consistente d'arco rispetto ad X_j il dominio rimane inalterato, altrimenti l'aggiornamento cancella da \mathcal{D}_i i valori che causano la mancata consistenza d'arco. Nel fare questo però può succedere che una variabile X_k vicina di X_i nel grafo che prima era consistente rispetto ad X_i smetta di esserlo, e quindi se si sono cancellati valori occorre reinserire in coda tutti gli archi (X_k, X_i) tali che X_k è vicino di X_i nella rete.

Esempio 6.2: Sulla necessità di reinserire archi vicini in coda

Si considerino tre variabili: X_k, X_i, X_j ed i seguenti vincoli:

- $\langle \{X_i, X_j\}, X_i = 2X_j \rangle$
- $\langle \{X_k, X_i\}, X_k = X_i^2 \rangle$

Supponiamo inoltre di avere domini $\mathcal{D}_i = \{0, 1, 2, 3, 4\}$, $\mathcal{D}_j = \{0, 2, 4, 5, 6, 7\}$ e che a seguito di un'aggiornamento avvenuto in precedenza il dominio della variabile X_k sia stato modificato per rendere questa consistente d'arco rispetto ad X_i ottenendo $\mathcal{D}_k = \{0, 1, 4, 9, 16\}$. Se si estrae da *coda* il vincolo su (X_i, X_j) il dominio della X_i viene modificato eliminando i valori in \mathcal{D}_i che causano l'inconsistenza d'arco, cioè in questo caso il solo valore 4. A seguito di ciò X_k cessa di essere consistente d'arco rispetto ad X_i , infatti non esiste nessun valore in X_i tale per cui $X_k = 16$ possa essere visto come X_i^2 , e quindi è necessario un'ulteriore aggiornamento sull'arco (X_k, X_i) . ■

L'algoritmo termina appena la coda si svuota oppure uno dei domini rimane vuoto a seguito di un aggiornamento, nel qual caso il problema di partenza non ammetteva soluzione. Di seguito è riportato uno pseudocodice a cui fare riferimento:

Algoritmo 11 Algoritmo che impone la consistenza d'arco di un grafo dei vincoli

```

1: function AC-3(problema)
2:   while coda non è vuota do
3:      $(X_i, X_j) \leftarrow \text{ESTRAI-PRIMO}( \textit{coda} )$ 
4:     if AGGIORNA( problema,  $X_i, X_j$  ) = true then
5:       if dimensione di  $\mathcal{D}_i$  è zero then
6:         return false
7:       for each  $X_k \in X_i.\text{VICINI} \setminus \{X_j\}$  do
8:         aggiungi  $(X_k, X_i)$  a coda
9:   return true
10:
11: function AGGIORNA(problema,  $X_i, X_j$ )
12:   modificato  $\leftarrow$  false
13:   for each  $x \in \mathcal{D}_i$  do
14:     if  $\nexists y \in \mathcal{D}_j$  tale che  $(x, y)$  soddisfi il vincolo su  $X_i$  ed  $X_j$  then
15:       rimuovi  $x$  da  $\mathcal{D}_i$ 
16:       modificato  $\leftarrow$  true
17:   return modificato

```

Nota che la finitezza dei domini ci garantisce la terminazione di AC-3; inoltre *coda* tipicamente non è implementata come una coda ma come un set. Per quanto riguarda l'analisi dell'algoritmo questa viene fatta in termini di n , numero di variabili del problema, di d , massima dimensione dei domini, e di c , numero di archi nel grafo dei vincoli nonché numero di vincoli nel problema. Ciascun arco può essere inserito in coda al massimo d volte nel caso peggiore, che si realizza quando l'aggiornamento cancella un solo valore ad ogni chiamata, nel qual caso il numero di estrazioni sarà al massimo cd . Per ogni arco estratto da *coda* viene eseguita la funzione AGGIORNA che considera tutte le coppie in $\mathcal{D}_i \times \mathcal{D}_j$ e per ciascuna di queste esegue un'operazione che richiede una quantità costante

di tempo; poiché i domini sono al massimo di dimensione d il costo di questa subroutine nel caso peggiore è un $O(d^2)$ e considerando che AC-3 la invoca al massimo cd volte, nel caso peggiore l'algoritmo avrà un costo che è un $O(cd^3)$, che può essere portato ad un $O(cd^2)$ con una versione aggiornata dell'algoritmo, detta AC-4, ma che ha un costo nel caso medio peggiore rispetto ad AC-3.

Infine osserviamo che la nozione di consistenza d'arco si presta ad una generalizzazione che è la cosiddetta **consistenza d'arco generalizzata**: assegnata una variabile X_i ed un vincolo su n variabili tale proprietà vale se per ogni assegnazione di X_i esiste un membro del vincolo che ha proprio quel valore assegnato ad X_i .

Consistenza di cammino

Definizione 6.3. Siano assegnate due variabili: $X_i, X_j \in X$, ed un'ulteriore variabile X_m , allora si dice che $\{X_i, X_j\}$ è **consistente di cammino** rispetto ad X_m se per ogni assegnazione di X_i ed X_j che rispetta i vincoli esiste un valore corrispondente da assegnare ad X_m tale da soddisfare i vincoli su $\{X_i, X_m\}$ ed $\{X_m, X_j\}$.

La nozione appena formulata è più forte rispetto alla semplice consistenza d'arco e può essere utilizzata per fare un'inferenza più approfondita, ma a costo maggiore. Si possono trovare delle situazioni in cui con questa nozione di consistenza si riesce ad accorgersi che un problema non ammette soluzioni laddove con la sola consistenza di cammino non si sarebbe riusciti.

K-consistenza

In questa sottosezione notiamo che nozioni più forti di consistenza possono essere formulate utilizzando la cosiddetta k -consistenza.

Definizione 6.4 (K-consistency). Un problema di soddisfacimento di vincoli si dice essere **k-consistente** se per ogni insieme di $k-1$ variabili ed ogni loro assegnazione consistente, può sempre essere assegnato un valore alla k -esima variabile di modo da soddisfare i vincoli.

In questa ottica possiamo immaginare che tutte le precedenti nozioni viste siano un caso particolare di k -consistenza: la 1-consistenza è stata chiamata consistenza di nodo, la 2-consistenza è detta consistenza d'arco ed infine la 3-consistenza si chiama consistenza di cammino.

Un problema di soddisfacimento di vincoli si dice essere **fortemente k-consistente** se è k -consistente, $(k-1)$ -consistente, $(k-2)$ -consistente e via dicendo. Avere un CSP con questa proprietà è estremamente vantaggioso: assumiamo di avere n variabili, allora si può assegnare un valore arbitrario ad X_1 e la 2-consistenza ci garantisce di poter scegliere un valore di X_2 consistente con questa assegnazione, inoltre poiché vale anche la 3-consistenza comunque si siano assegnati X_1 ed X_2 si ha garanzia di poter prendere un valore in \mathcal{D}_3 che preso con l'assegnazione di X_1, X_2 produce un'assegnazione consistente, e via dicendo; in questo modo si può riuscire a trovare una soluzione in tempo $O(n^2d)$. Purtroppo però l'algoritmo per imporre la k -consistenza forte richiede sia un tempo che una memoria esponenziali rispetto ad n , pertanto occorre stabilire con un compromesso il livello di consistenza più appropriato.

19/10/2020

7.1 Programmazione a vincoli: Backtracking

Ci sono problemi per i quali la sola inferenza eseguita con strategie viste nel capitolo precedente porta a trovare immediatamente una soluzione. In questo capitolo invece ci occupiamo di tutti quei problemi per i quali questo non avviene, quindi presenteremo e discuteremo una tecnica per risolvere problemi di soddisfacimento dei vincoli.

Un programmatore particolarmente poltrone potrebbe pensare di risolvere un CSP riformulandolo come un problema di ricerca ed applicare l'algoritmo di DLS: sarebbe sufficiente considerare come stati le diverse possibili assegnazioni alle variabili del problema, far corrispondere un'azione all'assegnare un valore ad una variabile e come suo risultato l'aggiungere tale assegnamento allo stato. Il problema è che così facendo non si riescono a sfruttare le buone proprietà del problema, infatti alla radice dobbiamo scegliere una variabile da assegnare tra n possibili ed un valore per questa tra d disponibili, per un totale di dn figli; al primo livello restano da assegnare $n - 1$ variabili con d possibili valori per un numero di figli pari ad $(n - 1)d$ per ciascun nodo, e via dicendo. Alla fine si ottiene un'albero con un numero di nodi interni pari ad $n!d^n$, risultato che può facilmente essere mostrato per induzione, considerando che dopo al massimo n azioni si è completata una qualsiasi assegnazione e quindi l'albero di ricerca ha un'altezza pari ad n .¹ Segue quindi che il fattore di ramificazione sarebbe pari ad nd , mentre il numero di assegnazioni distinte è ovviamente solo $d^n \ll n!d^n$. Questo scostamento è dovuto al fatto che gli algoritmi di ricerca visti fino a questo momento non sono in grado di sfruttare la **commutatività** del problema, ossia il fatto che per raggiungere un certo stato è irrilevante l'ordine con cui si sono eseguite le azioni: proprio per questa ragione, infatti, si può considerare una singola variabile per nodo piuttosto che tutte quelle rimaste da assegnare come in precedenza.

L'algoritmo che vediamo va sotto il nome di **backtracking search**, tradotto talvolta in italiano come *monitoraggio a ritroso*: si tratta di una variante della ricerca in profondità che ad ogni passaggio sceglie una variabile tra quelle non ancora assegnate e prova ad assegnargli uno per volta tutti i possibili valori nel suo dominio, con la particolarità che se viene rilevata un'inconsistenza coi vincoli a seguito di un'assegnazione si fa "marcia indietro" (*backtrack*, appunto) e prova un valore diverso.

Al fine di migliorare le prestazioni opzionalmente è possibile eseguire una procedura di inferenza volta a stabilire la consistenza d'arco, di cammino o k -consistenza in corso di svolgimento piuttosto che prima di lanciare l'algoritmo, in quanto ogni volta che si assegna un valore ad una variabile si presenta l'opportunità di ridurre nuovi domini. Diversamente dai tradizionali problemi di ricerca, inoltre, BACKTRACKING non fa uso di conoscenze specifiche sul problema, in quanto ogni CSP ha una rappresentazione standard, quella vista all'inizio dello scorso capitolo, che è generale e sufficiente per raggiungere il nostro fine. A seguire è riportato uno pseudocodice a cui fare riferimento.

¹Abbiamo infatti scelto DLS perché è nota a priori la profondità di un qualsiasi stato obiettivo

Algoritmo 12 Algoritmo basato su DEPTH-FIRST-SEARCH per risolvere CSP

```

1: function BACKTRACKING-SEARCH(csp)
2:   return BACKTRACKING( {}, csp )
3:
4: function BACKTRACKING( assegnamento, csp )
5:   if assegnamento è un assegnamento completo then
6:     return assegnamento
7:   variabile ← SCEGLI-VARIABILE-NON-ASSEGNATA( csp )
8:   for each valore in ORDINA-VALORI-DOMINIO(variabile,assegnamento,csp) do
9:     if valore è consistente con assegnamento then
10:      aggiungi {variabile = valore} ad assegnamento
11:      inferenze ← INFERENZA( csp, variabile, valore )
12:      if inferenze ≠ fallimento then
13:        aggiungi inferenze ad assegnamento
14:        risultato ← BACKTRACKING( assegnamento, csp )
15:        if risultato ≠ fallimento then
16:          return risultato
17:      rimuovi {variabile = valore} ed inferenze da assegnamento    ▷ Roll back
18:   return fallimento

```

7.1.1 Scelta della variabile da assegnare

Una delle questioni da affrontare è il comportamento della funzione SCEGLI-VARIABILE-NON-ASSEGNATA della linea 7; l'idea più semplice di tutte è quella di assegnare valori alle variabili seguendo un'ordinamento statico, per esempio dopo aver assegnato la variabile X_i si assegna la X_{i+1} fino a produrre un'assegnamento completo. L'idea è molto semplice ma purtroppo difficilmente fare così produce l'esecuzione più efficiente possibile, invece, una scelta più sensata è utilizzare come linea guida il principio noto come **minimum remaining values** (MRV), o anche **most constrained variable**, che suggerisce di scegliere ogni volta la variabile che ha nel proprio dominio il minor numero di valori rimasti. L'intuizione è che il fatto di avere domini piccoli possa derivare dall'avere numerosi vincoli sulla variabile in questione e pertanto seguire questa strada ci faccia fallire, se proprio si deve, il prima possibile, facendo risparmiare tempo. Allo stesso modo utilizzando questa strategia si possono verificare delle situazioni di pareggio in cui più variabili hanno la stessa quantità di valori nei propri domini e questa è minima tra tutte, e qui occorre stabilire come effettuare gli spareggi. Ovviamente si può scegliere la vincitrice dello spareggio a caso, ma si può fare di meglio, ossia usare la cosiddetta **degree heuristic**, che consiste nello scegliere la variabile coinvolta in un numero maggiore di vincoli insieme ad altre variabili non ancora assegnate.

7.1.2 Scelta dell'ordine dei valori

Una volta scelta la prossima variabile a cui dare un valore occorre stabilire con che ordine questi vadano considerati, ossia in pratica come venga implementata la funzione ORDINA-VALORI-DOMINIO della riga 8. In questo caso si è orientati verso una strada opposta rispetto a quella seguita per la scelta della variabile, e si sceglie un'approccio

di tipo *fail-last* di modo da lasciare la maggiore flessibilità possibile per la scelta dei valori delle variabili successive. La strategia in questione si chiama **least constraining value**: si sceglie il valore che cancella il minor numero di possibili valori per i domini delle variabili nel vicinato (sul grafo dei vincoli) di quella scelta. L'intuizione dietro a questa strategia è che meno valori un'assegnazione cancella e più è probabile che questa porti ad una soluzione. Se invece di essere interessati ad una sola soluzione ci interessano tutte, la scelta dell'ordine dei valori è irrilevante.

7.1.3 Inferenza

Se non abbiamo già imposto una qualche forma di consistenza prima di eseguire l'algoritmo si può cogliere l'occasione per fare inferenza in corso d'opera; concettualmente quest'idea potrebbe essere anche più potente, infatti ogni volta che si assegna un valore ad una variabile possiamo cancellare dei valori dai domini delle variabili confinanti sul grafo dei vincoli.

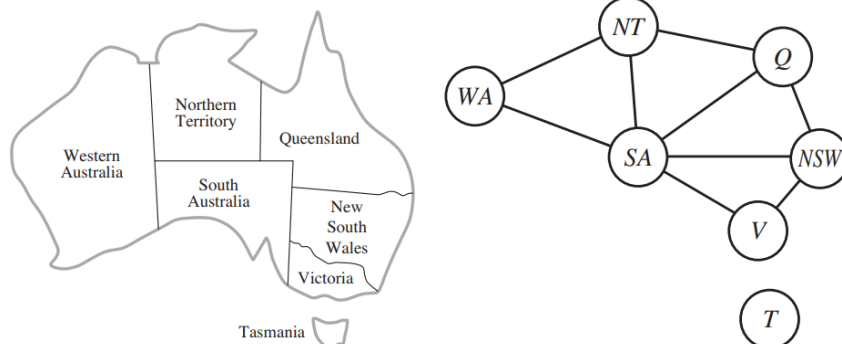
Un'idea semplice in questo caso è quella di eseguire **forward checking**: ogni volta che si assegna una variabile X si cancellano dal dominio di ogni variabile Y legata alla precedente da un vincolo tutti i valori che sono inconsistenti rispetto all'assegnamento scelto per X . Sebbene questa tecnica riesca a rilevare alcune forme di inconsistenza, non riesce a riconoscerle tutte, in quanto impone solo la consistenza d'arco coi nodi vicini e non con tutte le variabili. Spesso questa viene utilizzata insieme ad MRV.

Un'alternativa a forward checking è detta **Maintaining Arc Consistency**: appena si assegna un valore ad una variabile si esegue AC-3 (algoritmo 11), in cui non si inizializza la coda con tutti gli archi della rete ma con solo quelli della forma (X_j, X_i) dove X_j è vicino di X_i , ovvero la variabile assegnata, nel grafo dei vincoli. Questo permette inoltre di accorgersi subito quando una variabile è rimasta senza valori nel proprio dominio, infatti in quel caso AC-3 fallirà permettendoci di fare “marcia indietro” subito.

7.2 Sfruttare struttura del problema

In questa sezione mostriamo come possiamo approfittare di alcune strutture vantaggiose per risolvere i problemi in maniera efficiente.

Si consideri il seguente problema giocattolo: si vuole colorare la mappa dell'Australia utilizzando i colori {rosso, verde, blu} di modo che due qualsiasi regioni confinanti non abbiano mai lo stesso colore.



In questo caso risulta evidente che una qualsiasi assegnazione per la Tasmania presa insieme ad una qualunque soluzione per il continente rappresenti una soluzione per l'intero problema di partenza; più in generale possiamo sperare che sia possibile scomporre il CSP in vari sottoproblemi tutti indipendenti e quindi considerare come soluzione dell'intero problema l'unione delle soluzioni di tutti questi. Illustriamo il guadagno portato da questo approccio: sia n il numero di variabili in X e si assuma che ciascun sottoproblema riguardi c variabili, allora ognuno di questi avrà un costo massimo di d^c , che è il numero di assegnazioni distinte possibili per c variabili con valori presi da insiemi di dimensione d , e dovendone risolvere n/c si otterrà un costo nel caso peggiore pari ad $O(\frac{n}{c}d^c)$ piuttosto che l'usuale $O(d^n)$: siamo passati da un costo esponenziale nel numero di variabili ad uno lineare. Purtroppo però i problemi che permettono una simile fattorizzazione sono rari, quindi valutiamo altre strutture vantaggiose.

L'altra situazione in cui possiamo risolvere il problema in maniera efficiente è quella in cui il grafo dei vincoli associato al problema sia in realtà un albero: in questo caso possiamo esibire un algoritmo in grado di risolvere il problema ancora una volta in tempo lineare rispetto al numero di variabili del problema, ma prima necessitiamo di una ulteriore nozione di consistenza:

Definizione 7.1 (Directed Arc Consistency). Un problema CPS si dice essere **consistente d'arco orientato** rispetto ad una sequenza ordinata delle variabili se ogni variabile è consistente d'arco con tutte quelle che la succedono nella sequenza.

Per rendere un problema consistente rispetto a questa nuova nozione occorre dapprima trovare un'**ordinamento topologico** del grafo, ovvero una sequenza in cui, rispetto ad un nodo scelto a caso come radice, tutti i discendenti di un nodo n compaiono dopo di esso nella sequenza ordinata. Vedremo che è possibile rendere un problema consistente d'arco orientato in n passi ciascuno dei quali impone la consistenza d'arco ed ha quindi un costo di d^2 , per un tempo di $O(nd^2)$. Una volta fatto ciò abbiamo la garanzia che per ogni valore che può essere assegnato ad un genitore, esisterà un valore valido da poter assegnare al figlio e così via lungo tutto l'albero, il che significa che se eseguiamo BACKTRACKING non torneremo mai indietro. Avendo a disposizione un algoritmo efficiente per risolvere

Algoritmo 13 Algoritmo che risolve problemi CSP il cui grafo è un albero

```

1: function RISOLUTORE-CSP-ALBERO(problema)
2:    $n \leftarrow$  numero di variabili in  $X$ 
3:   assegnamento  $\leftarrow \emptyset$ 
4:   radice  $\leftarrow$  una variabile qualunque in  $X$ 
5:    $X \leftarrow$  ORDINAMENTO-TOPOLOGICO(  $X, \text{radice}$  )
6:   for  $j = n, n-1, \dots, 2$  do
7:     RENDI-CONSISTENTE-DI-ARCO( PARENT( $X_j$ ),  $X_j$  )
8:     if è impossibile imporre la consistenza then
9:       return fallimento
10:  for  $i = 1, 2, \dots, n$  do
11:    assegnamento[ $X_i$ ]  $\leftarrow$  un qualsiasi valore consistente in  $\mathcal{D}_i$ 
12:    if non esiste alcun valore consistente in  $\mathcal{D}_i$  then
13:      return fallimento
14:  return assegnamento
```

problemi con grafo associato che è un albero, ci si chiede se esista un modo per ricondurre a questo caso un generico problema di soddisfacimento di vincoli. Illustriamo a seguire due strategie per far sì che questo accada.

7.2.1 Cutset conditioning

La prima strategia che presentiamo prevede di eliminare alcuni nodi dal grafo. Nello specifico, talvolta è possibile individuare un insieme S , detto **cycle cutset**, tale per cui se si eliminano dal grafo dei vincoli tutti i nodi in S e tutti i vincoli riferiti a nodi in S questo diventa un albero. In tal caso per risolvere il problema, una volta individuato il cutset, possiamo seguire i due passaggi seguenti:

- (i) Per ogni possibile assegnazione delle variabili in S consistenti con i vincoli su S stesso si rimuovono dai domini delle variabili rimanenti i valori che sono inconsistenti con l'assegnazione considerata.
- (ii) Fatto ciò, si prova a risolvere il problema rimanente e se ammette soluzione si restituisce l'assegnazione per S e la soluzione trovata.

Per risolvere il problema su S non possiamo sfruttare scorciatoie e nel caso peggiore occorrerà provare tutte le d^c diverse possibili assegnazioni se c è il numero di variabili in S , e per ciascuna di esse risolvere un problema con $n - c$ variabili il cui grafo dei vincoli associato è un albero, per un costo complessivo nel caso peggiore di $O(d^c \cdot (n - c)d^2)$. La strategia è vantaggiosa quando il grafo dei vincoli originario era quasi un albero, in quel caso c è piccolo intuitivamente e quindi si ha un risparmio rispetto all'usuale $O(d^n)$.

7.2.2 Problema duale

La seconda strategia che vediamo prevede di unire dei nodi del grafo per costruire dei “mega-nodi”, ciascuno dei quali rappresenterà un sottoproblema su un sottoinsieme delle variabili di partenza, e sottoproblemi diversi potranno essere collegati tra di loro. Se i raggruppamenti vengono formati in maniera opportuna si può ottenere un nuovo problema ad esso equivalente, cioè con le medesime soluzioni, il cui grafo dei vincoli è un albero. È inoltre richiesto che l'albero in questione soddisfi alcuni requisiti:

- (i) Ciascuna delle variabili del problema di partenza deve comparire in almeno uno dei nodi dell'albero.
- (ii) Se due variabili sono legate da un vincolo binario allora deve esistere un nodo dell'albero che le contenga entrambe.
- (iii) Se una stessa variabile compare in due nodi allora essa dev'essere contenuta anche in ciascun nodo del cammino che li unisce.

Le prime due condizioni servono a garantire che il problema duale rappresenti tutte le variabili e tutti i vincoli di quello di partenza, mentre la terza è utile a garantire che se una variabile compare in più sottoproblemi allora il valore che gli viene assegnato sia lo stesso nella soluzione di ciascuno di questi. Il nuovo problema avrà una variabile per ciascun sottoproblema generato, il dominio associato ad un nodo sarà l'insieme delle soluzioni del sottoproblema corrispondente ed i vincoli riguarderanno l'uguaglianza dei valori delle variabili comuni. Una volta costruito il problema duale la soluzione può essere trovata efficientemente con il risolutore di CSP dell'algoritmo 13.

Per quanto riguarda l'analisi, assegnato un albero di quelli descritti sopra possiamo chiamare **larghezza** (*tree width*) la quantità $w := N - 1$, dove N è il massimo numero di variabili tra i sottoproblemi rappresentati nell'albero, ed inoltre assegnato un grafo possiamo generalizzare questo concetto chiamando la sua larghezza il minimo delle larghezze degli alberi ad esso associabili². Se è noto l'albero a larghezza minima risolvere ciascun sottoproblema richiede un tempo di $O(d^{w+1})$ e se n è il numero di nodi del problema duale, dovremo risolverne proprio n per un costo totale di $O(nd^{w+1})$.

Costruzione di un albero

Alla base della strategia che presentiamo per costruire un **albero di raggruppamenti** (*cluster tree*) associato ad un problema di soddisfacimento di vincoli c'è il seguente concetto:

Definizione 7.2 (Cricca). Assegnato un grafo \mathcal{G} chiamiamo **cricca** un suo sottografo \mathcal{G}' completamente connesso, ossia tale che per ogni coppia di nodi in \mathcal{G}' esista un arco che li connette. Se non esiste nessun'altra cricca contenente \mathcal{G}' si dice che questa è **massimale**.

Definizione 7.3 (Nodo Simpliciale). Assegnato un grafo \mathcal{G} e detto v un suo nodo, si dice che v è **simpliciale** se v ed i suoi vicini inducono un sottografo completo, i.e. in cui ogni coppia di nodi è connessa da un arco.

Proposizione 7.2.1 (Legame simplicialità - cricche). Sia \mathcal{G} un grafo e v un nodo di \mathcal{G} ; sono equivalenti i seguenti fatti:

- (i) v è un nodo simpliciale;
- (ii) Il sottografo indotto da $v \cup \text{VICINI}(v)$ è una cricca;

Per costruire un grafo di raggruppamenti a partire dal grafo dei vincoli del problema di partenza possiamo prendere come clusters proprio le sue cricche massimali ed aggiungere un arco tra due qualsiasi nodi che abbiano delle variabili in comune; inoltre ciascun arco avrà un costo associato pari alla cardinalità dell'intersezione dei nodi associati. Passando a questa riformulazione del problema si ottiene un grafo dei vincoli più "semplice", ma in generale questo non è ancora un albero. Inoltre, come già accennato, deve valere la seguente proprietà:

Proprietà 7.2.1 (Running Intersection). Un albero di raggruppamenti si dice godere della proprietà di **running intersection** se, preso il cammino tra due suoi qualsiasi nodi, la loro intersezione è contenuta in tutti i nodi del cammino.

Un albero di raggruppamenti che gode di questa proprietà viene detto un **junction tree**, e la sua importanza è descritta dal punto (iii) della lista precedente.

Assegnato il grafo di raggruppamenti ottenuto partendo dalle cricche massimali potremmo pensare di estrarre un suo albero ricoprente massimo usando gli algoritmi di Prim o Kruskal e considerare quello come albero del problema duale. Il fatto è che facendo in questo modo non abbiamo garanzie che l'albero ottenuto goda in effetti di running intersection, quindi come strategia generale non funziona. Fortunatamente esiste un algoritmo che ci permette di costruire un junction tree noto come **algoritmo di validazione delle variabili**. Introduciamo due concetti ed un risultato utili ad indirizzarci nella giusta via.

²In generale non esiste un unico albero associabile al grafo. Trovare quello migliore, ossia con larghezza minima, è un problema NP-Hard

Definizione 7.4 (Corda). Assegnato un grafo \mathcal{G} ed un ciclo C in tale grafo, chiamiamo **corda** un arco che appartiene al ciclo.



Nella figura precedente la corda è indicata con una freccia. Legato a questo concetto, vediamo il seguente:

Definizione 7.5 (Grafo Triangolato). Un grafo \mathcal{G} si dice essere **triangolato**, o **cordale**, se ogni ciclo C di lunghezza $\ell \geq 4$ ha una corda.

Diversamente da quanto si potrebbe pensare sulla base del nome, un grafo triangolato non è tale se è composto da triangoli. Vedremo un esempio nella lezione successiva.

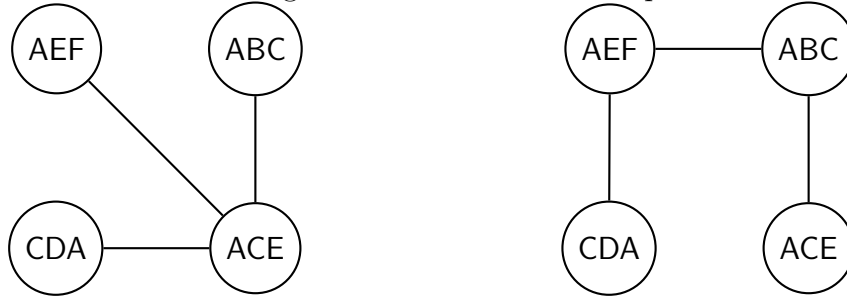
Teorema 7.2.1. Sia \mathcal{G} il grafo dei vincoli di un CSP, allora se esso è triangolato, un albero di copertura massimo del suo duale è un junction tree.

Teorema 7.2.2. Se \mathcal{G} è un grafo triangolato allora \mathcal{G} ha almeno due vertici simpliciali.

Questo teorema ci garantisce che da un grafo triangolato si riesce sempre ad estrarre almeno una cricca. Riguardo teorema che lo precede, notiamo che i pesi degli archi sono numeri interi ed in quanto tali si possono ordinare in tempo $O(c)$ se c sono gli archi del cluster graph, pertanto conviene impiegare l'algoritmo di Kruskal per estrarre l'albero ricoprente massimo a cui il teorema fa riferimento.

Esempio 7.1: Running Intersection

Illustriamo il concetto di running intersection con un esempio:



Consideriamo l'albero di raggruppamenti sulla sinistra: in esso è facile verificare che vale la running intersection, un esempio illustrativo è dato dal cammino $CDA - ABC$: i due nodi hanno in comune le variabili A e C , ed il cammino che connette questi due contiene A, C in tutti i nodi intermedi, in questo caso particolare il solo ACE .

Consideriamo adesso il raggruppamento sulla destra: in questo caso l'albero non gode della running intersection, ad esempio si può considerare il cammino $CDA - ACE$: i due nodi hanno A, C in comune ma nel cammino che li connette è presente un nodo, AEF , che non contiene C .

Facciamo osservare a scanso di equivoci che la proprietà illustrata deve valere per qualunque coppia di nodi. ■

22/10/2020

8.1 Triangolazione di un grafo

In questa sezione discuteremo una tecnica algoritmica che consente di ricavare un junction tree a partire da un grafo non orientato qualsiasi. Se il grafo è in questione è cordale allora per il teorema 7.2.1 è sufficiente costruire un grafo i cui nodi sono le cricche massimali ed i cui archi sono intersezioni tra nodi ed un suo albero ricoprente massimo è già un junction tree. In caso contrario si può applicare una tecnica nota come **triangolazione** che permette di passare da un grafo qualsiasi ad uno cordale e consiste nei seguenti passaggi:

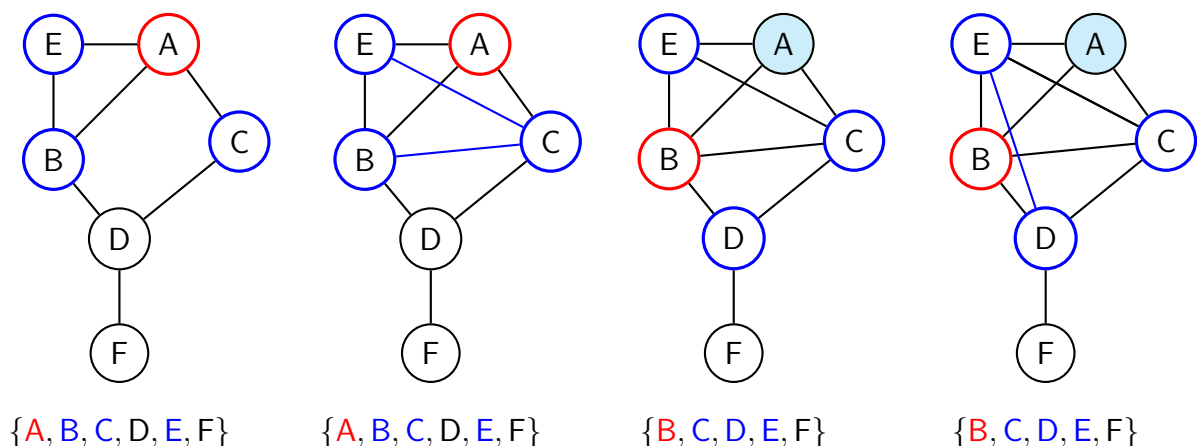
- (i) Si sceglie una sequenza di eliminazione delle variabili.
- (ii) Per ciascuna variabile si connettono attraverso un arco tutti i suoi vicini¹ che non sono già connessi tra di loro.
- (iii) Completato il punto precedente si elimina la variabile dalla sequenza e si ripete da capo fino a che non si è eseguita la procedura per tutte le variabili.

Osservazione 8.1.1. Ognuna delle $n!$ diverse sequenze di eliminazione per le n variabili porta a priori a costruire un junction tree diverso. La migliore sequenza è quella che genera un albero con raggruppamenti migliori², ma trovarla è un problema NP-Hard.

Osservazione 8.1.2. Se si esegue la triangolazione non occorre lanciare un ulteriore algoritmo per cercare le cricche del grafo, infatti quando una variabile viene eliminata essa è connessa ad ogni vicino e questi sono a loro volta connessi tra di sé, pertanto a quel punto la variabile stessa insieme ai suoi vicini costituisce una cricca.

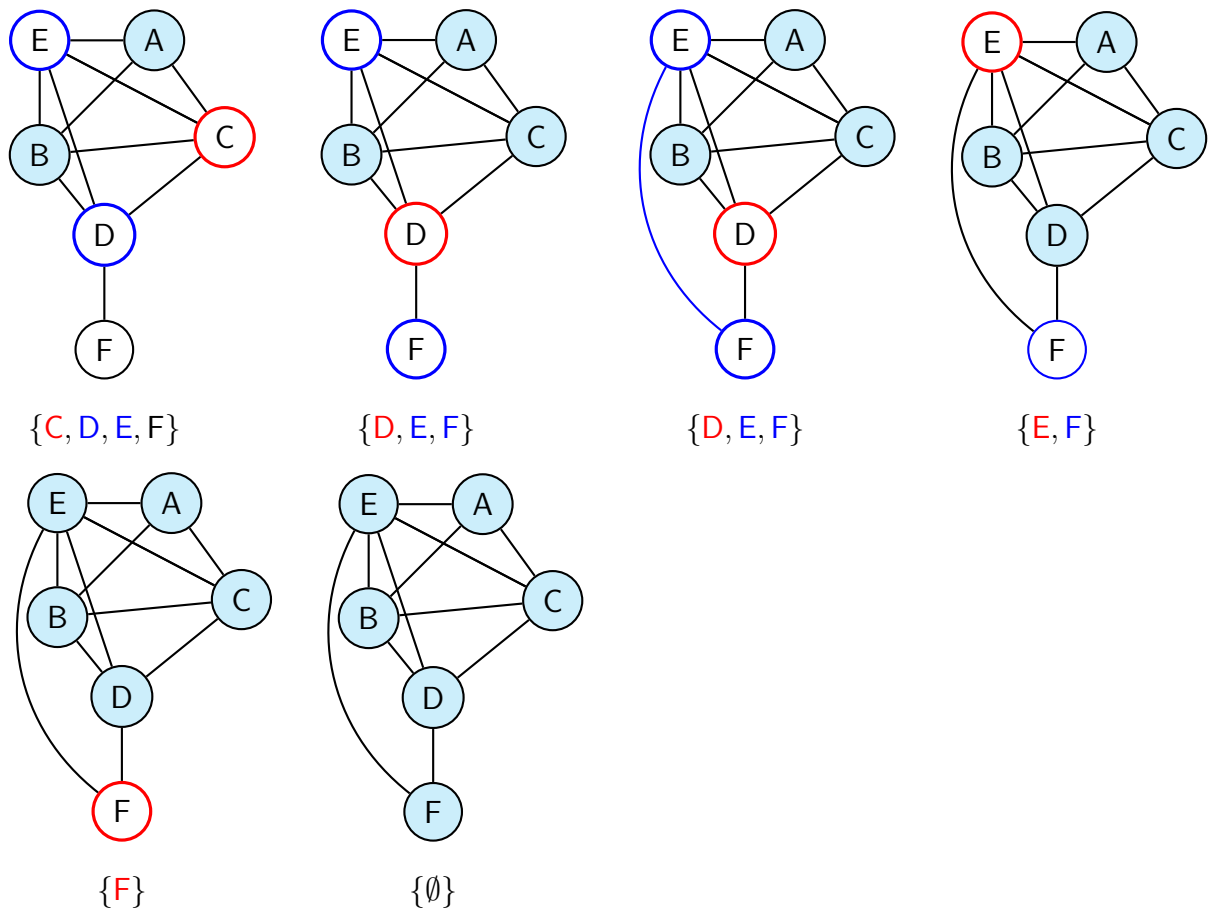
Esempio 8.1: Triangolazione di un grafo

Di seguito riportiamo una possibile esecuzione con una delle possibili sequenze di eliminazione per le variabili dell'algoritmo di triangolazione, a fini illustrativi.



¹Le variabili già eliminate non si considerano come vicini.

²Formalmente, vorremmo l'albero la cui cricca più grande è più piccola possibile



Il rosso indica che la variabile corrispondente è quella che viene considerata in quel passaggio in quanto scelta come prossima dalla sequenza di eliminazione. In blu invece sono segnati i vicini attivi della variabile in rosso, mentre un riempimento azzurro indica che la variabile non viene presa in considerazione perché già eliminata. Si osserva che consideriamo come vicini di una variabile anche quelli acquisiti durante l'esecuzione dell'algoritmo a causa dell'aggiunta di un arco. Gli archi in blu sono quelli aggiunti nel particolare passaggio. ■

Algoritmo 14 Algoritmo per estrarre i clusters da grafo triangolato

```

1: function CRICCHE-MASSIMALI-GRAFO-TRIANGOLATO( $\mathcal{G}$ )
2:   if  $\mathcal{G}$  non è triangolato then
3:     return fallimento
4:    $candidati \leftarrow \emptyset$ 
5:   repeat
6:      $v \leftarrow$  un nodo simpliciale di  $\mathcal{G}$ 
7:     elimina  $v$  dal grafo  $\mathcal{G}$ 
8:      $candidati \leftarrow candidati \cup \{v \cup \text{VICINI}(v)\}$ 
9:   until  $v \cup \text{VICINI}(v)$  contiene tutti i nodi rimasti
10:  for all  $C \in candidati$  do ▷ Pota candidati non massimali
11:    if  $\exists C' \in candidati$  t.c.  $C' \supset C$  then
12:      elimina  $C$  da  $candidati$ 
  
```

L'algoritmo qui sopra permette di estrarre le cricche massimali di cui si parla nell'osservazione 8.1.2. La creazione dell'insieme dei candidati può essere portata avanti all'atto di eliminare una variabile in fase di triangolazione, nel qual caso è sufficiente eseguire la sola potatura piuttosto che l'intero algoritmo 14.

Al termine di questo algoritmo l'insieme *candidati* contiene tutti i raggruppamenti che formeranno il cluster graph da cui estrarre un albero ricoprente massimo, una volta connessi tutti i gruppi con intersezione non vuota. Il teorema 7.2.2 ci garantisce che si riesce ad estrarre almeno un raggruppamento se il grafo di partenza è triangolato.

8.2 Ricerca locale per programmazione a vincoli

Per numerosi problemi abbiamo visto che le tecniche di ricerca locali si comportano piuttosto bene e questo ci spinge ad esplorare in questa direzione anche per la risoluzione di problemi di soddisfacimento di vincoli. In questa sezione vediamo un esponente di tale classe di algoritmi conosciuto come **min-conflicts**. L'algoritmo parte con un'assegnazione completa delle variabili del problema e ad ogni passaggio seleziona una variabile a cui cambiare valore. Il nuovo valore viene scelto sulla base di un'euristica che dà il nome a questo algoritmo: si sceglie infatti tra tutti quello che minimizza il numero di vincoli violati (*conflitti*, appunto). L'algoritmo può terminare perché ha trovato una soluzione o perché ha superato il tetto massimo di passaggi consentiti impostato come parametro.

L'algoritmo in questione si rivela efficace in numerose situazioni, soprattutto se le soluzioni del problema sono dense nello spazio degli stati come nel problema delle n regine, per il cui MIN-CONFLICTS rappresenta la soluzione standard. La problematica principale dell'algoritmo è che non di rado vari valori che possono essere assegnati per la variabile scelta violano lo stesso numero di vincoli, il che in termini del panorama dello spazio degli stati significa che questa euristica può presentare numerosi altopiani. Per non rimanere intrappolati si può applicare una tecnica detta **tabu search**: si mantiene una lista contenente un piccolo insieme degli stati visitati di recente e si impedisce all'algoritmo di rivisitarli sperando che questo porti nuovamente a migliorare il valore dell'euristica.

Un'alternativa è nota come **pesare i vincoli** (*constraint weighting*): ad ogni vincolo viene associato un peso W_i che inizialmente vale 1 e poi viene incrementato ogni volta che il vincolo in questione viene violato. Con questa variante l'algoritmo seleziona come valore quello che minimizza il peso dei vincoli violati. Così facendo, l'algoritmo aggiunge topologia agli altopiani ed indirizza allo stesso tempo verso i vincoli più importanti.

Algoritmo 15 Algoritmo di ricerca locale per risoluzione di CSP

```

1: function MIN-CONFLICTS(csp, max_steps)
2:   corrente  $\leftarrow$  un'assegnamento iniziale completo per il CSP
3:   for  $i = 1, 2, \dots, \text{max\_steps}$  do
4:     if corrente è una soluzione per il CSP then
5:       return corrente
6:     variabile  $\leftarrow$  una variabile a caso in csp.VARIABILI che viola qualche vincolo
7:     valore  $\leftarrow$  valore di variabile che minimizza CONFLITTI( variabile, valore, csp )
8:     imposta {variabile = valore} in corrente
9:   return fallimento

```

Queste tecniche inoltre si rivelano efficaci anche nel risolvere problemi che cambiano dinamicamente nel tempo, come l'organizzazione dei voli in aeroporti, dove il maltempo improvviso può rendere una programmazione precedentemente stabilita non più adatta.

8.3 Introduzione agli agenti logici

Il componente principale di un agente logico è la cosiddetta **base di conoscenze**. Taluna consiste in un insieme di **formule**, cioè frasi espresse in un opportuno linguaggio formale, chiamato “linguaggio di rappresentazione della conoscenza”, che esprimono affermazioni sul mondo.

Talvolta le formule qui presenti sono ricavate a partire da altre, mentre in qualche altro caso sono date, ed in tal caso si parla di **assiomi**.

Riguardo al linguaggio formale di rappresentazione delle conoscenze, in realtà è doveroso precisare che non ne esiste uno unico; quando si vuole lavorare con un agente e scrivere degli algoritmi sarà necessario operare una scelta e prendere uno di questi; qui discuteremo di caratteristiche comuni a tutti.

Ciascun linguaggio è dotato di almeno due elementi, una **sintassi** ed una **semantica**; dopodiché abbiamo detto che una formula è un'asserzione sul mondo. Qui proponiamo alcuni esempi di formule espresse in linguaggio naturale:

- Parigi è la capitale della Francia
- Ogni umano è mortale
- Ogni umano ha una madre ed un padre
- 0 è un numero naturale e se n è un numero naturale allora anche $n + 1$ è un numero naturale.

È chiaro che formule espresse così non sono particolarmente utili per lavorarci sù; occorre formalizzarle, cioè dobbiamo dotarci di una sintassi e di una semantica. Qui entrano in gioco due elementi fondamentali che sono l'**ontologia** e l'**epistemologia**. Con “ontologia” indichiamo cosa assumiamo ci sia nel mondo, e si parla di **impegno ontologico** quando si decide cosa siamo disposti ad ammettere che sia presente nel mondo. Per esempio potremmo decidere in tal senso che il mondo consista di fatti, oppure di oggetti, relazioni e funzioni.

Nel secondo caso, per esprimere che “Ogni umano è mortale” devo essere in grado di distinguere oggetti che sono umani da oggetti che non lo sono, e quindi abbiamo bisogno di una relazione unaria MAN che tra tutti gli oggetti di cui è composto il mondo riconosca gli uomini, poi dovremo avere una ulteriore relazione unaria MORTALS che identifica i mortali, e dire che ogni uomo è mortale equivale a dire che esiste una inclusione di MAN in MORTALS. Se invece vogliamo esprimere “Ogni umano ha una madre ed un padre” dobbiamo ammettere anche le funzioni: dovrà esistere una funzione FATHER che per ogni oggetto del mondo dice chi è il suo padre ed una analoga funzione MOTHER che fa altrettanto per la madre.

Invece per scrivere che “Parigi è la capitale della Francia”, non serve niente di tutto questo: taluno è un fatto. I due mondi, che richiedono ontologie diverse, hanno poteri espressivi profondamente diversi. Una logica composta di fatti si chiama **logica proposizionale**; una composta da oggetti, relazioni e funzioni si chiama **logica del primo ordine**. Logiche più avanzate sono algebricamente indecidibili e quindi non le affrontiamo.

L'epistemologia riguarda quello che possiamo sapere sul mondo. Una elementare epistemologia potrebbe consistere in questo: si può sapere solo se una formula è vera o falsa. Tale epistemologia ha tuttavia un potere espressivo più limitato: non possiamo esprimere con questa ad esempio formule come "L'Everest è una montagna alta", oppure "Biden vincerà le prossime elezioni", infatti ci può essere di mezzo una probabilità, ma nessuno può sapere se la proposizione precedente è vera o falsa.

8.3.1 Generico programma agente logico

Ovviamente un agente deve prevedere delle modalità per interagire con la base di conoscenze, e comunemente tale interazione consiste nella possibilità di svolgere le seguenti azioni:

- Inserire nuove formule nella base di conoscenze
- Interrogare la base di conoscenze

La prima delle due operazioni avviene attraverso una procedura detta TELL, mentre la seconda tramite una procedura di ASK; entrambe queste operazioni potrebbero "nascondere" dell'inferenza, ovvero può accadere che quando eseguite, si renda necessario derivare nuove formule da quelle presenti nella base di conoscenze.

Inizialmente la base di conoscenze potrebbe non essere vuota ma contenere della **conoscenza precostruita** (*background knowledge*).

Ogni volta che un programma agente viene chiamato effettua le tre seguenti azioni: dapprima comunica alla base di conoscenze attraverso un TELL cosa ha percepito, poi chiede attraverso un ASK alla base di conoscenze quale azione dovrebbe svolgere ed infine notifica la base di aver svolto tale azione nuovamente attraverso una procedura di TELL.

Algoritmo 16 Agente Logico Generale

```

1: function KB-AGENT(percezione)
2:   TELL(KB,MAKE-PERCEPT-SENTENCE(percezione, t))
3:   azione ← ASK(KB,MAKE-ACTION-QUERY(t))
4:   TELL(KB,MAKE-ACTION-SENTENCE(azione, t))
5:   t ← t + 1

```

Qui sopra abbiamo riportato lo pseudocodice per il funzionamento di un generico agente logico. La base di conoscenze è rappresentata da un oggetto KB e *t* è un contatore, inizialmente nullo, che scandisce il trascorrere del tempo. MAKE-PERCEPT-SENTENCE è una funzione che in qualche modo non meglio precisato traduce in formula (*sentence* in inglese) le percezioni ricevute in ingresso al tempo *t*, MAKE-ACTION-QUERY genera una richiesta per chiedere alla base di conoscenza quale azione svolgere al dato istante ed infine MAKE-ACTION-SENTENCE produce una formula che informa la base di conoscenza di aver eseguito l'azione "*azione*" specificata come parametro.

La base di conoscenza può essere costruita in due modi diversi: si possono, a partire da una base vuota, aggiungere una ad una tutte le formule che l'agente deve conoscere per poter lavorare nel dato ambiente attraverso una sequenza di TELL, e si parla in tal caso di **approccio dichiarativo**, oppure si possono "scolpire" (*hardcode*) direttamente nel codice le formule che la compongono, ed in tal caso si parla di **approccio procedurale**.

26/10/2020

9.1 Mondo dello Wumpus

Di seguito definiamo un possibile ambiente in cui un agente logico può operare che servirà come riferimento per diversi esempi che faremo durante la nostra trattazione della logica.

Il mondo dello Wumpus consiste di varie stanze connesse da dei corridoi. In una di queste si aggira il terrificante Wumpus, una belva che divora chiunque osi avventurarsi nella sua stanza. Tale mostro può essere colpito dall'agente, il quale però ha al proprio arco una sola freccia.

Alcune delle stanze contengono fosse senza fondo, che possono intrappolare chiunque vi si avventuri, tranne lo stesso Wumpus, che è troppo grande per caderci dentro.

L'unico aspetto positivo in questo truce mondo consiste nella possibilità di trovare un mucchietto d'oro.

L'agente non è in grado di vedere ciò che si trova nelle stanze adiacenti alla propria posizione ma può fare affidamento su alcuni segnali:

- Nelle stanze direttamente adiacenti, ma non diagonali, a quella in cui si trova lo Wumpus, l'agente può percepire un fetore.
- Nelle stanze direttamente adiacenti ad una fossa, l'agente può percepire un venticello.
- Nella stanza, unica, in cui è presente il mucchio d'oro l'agente può percepire uno scintillio.

Inoltre sono presenti due segnali “secondari”: quando l'agente sbatte contro un muro percepisce un *colpo*, e quando lo Wumpus muore emette un *grido* funesto avvertibile in tutta la caverna. Assumeremo quindi come percezione il seguente vettore: [*Fetore*, *Venticello*, *Scintillio*, *Colpo*, *Grido*].

Per quanto riguarda le azioni che l'agente può compiere, permetteremo i movimenti *avanti*, oltre che di *girarsi a destra* e *girarsi a sinistra*; permetteremo inoltre all'agente di *raccollecte* l'oro e di *sparare* la propria freccia (solo in linea retta nella direzione in cui esso è rivolto).

Per quanto riguarda l'ambiente, sarà sufficiente assumere che la caverna consista in una griglia di stanze 4×4 , e quella in cui si trova lo Wumpus viene scelta a caso con probabilità uniforme, così come la posizione del mucchio d'oro; invece in fase di generazione della caverna, si stabilisce che ciascuna cella possa contenere una fossa con probabilità 0.2, tranne quella iniziale che ovviamente non dovrà contenerne.

Infine, per quanto riguarda la misura di prestazioni, si stabilisce che l'agente verrà ricompensato con un +1000 sul punteggio se riesce a scappare dalla caverna con l'oro, -1000 se cade in una fossa o viene sbranato dalla belva. Ogni azione eseguita costerà -1 sul punteggio e lo sparo della freccia penalizza l'agente per -10 punti.

Quando l'agente viene divorato o riesce a fuggire dalla caverna, il gioco termina.

9.2 Elementi di logica

In conclusione della scorsa lezione abbiamo affermato che ciascuna formula viene espressa in base ad una sintassi ed una semantica. La prima delle due definisce quali formule sono ben formate e quali no; ad esempio se prendiamo come linguaggio di rappresentazione l'aritmetica, la formula “ $x + y = 4$ ” è valida, mentre “ $x4y+$ ” no.

Invece il concetto di semantica ha a che vedere con il significato delle formule: essa definisce infatti il **valore di verità** di ogni formula rispetto a tutti i possibili mondi; per esempio $x + y = 4$ è vera in un mondo in cui $x = 2$ ed $y = 2$ mentre non lo è in uno in cui $x = 2$ ed $y = 1$.

Quando è richiesto di essere formali, al posto di “possibile mondo” parleremo di **modello**¹; chiamiamo così un'astrazione matematica che assegna un valore di verità ad ogni formula rilevante. I vari modelli in questo senso consisteranno in tutte le possibili assegnazioni di valori di verità per le varie formule. Se una formula α è vera in un certo modello m , si dice che m **soddisfa** α , oppure a volte, che m **è un modello di** α , e l'insieme di tutti i modelli di α lo indicheremo come $M(\alpha)$.

9.2.1 Entailment

Uno strumento di cui ci serviremo estensivamente nella nostra trattazione è il cosiddetto **entailment** tra formule. Taluna è una relazione tra due formule: α e β , e matematicamente quando vale tale relazione si scrive che $\alpha \models \beta$. Diamo dunque una definizione formale:

Definizione 9.1 (Entailment). Siano α e β due formule, allora si dice che α **comporta** β , e si scrive che $\alpha \models \beta$, se e solo se ogni modello che rende vero α rende vero anche β , i.e.

$$\alpha \models \beta \iff M(\alpha) \subseteq M(\beta)$$

Quindi se α comporta β , la definizione precedente suggerisce che α debba essere una affermazione più forte di β ; per esempio rimanendo fedeli all'aritmetica, la formula “ $x = 0$ ” comporta “ $x \cdot y = 0$ ” in quanto ogni modello che rende vera la prima rende vera anche la seconda, ma il viceversa non è vero; infatti è più stringente richiedere che $x = 0$ piuttosto che $x \cdot y = 0$.

La definizione del concetto di **entailment** può essere applicata immediatamente per prendere delle decisioni, cioè effettuare dell'inferenza logica; è immediato pensare ad un algoritmo in grado di decidere se una formula α comporta una seconda formula β semplicemente verificando per ogni possibile modello se vale la definizione, e più in generale per immediata estensione, in grado di decidere se una base di conoscenze KB comporta una formula α . Un algoritmo che segue questa via per tale task si chiama **model checking**.

In generale non è detto che se $KB \models \alpha$ un generico algoritmo i riesca a ricavare α dalla base di conoscenza KB; se ci riesce si scrive che $KB \vdash_i \alpha$. In tal senso, definiamo due proprietà desiderabili di cui potrebbero o meno godere i vari algoritmi di inferenza: la prima proprietà che vediamo è la **ragionevolezza** (*soundness*). Diciamo che un algoritmo è “ragionevole” se ricava solamente frasi che derivano dalla base di conoscenza, o in

¹A lezione chiamiamo taluno *interpretazione*, e modello una di queste che rende vera una formula. Di seguito viene usata la terminologia di [RN10].

termini più formali, se $KB \vdash \alpha \implies KB \models \alpha$. La seconda proprietà che vediamo invece è la **completezza**: un algoritmo di inferenza si dice essere “completo” se ricava tutte le formule che seguono dalla base di conoscenza, ossia se $KB \models \alpha \implies KB \vdash \alpha$. Se un algoritmo è sia ragionevole che completo allora l’entailment corrisponde con la derivazione.

9.2.2 Logica proposizionale

Di seguito ci occupiamo della sintassi e della semantica della logica proposizionale, per poi passare alla relazione di entailment, mostrando come questa porti a definire un algoritmo semplice per l’inferenza logica.

Per quanto riguarda la sintassi, nella logica proposizionale le formule atomiche sono composte da **simboli proposizionali**, ciascuno dei quali rappresenta una proposizione che può essere vera o falsa. È consuetudine per tali simboli utilizzare alternativamente una lettera maiuscola, come P, Q , oppure parole che iniziano con lettere maiuscole, permettendo la presenza di pedici, ad esempio $W_{1,2}$. Oltre a questi esistono dei simboli proposizionali che hanno un valore di verità fissato, ad esempio *True* è la proposizione sempre vera e *False* è la proposizione sempre falsa. Le frasi atomiche vengono dette anche **letterali**.

A partire da formule atomiche se ne possono comporre di complesse combinando le prime attraverso connettivi logici e parentesi (per modificare le priorità). I connettivi che prendiamo in considerazione sono i seguenti:

- \neg (not). Assegnata una formula P , chiamiamo $\neg P$ la sua **negazione**. In tal senso un letterale che compare non negato si dice essere **positivo**, mentre in caso contrario, se esso appare negato, si dice essere un letterale **negativo**.
- \wedge (and). Assegnate due formule P e Q , chiamiamo la formula $P \wedge Q$ la **congiunzione** di P e Q , e i due operandi si chiamano **congiunti**.
- \vee (or). Assegnate due formule P e Q , chiamiamo la formula $P \vee Q$ la **disgiunzione** di P e Q .
- \implies (implicazione). Una formula della forma $P \implies Q$ prende il nome di **implicazione**; P viene detta **premessa** e Q **conclusione**. Altri nomi per questo tipo di formula sono “regole” o “istruzioni if-then”.
- \iff (se e solo se). La formula $P \iff Q$ viene detta **bicondizionale**.

Questi sono i simboli consentiti nella nostra logica proposizionale. Data una frase complessa, essa potrebbe contenere uno o più di questi simboli e quindi occorre, per eliminare eventuali ambiguità, definire delle precedenze; in tal senso si stabilisce che il connettivo \neg prende priorità massima. Ad esempio in $\neg P \wedge Q$ dapprima si “calcola” $\neg P$ e solo dopo la congiunzione. Dopodiché la congiunzione ha priorità sulla disgiunzione, e le due implicazioni hanno priorità minima. Ovviamente l’utilizzo di parentesi ha come effetto quello di modificare l’ordine di precedenza appena stabilito.

Per quanto riguarda la semantica, invece, dobbiamo definire le regole che permettano di stabilire i valori di verità di formule complesse rispetto ad un dato modello, che stabilisce il valore di verità di ogni simbolo proposizionale.

La semantica, quindi, deve specificare come calcolare il valore di verità di una qualsiasi frase assegnato il modello. Facciamo ciò in maniera ricorsiva. Tutte le formule sono composte a partire da formule atomiche, quindi dobbiamo indicare come calcolare il valore di verità di queste ultime e subito dopo, come calcolare il valore di verità di formule ottenute a partire dai cinque connettivi. Per quanto riguarda le frasi atomiche:

- Stabiliamo che il loro valore di verità di ciascuna di queste debba essere immediatamente fissato nel modello.
- *True* è vera in ogni modello, *False* è falsa in ogni modello.

Invece per quelle complesse, facciamo fede al seguente insieme di regole:

- $\neg P$ è una formula vera se e solo se P è falsa nel modello m .
- $P \wedge Q$ è una formula vera se e solo se sia P che Q sono vere nel modello m .
- $P \vee Q$ è una formula vera se e solo almeno una tra P e Q è vera nel modello m .
- $P \implies Q$ è una formula vera in tutti i casi tranne qualora P sia vera e Q sia falsa nel modello m .
- $P \iff Q$ è una formula vera se e solo se P e Q sono entrambe vere o entrambe false nel modello m .

Un modo veloce per esprimere queste regole è attraverso le **tabelle di verità**, che indicano il valore di verità della formula risultante per ogni possibile combinazione dei valori di verità dei suoi componenti, e queste possono essere usate per determinare ricorsivamente il valore di verità di una formula rispetto ad un qualsiasi modello.

Un semplice algoritmo di inferenza logica

Adesso il nostro obiettivo è quello di trovare un algoritmo in grado di stabilire, per qualche base di conoscenza KB e formula α , se tale base comporta α .

Qui sopra la funzione TT-ENTAILS ha come compito quello di chiamare la funzione TT-CHECK-ALL con gli opportuni parametri iniziali. Per quanto riguarda quest'ultimi, KB ed α è ovvio chi siano, *simboli* è la lista dei simboli non ancora assegnati nel modello in costruzione e *modello* è una assegnazione (parziale o completa) dei simboli proposizionali coinvolti nella richiesta.

Commentando questo algoritmo, è intuibile che anche lui come model checking è basato sulla definizione di entailment. Se chiamassimo taluno come segue:

$$\text{TT-CHECK-ALL}(KB, \alpha, [P, Q], [])$$

Si otterrebbe in risposta il risultato della seguente elaborazione:

```
(
  TT-CHECK-ALL(KB,  $\alpha$ , [], [P = true, Q = true])
  and
  TT-CHECK-ALL(KB,  $\alpha$ , [], [P = true, Q = false])
)
and
(
  TT-CHECK-ALL(KB,  $\alpha$ , [], [P = false, Q = true])
  and
  TT-CHECK-ALL(KB,  $\alpha$ , [], [P = false, Q = false])
)
```

Algoritmo 17 Entailment da Tabelle di Verità

```

1: function TT-ENTAILS( $KB, \alpha$ )
2:    $simboli \leftarrow$  Lista dei simboli proposizionali in  $KB$  ed  $\alpha$ 
3:   return TT-CHECK-ALL( $KB, \alpha, simboli, \{\}$ )
4:
5: function TT-CHECK-ALL( $KB, \alpha, simboli, modello$ )
6:   if ISEMPTY( $simboli$ ) then
7:     if PL-ISTRUE( $KB, modello$ ) then
8:       return PL-ISTRUE( $\alpha, modello$ )
9:     else
10:      return true
11:   else
12:      $P \leftarrow$  FIRST( $simboli$ )
13:      $restanti \leftarrow$  REST( $simboli$ )
14:     return ( TT-CHECK-ALL( $KB, \alpha, restantes, modello \cup \{P = true\}$ )
               and
               TT-CHECK-ALL( $KB, \alpha, restantes, modello \cup \{P = false\}$ ) )

```

La prima parte di TT-CHECK-ALL, che viene eseguita quando è stato assegnato un valore a tutti i simboli dal modello, controlla se il modello corrente, ad esempio $[P = true, Q = false]$, soddisfa la base di conoscenza. I modelli che passano questa condizione corrispondono a righe nella tabella di verità che hanno *true* nella colonna di KB , e per loro l'algoritmo verifica se il modello soddisfa anche la query α . Tutti i modelli che non soddisfano la base di conoscenza non sono rilevanti al fine di decidere se vale o meno la relazione di entailment e quindi ci si limita a restituire *true*, che è l'elemento neutro della congiunzione.

La seconda parte dell'algoritmo invece viene utilizzata per costruire tutti i possibili modelli: di volta in volta si prende il primo simbolo non ancora assegnato dal modello e lo si aggiunge a quest'ultimo, una volta con valore *true* ed una volta con *false* (ovviamente si generano così due modelli distinti).

In definitiva l'algoritmo si limita a verificare se $PL-ISTRUE(KB, modello) \implies PL-ISTRUE(\alpha, modello)$, non facendo altro che sfruttare la definizione di entailment, come già detto in precedenza.

Per quanto riguarda l'analisi di questo algoritmo, osserviamo che TT-ENTAILS si limita a generare ricorsivamente un insieme finito di assegnazioni ai simboli. Sicuramente si ha la **ragionevolezza** in quanto viene implementata direttamente la definizione di entailment, ed inoltre abbiamo anche **completezza** poiché l'algoritmo funziona chiunque siano KB ed α , e termina sempre, essendo i modelli in quantità finita.

Se l'insieme $simboli$ contiene n simboli proposizionali allora i diversi possibili modelli sono 2^n , e pertanto la complessità temporale è di $O(2^n)$, mentre la complessità spaziale è un $O(n)$, lavorando esso in maniera simile alla DEPTH-FIRST.

Teorema 9.2.1 (Decidibilità Entailment). In logica proposizionale l'entailment è decidibile

Dimostrazione. La dimostrazione di questo risultato è banale, infatti abbiamo esibito un

algoritmo in grado di enumerare tutti i possibili modelli, che per n simboli proposizionali sono 2^n e quindi in quantità finita, ed applicare la definizione di entailment. \square

9.2.3 Theorem Proving

Il **theorem proving** consiste nell'applicare regole di inferenza direttamente alle formule contenute nella base di conoscenza per costruire una dimostrazione di una formula desiderata, il tutto senza consultare alcun modello. Questo risulta vantaggioso quando il numero di possibili modelli diversi è grande ma la lunghezza della dimostrazione è piccola.

Prima di vedere algoritmi che usino questo metodo però, abbiamo bisogno di ulteriori concetti legati all'entailment. Il primo di questi è l'**equivalenza logica**: diciamo che due formule α e β sono logicamente equivalenti se sono vere nello stesso insieme di modelli, e si scrive che $\alpha \equiv \beta$. In alternativa, possiamo dire che due formule sono logicamente equivalenti se $\alpha \models \beta$ e $\beta \models \alpha$.

Il secondo concetto che vediamo è quello di **validità**: una formula α è valida se è vera in tutti i modelli; ad esempio $P \vee \neg P$ è valida. Formule di questo tipo sono note anche come **tautologie**, in quanto sono necessariamente sempre vere. Sfruttando questo concetto possiamo formulare il seguente risultato

Teorema 9.2.2 (Teorema di Deduzione). Per ogni coppia di formule α e β , $\alpha \models \beta$ se e solo se la formula $(\alpha \implies \beta)$ è valida.

Quindi per decidere se $\alpha \models \beta$ potremmo verificare se $\alpha \implies \beta$ è vera in tutti i modelli, ovvero se quest'ultima è logicamente equivalente a *True*.

Infine, come ultimo concetto, introduciamo quello di **soddisfacibilità**: una formula α si dice essere soddisfacibile se esiste almeno un modello che la rende vera. Il modo ovvio di verificare se una formula gode di questa proprietà è generare tutti i modelli finché non se ne incontra uno che la renda vera. Il problema di determinare la soddisfacibilità di una formula in logica proposizionale, noto come **SAT**, è però NP-completo.

Esiste un ovvio legame tra i concetti di soddisfacibilità e validità: infatti una formula α è valida se e solo se $\neg\alpha$ non è soddisfacibile. In particolare questa osservazione conduce al risultato che segue²:

$$\alpha \models \beta \text{ se e solo se la formula } (\alpha \wedge \neg\beta) \text{ non è soddisfacibile}$$

Regole di inferenza

Riportiamo di seguito regole di inferenza che possono essere applicate per produrre delle dimostrazioni. La prima di queste che affrontiamo è la seguente:

Definizione 9.2 (Modus Ponens). Siano α e β due formule, se sappiamo che $\alpha \implies \beta$ è vera e che α è vera, allora possiamo inferire che anche β è vera. Formalmente si scrive che

$$\frac{\alpha \implies \beta, \quad \alpha}{\beta}$$

Immediatamente dopo segue la regola di inferenza seguente:

²Ricorda che $\alpha \implies \beta \equiv \neg\alpha \vee \beta$

Definizione 9.3 (And-Elimination). Siano α e β due formule, se sappiamo che $\alpha \wedge \beta$ è vera allora possiamo inferire che α è vera. Formalmente si scrive che

$$\frac{\alpha \wedge \beta}{\alpha}$$

Questa regola d'inferenza può essere usata anche per inferire che β è vera, infatti $\alpha \wedge \beta$ è vera se e soltanto se lo sono entrambe le sue costituenti (Un esempio di applicazione di queste regole per ricavare una dimostrazione si può trovare in [RN10] al capitolo 7.5).

Il vantaggio fondamentale del theorem proving è che in molti casi esso risulta più efficiente di model checking, in quanto vengono ignorate proposizioni nella base di conoscenza che sono irrilevanti ai fini della dimostrazione, che esse siano due o due milioni; nel secondo caso l'algoritmo basato su tabelle di verità TT-ENTAILS visto nella sezione precedente verrebbe invece sopraffatto dall'incremento esponenziale nel numero di modelli.

Un'ultima proprietà che vediamo per quanto riguarda i sistemi logici è la **monotonicità**: l'insieme delle formule che seguono dalla base di conoscenze può solamente crescere quando si aggiungono informazioni alla base di conoscenza; formalmente, se KB è una base di conoscenza, α e β due formule

$$(KB \models \alpha) \implies (KB \wedge \beta \models \alpha)$$

Regola di risoluzione

Considerando tutti i possibili valori di verità per α e β è facile convincersi che le regole di inferenza precedentemente introdotte, come modus ponens ed And-elimination, siano ragionevoli; invece non ci siamo ancora chiesti se algoritmi che usano queste siano o meno completi. Un'idea possibile è quella di formulare il problema di trovare la dimostrazione di una formula come un problema di ricerca in cui lo stato iniziale è la base di conoscenza, le azioni sono l'insieme di regole di inferenza che possono essere applicate, il risultato di un'azione è quello di aggiungere la formula al “denominatore” della regola di inferenza applicata e l'obiettivo è uno stato che contenga la formula che stiamo cercando di dimostrare. Infine potremmo eseguire ITERATIVE-DEEPENING (algoritmo 8) come algoritmo di ricerca per risolvere il problema così formulato. La capacità di trovare formule che seguono dalla base di conoscenza usando questa strada dipende dall'insieme di regole di inferenza utilizzato: se questo non è adeguato allora possono esistere dei goals non raggiungibili, cioè può succedere che non esista una dimostrazione che utilizzi le sole regole messe a disposizione, quindi in generale non possiamo affermare di avere completezza.

Invece adesso presenteremo una regola di inferenza, detta **risoluzione** che porta ad algoritmi di inferenza completi quando utilizzata insieme ad algoritmi di ricerca completi.

Definizione 9.4 (Risoluzione). Siano P, Q, R tre formule, allora la regola di risoluzione, nella sua versione più elementare, afferma che

$$\frac{P \vee Q, \quad \neg P \vee R}{Q \vee R}$$

Convinciamoci che quanto suggerisce la regola di risoluzione sia effettivamente vero: dobbiamo supporre di sapere che $P \vee Q$ sia vero ed anche $\neg P \vee R$ sia vero. Si presentano le due seguenti possibilità

- Se P è vero allora, poiché $\neg P \vee R$ è vero, necessariamente R è vero;
- Se P è falso allora, poiché $P \vee Q$ è vero, necessariamente Q è vero;

Quindi in ogni caso, almeno uno dei due tra Q ed R dev'essere vero sotto la precedente ipotesi, e quindi deve essere vero $Q \vee R$, e quest'ultima formula viene chiamata **risolvente**.

Esiste una versione più generale del principio di risoluzione, che prende il nome di **risoluzione unitaria**, e si formula in questi termini: siano $\ell_1, \ell_2, \dots, \ell_k$ dei letterali e si supponga che ℓ_i ed m siano dei letterali complementari, i.e. con valori di verità opposti; allora si può inferire che

$$\frac{\ell_1 \vee \ell_2 \vee \dots \vee \ell_k, \quad m}{\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k}$$

Infatti, se m è vera necessariamente ℓ_i è falsa, quindi non era ℓ_i che rendeva vera la disgiunzione di ℓ_1, \dots, ℓ_k , e può essere perciò rimossa senza cambiare il fatto che la **clausola** (i.e. disgiunzione di letterali) risultante continui ad essere vera.

La formulazione più generale possibile, però, è ancora un'altra: possiamo considerare due clausole: la prima composta dai letterali $\ell_1, \ell_2, \dots, \ell_k$ e la seconda da m_1, m_2, \dots, m_k ; si suppone inoltre che ℓ_i ed m_j siano letterali complementari, allora si può inferire che

$$\frac{\ell_1 \vee \ell_2 \vee \dots \vee \ell_k, \quad m_1 \vee m_2 \vee \dots \vee m_k}{\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n}$$

Convinciamoci che questa regola di inferenza sia ragionevole: se ℓ_i è vera allora m_j è falsa e quindi deve essere vera $m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n$; d'altra parte invece se ℓ_i è falsa allora deve essere vera $\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k$ perché si sa che è vera $\ell_1 \vee \ell_2 \vee \dots \vee \ell_k$. Poiché non ci sono altre possibilità, almeno una tra le due clausole che compongono il “denominatore” deve essere vera, e quindi deve essere tale anche la loro disgiunzione, che è proprio ciò che asserisce la regola di risoluzione.

Forma normale congiuntiva

La regola di risoluzione si applica solamente a clausole, cioè a disgiunzioni di letterali; a priori potrebbe sembrare che sia utile solamente in quei casi in cui sia la base di conoscenza che la query consistano di clausole. In realtà esiste un risultato che ci garantisce che ogni formula della logica proposizionale è logicamente equivalente ad una congiunzione di clausole; un esempio può essere trovato nella sezione 7.5.2 di [RN10].

Una formula espressa nella forma di congiunzione di clausole si dice essere in **forma normale congiuntiva**, spesso abbreviato in CNF.

Un algoritmo di risoluzione

Gli algoritmi di inferenza basati sulla regola di risoluzione lavorano sulla base della **dimostrazione per assurdo**, ossia, per dimostrare che $\text{KB} \models \alpha$ fanno vedere che la formula $(\text{KB} \wedge \neg \alpha)$ non è soddisfacibile.

Di seguito viene riportato un algoritmo basato sulla regola di risoluzione: dapprima la formula $\text{KB} \wedge \neg \alpha$ viene convertita in forma normale congiuntiva e poi la regola di risoluzione viene applicata alle clausole risultanti. Ogni coppia di clausole che contiene letterali complementari produce una nuova clausola che viene aggiunta all'insieme delle clausole se non già presente.

La procedura si protrae fintantoché non si verifica una di queste situazioni:

- Non ci sono più clausole da aggiungere, nel qual caso $KB \not\models \alpha$
- Due clausole danno come risolvente la clausola vuota, nel qual caso $KB \models \alpha$

Si noti che la clausola vuota è logicamente equivalente a *False*, infatti una di queste è vera se e soltanto se almeno uno dei letterali che la compongono è vero; non contenendo alcun letterale, la clausola $\{ \}$ di fatto non può essere vera in nessun modello. Nella lezione successiva dimostreremo formalmente che questo algoritmo è anche completo.

Algoritmo 18 Algoritmo di inferenza basato su Risoluzione

```

1: function TT-RESOLUTION( $KB, \alpha$ )
2:    $clausole \leftarrow$  L'insieme delle clausole nella rappresentazione CNF di  $KB \wedge \neg \alpha$ 
3:    $nuove \leftarrow \{ \}$ 
4:   loop
5:     for each coppia di clausole  $C_i, C_j$  in  $clausole$  do
6:        $risolventi \leftarrow$  PL-RESOLVE( $C_i, C_j$ )
7:       if  $risolventi$  continene la clausola vuota then
8:         return true
9:        $nuove \leftarrow nuove \cup risolventi$ 
10:  if  $nuove \subseteq clausole$  then
11:    return false
12:   $clausole \leftarrow clausole \cup nuove$ 

```

29/10/2020

10.1 Completezza di TT-Resolution

In questa sezione si dimostra la completezza dell'algoritmo di inferenza basato su risoluzione presentato in chiusura della scorsa lezione. Introduciamo dapprima un concetto preliminare:

Definizione 10.1 (Chiusura per risoluzione). Sia S un insieme di clausole, allora chiamiamo **chiusura per risoluzione** di S , indicato come $RC(S)$, l'insieme di tutte le clausole derivabili dall'applicazione ripetuta della regola di risoluzione alle clausole in S e le sue derivate.

Se S è un insieme finito di clausole, P_1, P_2, \dots, P_k , allora è facile convincersi che $RC(S)$ debba essere a sua volta un insieme finito, infatti esiste solo un numero limitato di clausole che si possono costruire con questi simboli.

L'algoritmo TT-ENTAILS parte da un insieme di clausole "*clause*" che rappresenta la forma normale congiuntiva della formula $KB \wedge \neg\alpha$ e a partire da questa, applicando la regola di risoluzione ripetutamente (una o più volte), costruisce gradualmente la chiusura per risoluzione dell'insieme delle clausole in $KB \wedge \neg\alpha$, che alla fine dell'esecuzione sarà contenuta in *clause*. La finitezza di $RC(S)$ ci garantisce che l'algoritmo termini: se si è costruito tutto $RC(S)$ non si potrà fare altro che generare clausole già presenti in *clause* e quindi il test della riga 10 porterebbe a terminare l'esecuzione, oppure in alternativa potremmo terminare prima a causa del test della riga 7. Tuttavia la finitezza di $RC(S)$ non garantisce da sola che l'algoritmo sia corretto.

Teorema 10.1.1 (Teorema di Ground Resolution). Se un insieme di clausole S è insoddisfacibile allora $\{ \} \in RC(S)$, i.e. la clausola vuota sta nella sua chiusura per risoluzione.

Dimostrazione. Si procede sfruttando la regola di contrapposizione sull'asserto del teorema: $(a) \implies (b) \equiv \neg(b) \implies \neg(a)$. Si supponga che la chiusura per risoluzione di un insieme di clausole S non contenga la clausola vuota, allora vogliamo dimostrare che $RC(S)$ è soddisfacibile, ossia esiste una assegnazione ai simboli proposizionali in S , i.e. P_1, P_2, \dots, P_k , che renda vere tutte le clausole. A tale fine si propone la seguente procedura:

```
1: for each  $i \in \{1, 2, \dots, k\}$  do
2:   if esiste una clausola che contiene il simbolo  $\neg P_i$  and
      questa è falsa rispetto all'assegnazione di  $P_1, P_2, \dots, P_{i-1}$  then
3:      $P_i \leftarrow false$ 
4:   else
5:      $P_i \leftarrow true$ 
```


Facciamo vedere che tale procedura fornisce un modello per $RC(S)$. Supponiamo per assurdo che ciò non accada, ovvero esiste un passo i tale che a questo passo la procedura dia luogo alla prima clausola falsa. Affinché ciò sia possibile occorre che tale clausola, essendo disgiunzione di simboli proposizionali, sia in una di queste due forme:

$$(i) \text{ false} \vee \text{false} \vee \cdots \vee \neg P_i$$

$$(ii) \text{ false} \vee \text{false} \vee \cdots \vee P_i$$

Se siamo nel caso (i) allora la condizione dell'**if** a riga 2 della procedura è soddisfatta e si assegna $P_i \leftarrow \text{false}$, il che rende vera la clausola. Se invece siamo nel secondo caso, la condizione non è soddisfatta e viene eseguito il contenuto dell'**else** di riga 4, pertanto si assegna $P_i \leftarrow \text{true}$ rendendo ancora una volta vera la clausola.

Se in $RC(S)$ compare una sola delle due clausole, cioè solo (i) o solo (ii), siamo già giunti ad un assurdo in quanto si contraddice l'ipotesi che i sia il primo passo a cui compare una clausola falsa. Rimane invece da discutere cosa succede se sia (i) che (ii) sono in $RC(S)$. In questo caso, visto che la chiusura per risoluzione è chiusa rispetto all'applicazione della regola di risoluzione ai suoi elementi¹ si può pensare di applicare la regola ad (i) e (ii), che danno come risolvente $\text{false} \vee \text{false} \vee \cdots \vee \text{false}$. Questo è assurdo: infatti $RC(S)$ conterrebbe una clausola falsa già prima dell' i -esimo passaggio, contro quanto ipotizzato.

Segue quindi che se $RC(S)$ non contiene la clausola vuota² si può costruire un modello per esso (e pertanto anche di S , in quanto $S \subseteq RC(S)$). ■

Questo teorema garantisce la correttezza dell'algoritmo; se $\text{KB} \wedge \neg\alpha$ è insoddisfacibile allora $\text{KB} \models \alpha$ e adesso abbiamo la garanzia che se questo succede, dobbiamo trovare nella chiusura per risoluzione di S anche la clausola vuota, i.e. prima o poi è soddisfatta la condizione della riga 7 dell'algoritmo.

Esempio 10.1: Incompleta generatività di risoluzione

Si noti che l'insieme S deve essere costruito su $\text{KB} \models \alpha$ e non sul solo KB . Uno potrebbe infatti avanzare la seguente proposta di algoritmo per decidere se $\text{KB} \models \alpha$: si genera tutto $RC(S)$ finché applicando la risoluzione a due clausole non si ottiene proprio la query α . Mostriamo qui con un controesempio che un tale algoritmo non potrebbe funzionare.

Prendiamo come base di conoscenza la formula $A \wedge B$, e ci chiediamo se, detta $\alpha = A \vee B$, vale la relazione di entailment $\text{KB} \models \alpha$ (La risposta è senz'altro affermativa: tutte le interpretazioni che rendono vero $A \wedge B$ rendono vero anche $A \vee B$). Vogliamo rispondere usando la proposta avanzata in apertura dell'esempio. Qui le clausole in KB sono $\{A, B\}$, e nessuna di queste contiene letterali complementari, pertanto non è applicabile la risoluzione; con questa strategia non si produce niente, tantomeno α . Si dice che la risoluzione è **non generativamente completa**. ■

¹Si intende con ciò che applicando la regola di risoluzione a due elementi in $RC(S)$ otteniamo ancora qualcosa che sta in $RC(S)$

²In quel caso sei fregato: è impossibile rendere vera la clausola vuota

10.2 Forward e backward chaining

10.2.1 Clausole definite e clausole di Horn

Ci sono numerosi altri algoritmi che si possono usare per dimostrare teoremi; talvolta si riescono a migliorare le prestazioni semplicemente imponendo qualche vincolo aggiuntivo sul linguaggio, cioè ammettendo solo alcune formule della logica proposizionale piuttosto che tutte. Vediamo di seguito alcune restrizioni che possiamo ammettere.

Definizione 10.2 (Clausola definita). Una clausola si dice essere **definita** se essa contiene esattamente un letterale positivo (i.e. non negato)

Per esempio la clausola $\neg A \vee \neg B \vee C$ è una clausola definita. La particolarità di queste clausole è che queste corrispondono a delle implicazioni in cui la premessa è una congiunzione di letterali positivi mentre la conclusione è un singolo letterale positivo. È facile convincersi che

$$\neg A \vee \neg B \vee C \equiv \neg(A \vee B) \vee C \equiv (A \wedge B) \implies C$$

Dove l'ultima equivalenza logica vale applicando De Morgan e ricordando la definizione di implicazione logica. Oltre a queste vediamo un'altra categoria di clausole:

Definizione 10.3 (Clausola di Horn). Una clausola si dice essere **di Horn** se essa contiene al più un letterale positivo

Una clausola che non contiene letterali positivi può essere utilizzata per esprimere dei vincoli. Ad esempio $\neg A \vee \neg B$, che è una clausola di Horn per definizione, è equivalente alla formula $A \wedge B \implies \text{False}$, e quindi questa è una affermazione che esprime il fatto che A e B non possano essere veri insieme.

Per quanto riguarda quest'ultimo tipo di clausole, vedremo che l'inferenza può essere effettuata attraverso algoritmi di **forward chaining** e **backward chaining**, ed inoltre decidere se vale l'entailment con clausole di Horn richiede un tempo lineare rispetto alla dimensione della base di conoscenza. Infine si può dimostrare che applicando la risoluzione tra due clausole di Horn, il risolvete è a sua volta una clausola di Horn.

Esempio 10.2: Chiusura delle clausole di Horn rispetto a risoluzione

Si considerino le due seguenti formule:

$$A \Leftarrow B \wedge C$$

$$C \Leftarrow D \wedge E$$

La prima può essere espressa esplicitando la definizione di implicazione come $\neg B \vee \neg C \vee A$, mentre la seconda come $\neg D \vee \neg E \vee C$, e queste sono entrambe clausole di Horn. Applichiamo la risoluzione tra le due clausole: elidendo i letterali complementari

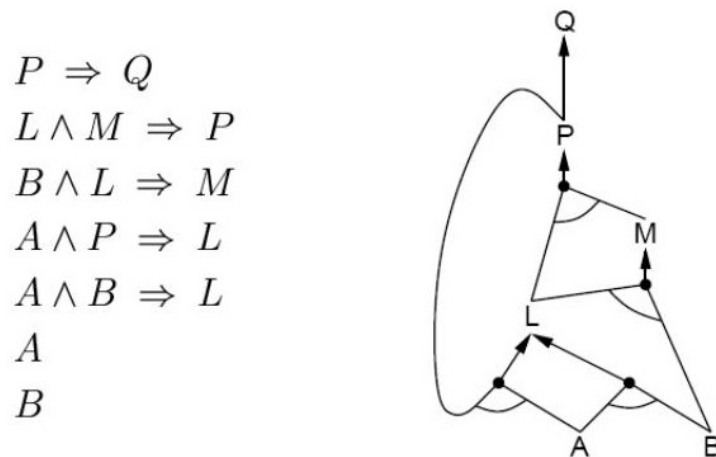
$$\frac{\neg B \vee \neg C \vee A \vee \neg D \vee \neg E \vee C}{\neg B \vee A \vee \neg D \vee \neg E}$$

Il risolvete, che è una clausola di Horn, può essere riscritto sotto forma di implicazione come $A \Leftarrow B \wedge D \wedge E$; tale risultato è anche intuitivo, infatti per dimostrare A dobbiamo dimostrare B e C , ma per dimostrare C dobbiamo dimostrare D ed E , e pertanto in definitiva per dimostrare A dobbiamo dimostrare B, D ed E . ■

10.2.2 Algoritmi di inferenza per clausole di Horn

In questa sezione presentiamo l'algoritmo di **forward chaining**. Taluno è un algoritmo per decidere se vale l'entailment tra una base di conoscenza ed una formula assegnata, e procede secondo l'approccio bottom-up: a partire dai fatti noti, cioè da formule contenute nella base di conoscenza che si sanno essere vere, esso dimostra quante più formule possibili.

Una intuizione grafica di come opera questo algoritmo si può ottenere rappresentando la base di conoscenza attraverso un **grafo and-or**. Supponiamo che la base contenga una formula del tipo $L \wedge M \Rightarrow P$, allora in tale grafo ci saranno due archi: uno che parte da L ed uno che parte da M , e questi si ricongiungeranno in un arco che punta verso P . I due archi saranno anche evidenziati da un "angolo" tra i due, che nell'ambito di questi grafi significa "congiunzione". Riportiamo di seguito la figura di una base di conoscenza e la rispettiva rappresentazione su grafo



L'algoritmo mantiene una tabella di contatori "*conteggio*", uno per ciascuna formula, che conta il numero di simboli proposizionali nella premessa. Inoltre viene mantenuta una seconda tabella "*inferito*" che inizialmente contiene *false* per ogni simbolo. Infine viene mantenuta una coda "*agenda*" che contiene inizialmente i simboli proposizionali che sono noti come fatti (nell'esempio $agenda = \{A, B\}$).

Algoritmo 19 Algoritmo di forward chaining per clausole di Horn

```

1: function FORWARD-CHAINING( $KB, \alpha$ )
2:   while  $agenda$  non è vuota do
3:      $p \leftarrow POP(agenda)$ 
4:     if  $p = \alpha$  then
5:       return true
6:     if  $inferito[p] = false$  then
7:        $inferito[p] \leftarrow true$ 
8:     for all clausola  $c \in KB$  tale che  $p \in c.PREMESSA$  do
9:       decrementa  $conteggio[c]$ 
10:      if  $conteggio[c] = 0$  then
11:        aggiungi  $c.CONCLUSIONE$  all' $agenda$ 
12:   return false

```

Ad ogni passo l'algoritmo estrae un elemento dall'*agenda*, che contiene tutte le formule che abbiamo dimostrato essere vere. Se viene estratta la query α allora $\text{KB} \models \alpha$ e l'algoritmo restituisce *true*, altrimenti se si è estratto un generico elemento p , se già si sapeva essere vero non si fa niente, altrimenti si decrementa il contatore *conteggio* per ciascuna clausola che ha p come premessa; se una di queste clausole raggiunge il valore di *conteggio* 0 la sua conclusione viene aggiunta alle formule dimostrate (succede infatti solo quando si è dimostrata la sua premessa). Si procede fintantoché non si estrae α dall'*agenda* oppure non rimangono più formule da dimostrare, nel qual caso $\text{KB} \not\models \alpha$.

Il difetto principale di questo algoritmo è che dimostra un sacco di implicazioni che non sono direttamente collegate a ciò che si doveva dimostrare, con il rischio di perdere molto tempo. L'alternativa, quando si sa già cosa si vuole dimostrare, è ragionare al contrario, lavorando top-down piuttosto che bottom-up. Nasce così **backward chaining**: esso individua nella base di conoscenza una formula che ha la query α come conclusione, e ricorsivamente cerca di dimostrare le sue premesse, finché non si arriva a trovare formule o simboli che sappiamo essere veri.

Con la base di conoscenze dell'esempio precedente, per dimostrare Q (che in quel caso sarebbe la query), occorre dimostrare P per cui si effettua una chiamata a sottoprogramma per cercare di dimostrare P . Nella base proposta c'è una sola clausola che ha P come conclusione, che è $L \wedge M \implies P$, per cui si cerca ricorsivamente di dimostrare L ed M , e via dicendo finché non si incontrano A e B .

Si può dimostrare che con clausole di Horn sia forward chaining che backward chaining sono corretti e completi.

10.3 DPLL: Un algoritmo per SAT

L'algoritmo che vediamo adesso venne proposto nel 1962 da Davis e Putman, e venne migliorato più avanti da Logemann e Loveland, da cui il nome DPLL. Questo è un algoritmo per decidere la soddisfacibilità di un insieme S di clausole che somiglia fortemente al BACKTRACKING visto per i problemi di soddisfacimento di vincoli.

È piuttosto intuitivo infatti che il problema della soddisfacibilità di possa riformulare come un CSP in cui le variabili sono i simboli proposizionali, i domini sono $\{true, false\}$ ed infine i vincoli sono che ciascuna delle clausole dev'essere *true*. Poiché le variabili sono simboli della logica proposizionale si possono usare delle euristiche mirate piuttosto che alcune generiche.

L'algoritmo ha una natura ricorsiva come BACKTRACKING ed usa tre euristiche:

- (i) **Terminazione prematura.** Come criterio di arresto per la ricorsione DPLL è in grado di controllare se una formula sia vera o falsa anche con un modello solo parzialmente completato; ad esempio per far sì che una clausola sia vera è sufficiente che contenga un letterale vero e questo può portare a determinare che la formula sia vera anche senza il modello completo. D'altra parte è sufficiente che una formula in S sia resa falsa dall'assegnazione parziale operata fino a quel punto per poter dire che quel modello non soddisfa S e poter dunque fare retromarcia.

Si immagini che $\text{KB} = \{\{A, B\}, \{A, C\}, \dots\}$, che è la rappresentazione in forma insiemistica di $(A \vee B) \wedge (A \vee C) \wedge \dots$ allora se l'assegnazione parziale fissa $A \leftarrow true$ possiamo già dire, anche senza fissare B o C , che $\{A, B\}$ ed $\{A, C\}$ sono entrambe

rese vere. Invece, se il modello fissa $A \leftarrow false$ e $B \leftarrow false$ la clausola $\{A, B\}$ è resa falsa ed il modello non può quindi soddisfare S ; si può fare marcia indietro.

- (ii) **Euristica del simbolo puro.** Un simbolo si dice essere puro se compare sempre come letterale positivo o sempre come letterale negativo. Se una formula ammette un modello, allora ne ha anche uno in cui i simboli puri sono assegnati ad un valore di verità che rende i corrispondenti letterali veri e questa osservazione ci permette di decidere immediatamente il valore da assegnare a loro. Nello stabilire se un simbolo sia vero o meno l'algoritmo può non tenere conto delle clausole che già si sanno essere vere.

Se $KB = \{\{A, \neg B\}, \{\neg B, \neg C\}, \{C, A\}\}$ allora A e B sono simboli puri in quanto A compare sempre positivo e B sempre negativo, ma C non è puro. In questo caso si può assegnare $A \leftarrow true$ e $B \leftarrow false$, ed è facile convincersi che fare questo non possa rendere in alcun caso falsa nessuna clausola.

- (iii) **Clausole unitarie.** Una clausola si dice essere unitaria se contiene un solo letterale (durante l'esecuzione di DPLL si può considerare come tale una clausola che contiene un solo letterale *non ancora assegnato*). Se sono presenti allora c'è un'unica possibile assegnazione che possa renderla vera, ad esempio se KB contiene la clausola $\neg C$ è ovvio che per soddisfare S devo assegnare $C \leftarrow false$. Inoltre assegnare un valore ad un simbolo può generare delle clausole unitarie: se $KB = \{\{A, \neg B\}, \{\neg B, \neg C\}, \{C, A\}\}$, assegnando $B \leftarrow true^3$ allora KB è logicamente equivalente a $\{\{A\}, \{\neg C\}, \{C, A\}\}$ per cui siamo obbligati ad assegnare $A \leftarrow true$ e $C \leftarrow false$ per soddisfare le due clausole unitarie; fortunatamente questo soddisfa anche tutte le clausole rimanenti. Si parla di **propagazione delle clausole unitarie** (strettamente legato a FORWARD-CHECKING).

Di seguito viene riportato uno pseudocodice per l'implementazione di questo algoritmo. In breve, DPLL-SODDISFACIBILE è una funzione di appoggio che predispone il lavoro per il DPLL vero e proprio. Quest'ultimo, dapprima verifica se il modello costruito fino ad ora soddisfa S , nel qual caso termina, altrimenti prova ad applicare nell'ordine in cui le abbiamo presentate le tre euristiche. Se nessuna di queste risulta applicabile allora non rimane altro che provare tutti i possibili valori per il simbolo P estratto, ossia *true* (c.f.r prima chiamata ricorsiva) oppure *false* (c.f.r seconda chiamata ricorsiva). In altre parole, si dirama e guarda cosa succede nei due sottoalberi.

Algoritmo 20 Algoritmo DPLL per decidere la soddisfacibilità di un insieme di clausole

```

1: function DPLL-SODDISFACIBILE( $s$ )
2:    $clausole \leftarrow$  L'insieme delle clausole nella rappresentazione CNF di  $s$ 
3:    $simboli \leftarrow$  L'insieme di tutti i simboli proposizionali  $s$ 
4:   return DPLL( $clausole, simboli, \{\}$ )
5:
6: function DPLL( $clausole, simboli, modello$ )
7:   if ogni clausola  $c \in clausole$  è vera nel  $modello$  then
8:     return true
```

³Il professore dice “assegnando $B \leftarrow false$ ” ma essendo una clausola, tale assegnazione la renderebbe automaticamente vera, rientrando nella prima euristica (? magari mi sbaglio).

```

9:   if esiste una clausola  $c \in \text{clausole}$  falsa nel modello then ▷ Euristic (i)
10:    return false
11:  ( $P, \text{valore}$ )  $\leftarrow$  TROVA-SIMBOLO-PURO(clausole, simboli, modello) ▷ Euristic (ii)
12:  if  $P \neq \text{null}$  then
13:    return DPLL(clausole,  $\text{simboli} \setminus \{P\}$ ,  $\text{modello} \cup \{P = \text{valore}\}$ )
14:  ( $P, \text{valore}$ )  $\leftarrow$  TROVA-CLAUSOLA-UNITARIA(clausole, modello) ▷ Euristic (iii)
15:  if  $P \neq \text{null}$  then
16:    return DPLL(clausole,  $\text{simboli} \setminus \{P\}$ ,  $\text{modello} \cup \{P = \text{valore}\}$ )
17:   $P \leftarrow$  PRIMO(simboli) ▷ Nessuna euristica applicabile
18:  restanti  $\leftarrow$  RESTANTI(simboli)
19:  return DPLL(clausole, restanti,  $\text{modello} \cup \{P = \text{true}\}$ ) or
    DPLL(clausole, restanti,  $\text{modello} \cup \{P = \text{false}\}$ )

```

10.4 Ricerca locale stocastica per SAT

La ricerca stocastica usa dei cosiddetti “algoritmi Las Vegas”. Un algoritmo è tale se è garantito che sia corretto, ma non necessariamente completo: se trovano un’assegnazione dei simboli proposizionali usando tecniche stocastiche che renda vera la formula, non c’è dubbio che essa sia soddisfacibile. D’altra parte, il fatto di non trovarla in un certo tempo assegnato non garantisce che essa sia insoddisfacibile, magari se fosse proseguito avrebbe trovato un’assegnazione, o forse no.

Questa classe di algoritmi non può essere usata per decidere l’insoddisfacibilità in quanto ai fini di questo taluni non sono completi⁴, e quindi non sono adatti ad utilizzi come la decisione dell’entailment $\text{KB} \models \alpha$ attraverso la dimostrazione che $\text{KB} \wedge \neg \alpha$ non è soddisfacibile.

Uno degli algoritmi più famosi di questa classe è WALK-SAT. Esso accetta come ingresso un insieme di clausole in CNF, una probabilità p ed un intero che rappresenti il numero massimo di step che l’algoritmo può eseguire prima di doversi fermare. Si parte con una assegnazione casuale ai simboli proposizionali; se in questo modo si è prodotto un modello per la formula da soddisfare allora abbiamo concluso, in caso contrario sceglie una clausola a caso tra quelle non soddisfatte e prova ad assegnare a caso uno dei simboli ivi contenuti.

Similmente a SIMULATED-ANNEALING, l’algoritmo ha come obiettivo quello di massimizzare il numero di clausole soddisfatte, ma si accettano anche mosse sfavorevoli come assegnazioni ai simboli proposizionali; in questo contesto, p rappresenta la probabilità di fare una mossa a caso piuttosto di una mirata a qualche forma di ottimizzazione. Nello pseudocodice RANDOM è un generatore casuale di numeri in $[0, 1]$.

Se si generano dei problemi casuali, cioè delle clausole a caso, e si mettono in “and” tra di loro, a seconda di quante clausole si sono generate e quanti simboli proposizionali si sono presi, la probabilità di aver generato un problema soddisfacibile o non soddisfacibile è stranamente molto vicina a 0 o molto vicina ad 1.

Tipicamente si decide di generare problemi in cui le clausole hanno una certa lunghezza fissata $k \geq 3$ (questo perché con $k = 2$ SAT è un problema polinomiale⁵), con un numero

⁴sono però ragionevoli nel decidere la soddisfacibilità

⁵Il lettore provi a convincersene, lavorando con grafi e pensando al 2 come numero di archi.

Algoritmo 21 Algoritmo di ricerca locale stocastica per SAT

```

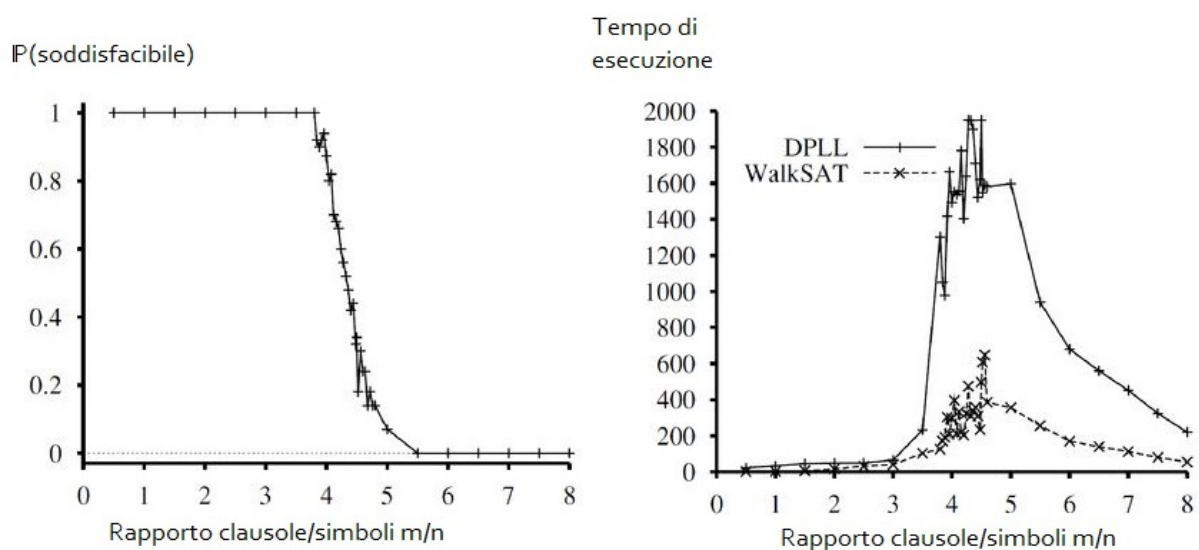
1: function WALK-SAT(clausole, p, max_tentativi)
2:   modello  $\leftarrow$  un'assegnazione a caso dei simboli in clausole
3:   for  $t \in \{1, 2, \dots, \text{max\_tentativi}\}$  do
4:     if modello soddisfa tutte le clausole then
5:       return modello
6:     clausola  $\leftarrow$  una clausola a caso tra quelle non soddisfatte da modello
7:     if RANDOM() > p then
8:       inverti il valore di un simbolo a caso in clausola
9:     else
10:      inverti il valore del simbolo in clausola che massimizza
        il numero di clausole soddisfatte

```

stabilito n di simboli proposizionali. Chiamiamo invece m il numero di diverse clausole generate. La classe dei problemi siffatti viene indicata come $CNF_k(n, m)$ ed il processo generativo funziona in questo modo:

- (i) Seleziona k letterali tra i $2n$ disponibili ($2n$ perché n sono positivi ed n negativi).
- (ii) Se si è prelevato uno stesso simbolo più di una volta, e quindi la clausola generata ha lunghezza minore di k , scarta la clausola.
- (iii) Se la clausola generata è una tautologia allora la si scarta.
- (iv) Se la clausola generata è un duplicato la si scarta.
- (v) Ripeti da capo finché non ottieni m clausole.

Una volta generato un problema si può lanciare DPLL che risponde *true* o *false* in base al fatto che la formula generata sia soddisfacibile o meno. Se generiamo un largo numero di questi problemi ci si può chiedere quanti sono soddisfacibili e quanto non lo sono, e quindi stimare la probabilità che uno di questi problemi sia soddisfacibile.



Il comportamento è quello raffigurato qui sopra ($k = 3, n = 50, m$ variabile): se si hanno numerose clausole intuitivamente il problema è insoddisfacibile perché si hanno troppi

vincoli (ogni clausola è un vincolo: essa deve essere *true*), però questo dipende anche dal numero di simboli perché ciascuno di questi ci dà dei gradi di libertà che possono essere usati per trovare soluzioni. Osservando il grafico sulla sinistra si nota che fino ad un certo valore del rapporto tra clausole e simboli, in figura pari a 4.3, che funge da **punto di transizione di fase**, tutti i problemi generati erano soddisfacibili, e dopo una rapida transizione tutti i problemi generati diventano insoddisfacibili.

Osservando il tempo di esecuzione, grafico sulla destra, si nota che DPLL è velocissimo sui problemi facili (i.e. per cui m/n è piccolo), e tende ad essere abbastanza veloce anche quando il problema è insoddisfacibile, ma nella regione di transizione, intorno a $m/n = 4.3$, si manifesta la complessità computazionale di DPLL e più si è vicini al rapporto limite più il tempo di esecuzione “esplode”.

2/11/2020

11.1 Probabilità e assiomi di Kolmogorov

Sia $(\Omega, \mathcal{E}, \mathbb{P})$ uno spazio di misura, allora diciamo che \mathbb{P} è una **probabilità** se gode di ciascuna di queste proprietà (sia $\omega \in \Omega$ un possibile “mondo”):

- (i) $0 \leq \mathbb{P}(\omega) \leq 1$
- (ii) $\sum_{\omega \in \Omega} \mathbb{P}(\omega) = 1$
- (iii) $\forall A, B \in \mathcal{E} \quad \mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A \cap B)$

Esempio 11.1: Assiomi di Kolmogorov in pratica

Sia R = “Fuori piove” ed S = “Fuori c’è il sole” e si assuma la seguente distribuzione:

	R	S	\mathbb{P}
ω_1	0	0	0.25
ω_2	0	1	0.2
ω_3	1	0	0.5
ω_4	1	1	0.05

Verifichiamo che quella data è effettivamente una probabilità, cioè che sono soddisfatti gli assiomi precedenti. La verifica di (i) e (ii) segue banalmente per ispezione della tabella, mentre per quanto riguarda il terzo, possiamo ricavare le probabilità degli eventi R ed S per marginalizzazione:

$$\mathbb{P}(R) = \omega_3 + \omega_4 = 0.5 + 0.05 = 0.55 \quad (11.1)$$

$$\mathbb{P}(S) = \omega_2 + \omega_4 = 0.2 + 0.05 = 0.25 \quad (11.2)$$

$$\mathbb{P}(R \vee S) = \omega_2 + \omega_3 + \omega_4 = 0.2 + 0.5 + 0.05 = 0.75 \quad (11.3)$$

$$\mathbb{P}(R \wedge S) = \omega_4 = 0.05 \quad (11.4)$$

Usando le (11.1), (11.2), (11.4) si può dimostrare che vale anche (iii), che si chiama *principio di inclusione-esclusione*, infatti:

$$\mathbb{P}(R \vee S) = 0.75 = 0.55 + 0.25 - 0.05 = \mathbb{P}(R) + \mathbb{P}(S) - \mathbb{P}(R \wedge S)$$

Prendendo il primo e l’ultimo membro della catena precedente si conclude che effettivamente siamo di fronte ad una probabilità. ■

Adesso introduciamo una ulteriore nozione che è quella di **probabilità condizionata**, che è uno strumento spendibile per aggiornare le probabilità a priori (*priors*) una volta acquisite nuove informazioni.

Siano assegnati due eventi, $A, B \in \mathcal{E}$, di cui B non impossibile. Allora si pone per definizione la probabilità di A condizionata da B come segue:

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

Ovviamente la condizione di non impossibilità di B serve unicamente a garantire di poter calcolare tale frazione. Quando ben definita questa probabilità, si può invertire la formula per ottenere la cosiddetta “formula di probabilità congiunta”:

$$\mathbb{P}(A \wedge B) = \mathbb{P}(A | B) \cdot \mathbb{P}(B)$$

Questa ci fornisce anche una interpretazione per la formula di probabilità condizionata: la probabilità che due eventi A e B si verifichino insieme si calcola tenendo conto della probabilità che si verifichi B e che si verifichi A sapendo che si è verificato B .

11.2 Beliefs, probabilità soggettive

Classicamente il valore assunto da una probabilità si interpreta come il rapporto tra numero di casi favorevoli e numero di casi totali su un numero sufficientemente largo di ripetizioni di uno stesso esperimento; si parla in questo caso di **approccio frequentista**. Con riferimento all'esempio precedente, se diciamo che $\mathbb{P}(S) = 0.25$ si può pensare che il rapporto tra numero di giorni soleggiati ed il totale di quelli considerati è proprio pari a 0.25 (ad esempio a partire da quando vengono registrate le informazioni meteo).

Questa idea, sebbene sembri funzionare, è difficile da applicare ad alcune situazioni che uno potrebbe voler considerare; rendiamocene conto con un esempio: consideriamo la seguente uguaglianza:

$$\mathbb{P}(\text{Joe Biden vincerà le prossime elezioni}) = 0.7$$

D'altra parte però è difficile pensare di aver ripetuto qualche centinaio di volte le elezioni negli USA e 7 volte su 10 ha vinto Biden. Tale numero è più sensato associarlo ad una **credenza** (*belief*); ognuno di noi di fatto è incline a pensare che sia ragionevole esprimere una certa probabilità che un dato evento si verifichi. Ovviamente quando vogliamo lavorare con delle credenze sorge spontaneo un problema: come si può stabilire se una di queste è utilizzabile per effettuare delle operazioni rigorose?

Vedremo di seguito i tentativi di tre matematici di dare una impostazione formale ad un sistema fatto di credenze.

11.2.1 Assiomi di Cox e Jaynes

Se x è un simbolo proposizionale ed I è l'insieme delle informazioni in nostro possesso, quanto si crede in x sulla base delle informazioni I è espresso da una opportuna funzione $\text{Bel}(x | I)$. Taluna deve soddisfare il seguente insieme di assiomi:

- (i) Se $\text{Bel}(x | I) > \text{Bel}(y | I)$ e $\text{Bel}(y | I) > \text{Bel}(z | I)$ allora $\text{Bel}(x | I) > \text{Bel}(z | I)$
- (ii) Esiste una funzione F tale che $\text{Bel}(x | I) = F(\text{Bel}(\neg x | I))$
- (iii) Esiste una funzione G che soddisfa la seguente equazione:

$$\text{Bel}(x \wedge y | I) = G(\text{Bel}(x | I), \text{Bel}(y | x \wedge I))$$

La (i) è una semplice ed intuitiva proprietà transitiva; la (ii) suggerisce che dobbiamo essere in grado di esprimere quanto si crede in x in termini di quanto si crede in $\neg x$; in tal senso si osservi che non abbiamo nessun vincolo di valori ed una credenza non è forzata ad essere nell'intervallo $[0, 1]$; se avessimo avuto una probabilità si avrebbe avuto che $F(p) = 1 - p$. Infine, la (iii) ci suggerisce che dovremmo essere in grado di esprimere quanto si crede in $x \wedge y$ in termini di quanto si crede in x e di quanto si crede in y se aggiungiamo x ai fatti noti.

Si può dimostrare, e non lo faremo, che a meno di un fattore di normalizzazione le credenze si comportano come probabilità, ossia obbediscono agli assiomi di Kolmogorov di inizio sezione.

11.2.2 Credenze secondo De Finetti

Il matematico italiano De Finetti definisce la credenza in un evento x come la somma che una persona sarebbe disposta ad esborsare per pagare un biglietto della lotteria la cui vincita è legata al verificarsi dell'evento x .

Esempio 11.2: Intuizione sulla definizione

Un biglietto della lotteria viene venduto; i bookmakers stabiliscono che se Biden vincessi le elezioni allora il suo possessore verrebbe pagato 1€, mentre in caso di sconfitta il biglietto non porta nessun incasso. Ci si chiede: a quanto una persona sarebbe disposta a vendere tale biglietto? Ovviamente chiunque sia ragionevole accetterebbe di venderlo per 1€ o più in quanto tale somma coinciderebbe o supererebbe la potenziale vincita. D'altra parte, quanto sarebbe disposta una persona a pagare tale biglietto? Sicuramente per 0€ chiunque, si fronte alla potenziale vincita e assenza di rischi, accetterebbe di acquistarlo.

In questo contesto, definiamo la credenza in x come il prezzo per il quale una persona sarebbe ugualmente disposta a vendere e a comprare. ■

Scommessa olandese - legame De Finetti - probabilità

Si considera una corsa di cavalli e gli eventi $A :=$ "Il cavallo A si qualifica", $B :=$ "Il cavallo B si qualifica". Il bookmaker vende i seguenti quattro biglietti (riportiamo sotto ciascuno il costo e supponiamo che la vincita paghi 1€):

A si qualifica	B si qualifica	Né A né B si qualificano	Solo uno tra A e B si qualifica
0.60	0.50	0.25	0.70

Supponiamo inoltre di comprare 100 biglietti di ciascun tipo; allora si ottiene la seguente rappresentata in tabella; qui assumiamo che *True* corrisponda al fatto che l'evento indicato si sia verificato e *False* al fatto che l'evento non si sia verificato. Le celle interne

della tabella rappresentano la vincita relativa alla riga in cui si trovano.

A	B	Biglietto 1	Biglietto 2	Biglietto 3	Biglietto 4	Totale
True	True	+40	+50	-25	-10	-5
True	False	+40	-50	-25	+30	-5
False	True	-60	+50	-25	+30	-5
False	False	-60	-50	+75	+30	-5

Questa tabella rende evidente che qualsiasi mondo si realizzi lo scommettitore perde sempre 5€ ed il bookmaker ci guadagna sempre; in questo caso si dice che le credenze sono **inconsistenti**. De Finetti riuscì a dimostrare che questo avviene se e solo se le credenze non sono delle probabilità, ovvero non obbediscono agli assiomi di Kolmogorov.

Portando avanti l'esempio, si osservi che

$$\mathbb{P}(A) = 0.6 \quad (11.5)$$

$$\mathbb{P}(B) = 0.5 \quad (11.6)$$

$$\mathbb{P}(A \vee B) = 1 - \mathbb{P}(\neg(A \vee B)) = 1 - \mathbb{P}(\neg A \wedge \neg B) = 1 - 0.25 = 0.75 \quad (11.7)$$

$$\mathbb{P}(A \wedge B) = 1 - \mathbb{P}(\neg(A \wedge B)) = 1 - \mathbb{P}(\neg A \vee \neg B) = 1 - 0.7 = 0.3 \quad (11.8)$$

Si può quindi verificare in particolare che i numeri dati non soddisfano il principio di inclusione-esclusione, infatti

$$0.75 = \mathbb{P}(A \vee B) \neq \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A \wedge B) = 0.7$$

Ed in particolare si osserva che -5 , vincita prospettata del bookmaker, corrisponde alla differenza tra il valore vero di $\mathbb{P}(A \vee B)$ e quello calcolato tramite l'applicazione dell'assioma (iii) delle probabilità, espresso in percentuale. Questo risultato riconcilia il mondo della probabilità soggettiva con quello della probabilità frequentista.

11.3 Ragionamento probabilistico secondo Bayes

Usando la definizione di probabilità condizionata insieme al fatto che se A e B sono due eventi, $A \wedge B \equiv B \wedge A$ si ottiene un risultato fondamentale per il ragionamento probabilistico che è la formula di Bayes:

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

Di seguito vediamo una carrellata di risultati utili nel mondo del ragionamento probabilistico. Il primo riguarda le probabilità condizionate, ed è il seguente: se A, B, C sono tre eventi tali che C sia non impossibile e B e C non siano incompatibili, allora vale la seguente uguaglianza:

$$\mathbb{P}(A \wedge B | C) = \mathbb{P}(A | B \wedge C) \cdot \mathbb{P}(B | C) \quad (11.9)$$

Una dimostrazione formale di questo fatto può essere trovata in [PDR20] al capitolo 10.

Un'altro risultato rilevante è la **formula di probabilità totale**: sia Ω uno spazio di eventualità ed $A, B: \Omega \mapsto \mathbb{R}$ due variabili aleatorie ivi definite; allora per ogni possibile realizzazione di A vale la seguente relazione:

$$\mathbb{P}(A) = \sum_B \mathbb{P}(A \wedge B)$$

La precedente è una notazione compatta: con “somma su B ” si intende in realtà “somma su $b \in \text{Dom}(B)$ ”, e per esteso avremmo dovuto scrivere che

$$\forall A \in \text{Dom}(A) \quad \mathbb{P}(A = a) = \sum_{b \in \text{Dom}(B)} \mathbb{P}(A = a \wedge B = b)$$

Infine, se si sa qualcosa su una variabile B e si vuole ragionare su A , il modo ovvio per introdurre B è applicare la **marginalizzazione**, ossia calcolare $\mathbb{P}(A)$ come segue:

$$\mathbb{P}(A) = \sum_B \mathbb{P}(A | B) \cdot \mathbb{P}(B)$$

Conclusa la carrellata, ritorniamo sulla formula di Bayes. Osservando taluna si osserva che è della forma seguente:

$$\mathbb{P}(A | B) \propto \underbrace{\mathbb{P}(B | A)}_{\text{Likelihood}} \cdot \underbrace{\mathbb{P}(A)}_{\text{Prior}}$$

Dove con **prior** si indica la probabilità a priori che si verifichi l'evento in questione, in assenza di ogni altra informazione. Il termine $\mathbb{P}(B|A)$ è chiamato “simiglianza” ed esprime quanto sarebbe probabile B se mettessimo A tra le informazioni note. La probabilità aggiornata talvolta viene chiamata “probabilità a posteriori” (*posterior*). Preferiremo spesso questo modo di scrivere la formula di Bayes; di fatto il suo denominatore è solo un termine di normalizzazione che serve a garantire che $\mathbb{P}(A | B) + \mathbb{P}(\neg A | B) = 1$, mentre non ha una interpretazione rilevante. Anche nello scrivere algoritmi, talvolta tale fattore viene ignorato e riaggiunto soltanto alla fine.

La formula di Bayes può essere facilmente estesa a tre variabili nel modo seguente:

$$\mathbb{P}(A | B \wedge C) = \frac{\overbrace{\mathbb{P}(B | A \wedge C)}^{\text{Likelihood}} \cdot \overbrace{\mathbb{P}(A | C)}^{\text{Vecchio posterior}}}{\mathbb{P}(B | C)}$$

Assumendo che B sia l'informazione nuova ed A, C quelle già note. Questa regola si può anche estendere ad un numero arbitrario di variabili: ogni volta che ne osserviamo una nuova, il nuovo posterior sarà direttamente proporzionale al prodotto tra il precedente posterior e la verosimiglianza della nuova variabile osservata rispetto alle informazioni precedentemente note.

11.3.1 Indipendenze marginali e condizionali

Si osservi che al crescere del numero di eventi al termine condizionante cresce in maniera esponenziale anche il numero di possibili mondi per quella probabilità condizionata. Fortunatamente ci vengono in soccorso i concetti di indipendenza marginale ed indipendenza condizionale:

Definizione 11.1 (Indipendenza Marginale). Siano $A, B: \Omega \mapsto \mathbb{R}$ due variabili aleatorie; si dice che A è **marginalmente indipendente** da B se

$$\mathbb{P}(A = a \mid B = b) = \mathbb{P}(A = a) \quad \forall b \in \text{Dom}(B) \text{ t.c. } \mathbb{P}(B = b) > 0$$

All'atto pratico, dire che A è marginalmente indipendente da B significa affermare che B non è rilevante per predire A .

Definizione 11.2 (Indipendenza Condizionale). Siano $A, B, C: \Omega \mapsto \mathbb{R}$ tre variabili aleatorie; si dice che A è **condizionalmente indipendente** da B dato C , e si scrive che $A \perp B \mid C$, se

$$\mathbb{P}(A = a \mid B = b \wedge C = c) = \mathbb{P}(A = a \mid C = c) \quad \forall b, c \text{ valido}$$

Il significato dell'indipendenza condizionale è che B diventa irrilevante nel predire A se si conosce C .

Normalmente si vuole calcolare una probabilità della forma $\mathbb{P}(\text{ipotesi} \mid \text{tesi})$, e poiché le tabelle delle distribuzioni crescono esponenzialmente, se nel tempo arrivano le informazioni e_1, e_2, \dots, e_{t-1} ed al tempo t arriva l'informazione e_t , la calcoliamo come segue:

$$\mathbb{P}(h \mid e_1, e_2, \dots, e_t) \propto \mathbb{P}(e_t \mid h, e_1, e_2, \dots, e_{t-1}) \cdot \mathbb{P}(h \mid e_1, e_2, \dots, e_{t-1})$$

Cioè tenendo conto di quanto è ragionevole la nuova evidenza rispetto a quanto osservato fin'ora e della conoscenza pregressa (secondo fattore).

Esempio 11.3: Vantaggio dell'indipendenza condizionale

14	24	34	44
13	23	33	43
¹² ¬B	22	32	42
¹¹ OK	²¹ B	31	41

Consideriamo la seguente base di conoscenze: $\text{KB} = \{b, k\}$, dove si pone

$$b = \neg b_{11} \wedge b_{12} \wedge b_{21}$$

$$k = \neg p_{11} \wedge \neg p_{12} \wedge \neg p_{21}$$

Dove il simbolo b_{ij} indica la presenza di un venticello nella cella (i, j) e p_{ij} indica la presenza di una fossa nella cella (i, j) . Chiamiamo inoltre “frontiera” l'insieme $f = \{(2, 2), (3, 1)\}$ e “resto” l'insieme r di tutte le celle che non rientrano nella frontiera e sono vuote nella figura precedente. Si vuole stabilire la probabilità p che sia presente una fossa nella cella $(1, 3)$. Possiamo affermare che

$$p := \mathbb{P}(p_{13} \mid b \wedge k) = \sum_f \sum_r \mathbb{P}(p_{13} \wedge f \wedge r \mid b \wedge k)$$

Nella precedente equazione la sommatoria su f corrisponde a 4 sommatorie diverse mentre la sommatoria su r corrisponde a 1024 sommatorie diverse per un totale di

$1024 \cdot 4 = 4096$ sommatorie al variare dei possibili mondi. Applicando la formula di Bayes a questa si trova che

$$p = \sum_f \sum_r \mathbb{P}(b \mid p_{13} \wedge f \wedge r \wedge k) \cdot \mathbb{P}(p_{13} \wedge f \wedge r \wedge k)$$

Data la conoscenza sulla presenza di una buca in $(1, 3)$, la frontiera f e la base KB, b è condizionalmente indipendente da tutto il resto, dunque possiamo eliminare l' r dal primo fattore; inoltre si può marginalizzare rispetto ad r il risultato, da cui

$$\begin{aligned} p &\propto \sum_f \mathbb{P}(b \mid p_{13} \wedge f \wedge k) \cdot \sum_r \mathbb{P}(p_{13} \wedge f \wedge r \wedge k) \\ &= \sum_f \mathbb{P}(b \mid p_{13} \wedge f \wedge k) \cdot \mathbb{P}(p_{13} \wedge f \wedge k) \end{aligned}$$

Grazie all'indipendenza condizionale siamo riusciti a passare dunque da 4096 sommatorie distinte a sole 4. Inoltre, ipotizzando l'indipendenza tra p_{13} , f , k si può riscrivere in definitiva p nel modo seguente:

$$p \propto \left[\sum_f \mathbb{P}(b \mid p_{13} \wedge f \wedge k) \cdot \mathbb{P}(f) \right] \cdot \mathbb{P}(p_{13})\mathbb{P}(k)$$



5/11/2020

12.1 Modelli grafici orientati: Reti di Bayes

È dato un universo $U = \{X_1, X_2, \dots, X_n\}$ di variabili aleatorie e ciascuna di queste avrà una propria distribuzione. Chiamiamo **evidenza** un insieme $E \subseteq U$ e **query** un ulteriore sottoinsieme $Q \subseteq U$ e tipicamente si vorrà calcolare $\mathbb{P}(Q | E)$. Ovviamente si può almeno in teoria procedere per definizione e calcolarlo nel modo seguente per marginalizzazione¹:

$$\mathbb{P}(Q | E) = \frac{\mathbb{P}(Q \cap E)}{\mathbb{P}(E)} = \frac{\sum_{U \setminus (Q \cup E)} \mathbb{P}(U)}{\sum_{U \setminus E} \mathbb{P}(U)}$$

Tuttavia il numero di termini in questa formula può essere estremamente grande anche in casi semplici, rendendo il problema seriamente intrattabile. Ad esempio se U fosse composto da $n = 20$ variabili tutte di Bernoulli, E da 5 variabili e Q fosse una singola variabile allora il numeratore consisterebbe in $2^{20-6} = 2^{14}$ termini, mentre il denominatore in $2^{20-5} = 2^{15}$ termini. Come se non bastasse, per memorizzare $\mathbb{P}(U)$ serve una memoria che è un $O(d^n)$ se X_1, \dots, X_n sono variabili aleatorie categoriche con domini di dimensione al massimo d , infatti $\mathbb{P}(U)$ contiene un elemento per ogni possibile configurazione (x_1, \dots, x_n) e le diverse possibili sono un $O(d^n)$. Infine, anche il problema di decidere il valore dei parametri è altrettanto rilevante.

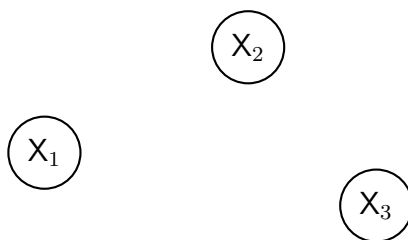
+ Di seguito introduciamo degli strumenti grafici per rappresentare le indipendenze condizionali di un universo di variabili U e vedremo come queste ci possono aiutare ad ottenere semplificazioni per le problematiche appena illustrate; si parla di **modelli grafici probabilistici**. Di questi ne esistono due classi principali, che sono quelli orientati e quelli non orientati, e almeno in questa lezione tratteremo i primi.

Definizione 12.1 (Rete di indipendenza condizionale). Chiamiamo **rete di indipendenza condizionale** un grafo \mathcal{G} aciclico orientato che definisce una famiglia di distribuzioni caratterizzate dal verificare la seguente uguaglianza:

$$\mathbb{P}(X_1, \dots, X_n) = \prod_{i=1}^n \mathbb{P}(X_i | \text{PARENTS}(X_i));$$

Dove in \mathcal{G} i nodi corrispondono alle variabili aleatorie e gli archi rappresentano delle forme di dipendenza/indipendenza tra le due collegate. Più frequentemente tali grafi vengono chiamati anche **reti di Bayes**.

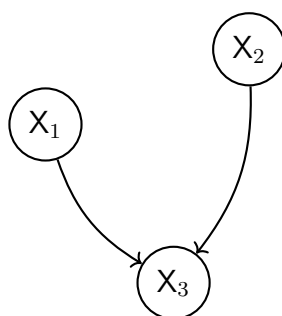
¹Nota che $\mathbb{P}(Q \cap E)$ è una tabella di probabilità definita sull'unione delle variabili in Q ed E .

Esempio 12.1: Fattorizzazione definita da rete di Bayes

La precedente rete di indipendenza condizionale corrisponde seguente fattorizzazione

$$\mathbb{P}(X_1, X_2, X_3) = \mathbb{P}(X_1) \cdot \mathbb{P}(X_2) \cdot \mathbb{P}(X_3)$$

Ovvero rappresenta tre variabili aleatorie che sono tutte indipendenti l'una dall'altra. ■

Esempio 12.2: Fattorizzazione definita da rete di Bayes

La precedente rete di indipendenza condizionale corrisponde alla fattorizzazione

$$\mathbb{P}(X_1, X_2, X_3) = \mathbb{P}(X_1) \cdot \mathbb{P}(X_2) \cdot \mathbb{P}(X_3 | X_1, X_2)$$

■

Adesso riportiamo una rete di indipendenza condizionale che useremo come appoggio per fare vari esempi, leggermente più articolata delle due precedentemente viste. Taluna dovrebbe modellare la situazione seguente: una persona vive in un quartiere in cui sono frequenti episodi malavitosi come furti e si verificano sporadicamente dei terremoti. Per questo motivo, chiede ai vicini di casa di chiamarlo al telefono quando scatta l'allarme, che può essere attivato sia da un furto in casa che dal terremoto. Inoltre la persona in questione ascolta una radio che lo aiuta a tenersi informato sui terremoti che si sono verificati in zona. Il nostro universo sarà composto dalle variabili aleatorie A, B, C, E, R che sono tutte bernoulliane. Si assume la seguente rete di indipendenze: Ovviamente, ignorando momentaneamente la fattorizzazione indotta dal grafo riportato qui sopra, un modo per riscrivere la probabilità congiunta di A, B, C, E, R è la seguente:

$$\mathbb{P}(C, A, B, R, E) = \mathbb{P}(C | A, B, R, E) \cdot \mathbb{P}(A | B, R, E) \cdot \mathbb{P}(B | R, E) \cdot \mathbb{P}(R | E) \cdot \mathbb{P}(E)$$

Ci chiediamo se esistano delle distribuzioni che non verificano una simile fattorizzazione. In effetti questa è stata ottenuta applicando iterativamente la proprietà $\mathbb{P}(E, G) = \mathbb{P}(E|G) \cdot \mathbb{P}(G)$, nota come **formula di probabilità composta**, che vale indipendentemente dalle

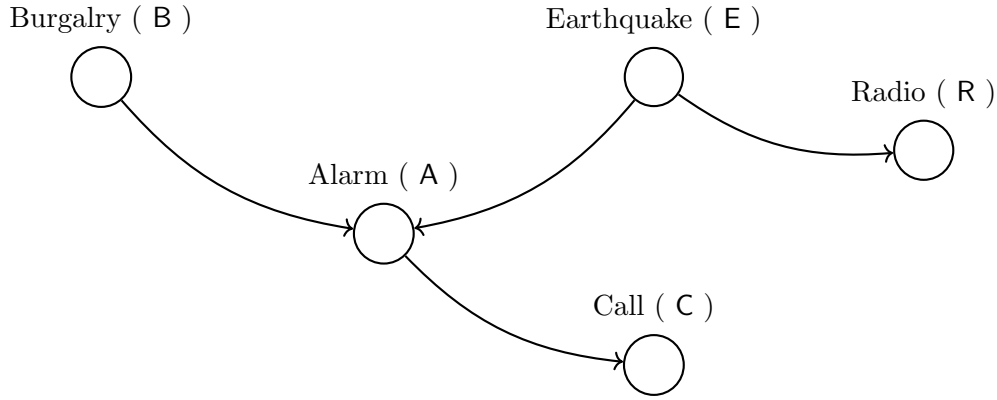
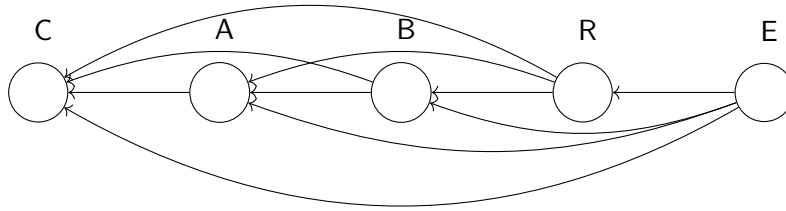


Figura 12.1: Rete di Bayes da usare come esempio di riferimento

distribuzioni delle variabili aleatorie coinvolte, quindi la risposta alla nostra domanda è negativa. Il grafo completo associato a questa distribuzione sarebbe il seguente:



Gli archi mancanti, rispetto a questo grafo, nella rete di indipendenze dell'immagine 12.1 rappresentano quindi delle indipendenze condizionali.

In questo caso il “gioco” ha funzionato bene, cioè ha portato a delle semplificazioni, perché siamo partiti da un ordinamento adatto dei nodi e non uno a caso dei $5!$ possibili; in particolare si è scelto un **ordinamento topologico**, ovvero uno in cui ciascun nodo compare nella sequenza ordinata dopo tutti i suoi genitori. La fattorizzazione che si ottiene è la seguente:

$$\mathbb{P}(C, A, B, R, E) = \mathbb{P}(C | A) \cdot \mathbb{P}(A | B, E) \cdot \mathbb{P}(B) \cdot \mathbb{P}(E) \cdot \mathbb{P}(R | E)$$

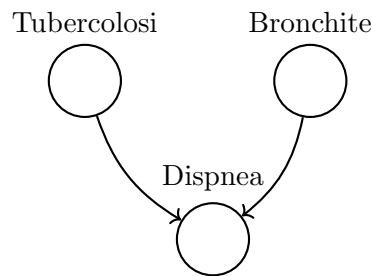
A differenza della precedente, questa relazione *non* è verificata da una distribuzione qualsiasi ma da soltanto quelle che realizzano queste indipendenze condizionali.

Soffermiamoci adesso sulla questione parametri: sempre con riferimento al modello proposto come esempio, usando la rete completa avremmo dovuto assegnare $2^5 - 1$ parametri, dove il -1 è dovuto al fatto che se conosciamo 31 parametri possiamo ricavare il rimanente imponendo che la somma delle probabilità faccia 1. La fattorizzazione indotta dalla rete di indipendenza condizionale invece porta a necessitare di soli 10 parametri piuttosto che 31. Il numero di parametri nel caso di variabili bernoulliane si calcola come

$$N_p = \sum_{i=1}^n 2^{\#\{\text{Genitori di } X_i\}}$$

Possiamo proporre un breve esempio per visualizzare meglio cosa si intende per “parametri”. Data una persona vogliamo predire la probabilità che questa soffra di dispnea

(difficoltà respiratorie) sapendo se soffre di bronchite e/o tubercolosi o nessuna di queste. Assumiamo la seguente rete di indipendenze condizionali:

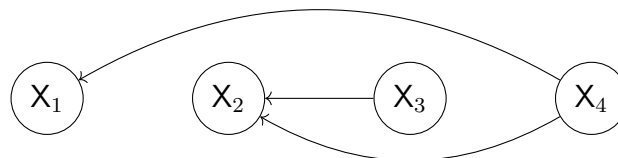


Ed anche la seguente tabella di probabilità condizionale; i valori inseriti sono una possibile assegnazione dei parametri ma non l'unica.

Probabilità di dispnea				
Tubercolosi	Bronchite		Dispnea	
	Sì	No	Sì	No
Sì	0.9	0.8	0.7	0.1
No	0.1	0.2	0.3	0.9

Notasi che la seconda riga interna della tabella si ottiene, elemento per elemento, come uno meno il valore nella prima riga della colonna corrispondente visto che siamo di fronte a variabili di Bernoulli. In totale i parametri sono 4, i.e. i valori che si leggono nella prima riga.

Esempio 12.3: Separazione condizionale dal grafo



La rete di Bayes riportata qui sopra corrisponde alla fattorizzazione

$$\mathbb{P}(X_1, X_2, X_3, X_4) = \mathbb{P}(X_1 | X_4) \cdot \mathbb{P}(X_2 | X_3, X_4) \cdot \mathbb{P}(X_3) \cdot \mathbb{P}(X_4)$$

Osservando in cosa questa differisce dalla fattorizzazione che si otterrebbe applicando iterativamente la formula di probabilità composta si nota che valgono le seguenti indipendenze:²

- (i) $\{X_1\} \perp \{X_2, X_3\} \mid \{X_4\}$
- (ii) $\{X_3\} \perp \{X_4\}$

Ci chiediamo se inoltre si ha anche che (iii) $\{X_1\} \perp \{X_2\} \mid \{X_4\}$. Rispondere a questa domanda a priori non è banale, ma in ogni caso possiamo riformulare il quesito nel modo seguente: (i), (ii) \models (iii)? Ossia, ogni distribuzione che obbedisce ad (i) e (ii) obbedisce anche a (iii)?

Il problema dell'entailment per le indipendenze condizionali è tuttavia indecidibile, infatti le possibili diverse distribuzioni sono un'infinità non numerabile in quanto nel

caso generale i parametri sono da scegliere in \mathbb{R} . In questo caso tuttavia possiamo procedere diversamente:

$$\mathbb{P}(X_1, X_2 | X_4) = \sum_{X_3} \frac{\mathbb{P}(X_1, X_2, X_3, X_4)}{\mathbb{P}(X_4)} = \sum_{X_3} \frac{\mathbb{P}(X_1 | X_4) \cdot \mathbb{P}(X_2 | X_3, X_4) \cdot \mathbb{P}(X_3) \cdot \mathbb{P}(X_4)}{\mathbb{P}(X_4)}$$

Da cui, semplificando il termine $\mathbb{P}(X_4)$ a denominatore con quello a numeratore e portando fuori $\mathbb{P}(X_1 | X_4)$ che non dipende dall'indice della sommatoria

$$\mathbb{P}(X_1, X_2 | X_4) = \mathbb{P}(X_1 | X_4) \cdot \sum_{X_3} \mathbb{P}(X_2 | X_3, X_4) \cdot \mathbb{P}(X_3) \quad (12.1)$$

Calcoliamo adesso invece $\mathbb{P}(X_2 | X_4)$, procedendo per marginalizzazione:

$$\mathbb{P}(X_2 | X_4) = \sum_{X_1, X_3} \frac{\mathbb{P}(X_1, X_2, X_3, X_4)}{X_4} = \frac{\mathbb{P}(X_4) \cdot \sum_{X_1} \mathbb{P}(X_1 | X_4) \cdot \sum_{X_3} \mathbb{P}(X_2 | X_3, X_4) \cdot \mathbb{P}(X_3)}{\mathbb{P}(X_4)}$$

Semplificando il termine $\mathbb{P}(X_4)$ a denominatore con quello a numeratore e notando che la sommatoria su X_1 è uguale ad 1 si ricava il seguente risultato:

$$\mathbb{P}(X_2 | X_4) = \sum_{X_3} \mathbb{P}(X_2 | X_4) = \sum_{X_3} \mathbb{P}(X_2 | X_3, X_4) \cdot \mathbb{P}(X_3) \quad (12.2)$$

Se sostituiamo la relazione 12.2 nella 12.1 si ottiene che

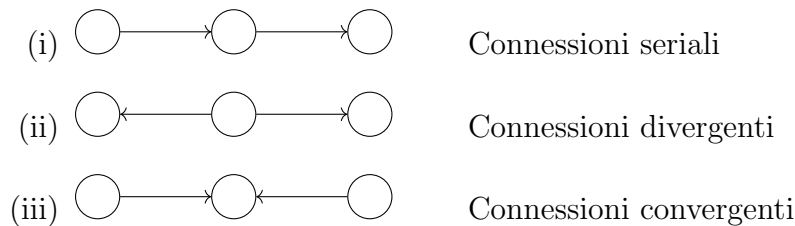
$$\mathbb{P}(X_1, X_2 | X_4) = \mathbb{P}(X_1 | X_4) \cdot \mathbb{P}(X_2 | X_4)$$

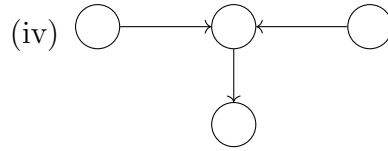
Ossia che X_1 e X_2 sono condizionalmente indipendenti data X_4 , infatti la probabilità che si verifichino insieme data X_4 è proprio il prodotto delle probabilità, come da definizione di indipendenza. ■

¹La notazione fa uso di parentesi graffe perché l'indipendenza condizionale e quella marginale sono relazioni tra insiemi

12.2 D-separazione

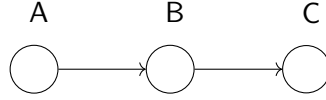
Introduciamo in questa sezione una ulteriore nozione che è quella di **d-separazione**. Questa riguarda tre insiemi di nodi in un grafo, che indicheremo come X, Y e Z e quando sussiste si scrive che $X \perp_D Y | Z$. La definizione vera e propria si articola in 4 casi distinti, che elenchiamo di seguito. Premettiamo comunque che ogni volta che parleremo di cammini, nonostante il grafo in questione sia orientato, intenderemo riferirci a cammini non orientati, i.e. non è rilevante il verso delle frecce.





Connessioni convergenti con discendente sul nodo di convergenza

Caso (i) - connessioni seriali

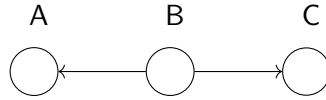


Se si mette evidenza in B , ossia si suppone di conoscere il valore che tale variabile assume, allora si blocca la propagazione del belief nel cammino non orientato da A verso C . Verifichiamo che in effetti è così, ovvero che $\{C\} \perp \{A\} \mid \{B\}$, i.e. che $\mathbb{P}(C, A \mid B) = \mathbb{P}(C \mid B) \cdot \mathbb{P}(A \mid B)$:

$$\mathbb{P}(C, A \mid B) \stackrel{def}{=} \frac{\mathbb{P}(C, A, B)}{\mathbb{P}(B)} \stackrel{rete}{=} \frac{\mathbb{P}(A) \cdot \mathbb{P}(B \mid A) \cdot \mathbb{P}(C \mid B)}{\mathbb{P}(B)} = \mathbb{P}(A \mid B) \cdot \mathbb{P}(C \mid B)$$

Dove l'uguaglianza segue riconoscendo nel membro di sinistra la formula di Bayes se si esclude il termine $\mathbb{P}(C \mid B)$. Considerando il primo e l'ultimo membro della catena si è effettivamente verificato che l'evidenza in B blocca la propagazione da A verso C come proposto.

Caso (ii) - connessioni divergenti

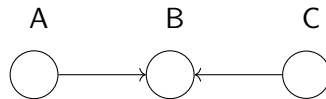


Vogliamo far vedere che se poniamo evidenza in B allora non ci può essere propagazione del belief da A verso C e viceversa. In altre parole, vogliamo dimostrare che $\{A\} \perp \{C\} \mid \{B\}$, e lo faremo facendo vedere che $\mathbb{P}(A, C \mid B) = \mathbb{P}(A \mid B) \cdot \mathbb{P}(C \mid B)$:

$$\mathbb{P}(A, C \mid B) \stackrel{def}{=} \frac{\mathbb{P}(A, C, B)}{\mathbb{P}(B)} \stackrel{rete}{=} \frac{\mathbb{P}(B) \cdot \mathbb{P}(A \mid B) \cdot \mathbb{P}(C \mid B)}{\mathbb{P}(B)} = \mathbb{P}(A \mid B) \cdot \mathbb{P}(C \mid B)$$

Se si considera il primo e l'ultimo membro della catena di uguaglianze si è verificato.

Caso (iii) - connessioni convergenti



Facciamo vedere che in assenza di informazioni A e C sono indipendenti, mentre non appena si inserisce l'evidenza in B si perde tale indipendenza. La verifica dell'indipendenza è immediata, infatti

$$\mathbb{P}(A, C) = \sum_B \mathbb{P}(A, B, C) \stackrel{rete}{=} \sum_B \mathbb{P}(B \mid A, C) \cdot \mathbb{P}(A) \cdot \mathbb{P}(C) = \left[\sum_B \mathbb{P}(B \mid A, C) \right] \cdot \mathbb{P}(A) \cdot \mathbb{P}(C)$$

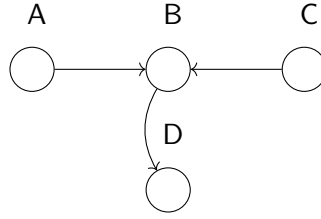
La sommatoria tra parentesi quadre fa uno e pertanto come volevamo far vedere $\mathbb{P}(A, C) = \mathbb{P}(A) \cdot \mathbb{P}(C)$. Adesso mostriamo che con evidenza in B non si ha indipendenza condizionale di A da C :

$$\mathbb{P}(A, C | B = b) = \frac{\mathbb{P}(A, C, B = b)}{\mathbb{P}(B = b)} \stackrel{rete}{=} \frac{\mathbb{P}(B = b | A, C) \cdot \mathbb{P}(A) \mathbb{P}(C)}{\mathbb{P}(B = b)} = \frac{\mathbb{P}(B = b | A, C)}{\mathbb{P}(B = b)} \cdot \mathbb{P}(A) \cdot \mathbb{P}(C)$$

Nessuno tuttavia può garantire che il fattore che moltiplica $\mathbb{P}(A) \cdot \mathbb{P}(C)$ sia pari ad uno e quindi in effetti non si può concludere l'indipendenza condizionale. Si parla in questo caso di **explaining away**: è intuitivo che se si ha evidenza in B , allora A e C rappresentano due possibili spiegazioni e quindi se si trova una spiegazione alternativa, ad esempio si mette evidenza anche in C , allora il belief nell'altra spiegazione debba calare, da cui la mancanza di indipendenza. Questo caso inoltre è quello che giustifica la necessità di usare frecce orientate.

Un nodo come quello centrale che rappresenta la variabile B , che nel cammino ha le frecce convergenti, si dice essere un **collider**; si noti in tal senso che la proprietà di un nodo di essere un collider dipende dal cammino considerato: esso potrebbe esserlo per un cammino e non esserlo per uno diverso. Si può notare che mettere evidenza in un collider fa passare il belief, mentre se nel collider non c'è evidenza si blocca il suo passaggio.

Caso (iv) - connessioni convergenti con discendente



In questo caso ci si chiede se sia la stessa cosa mettere evidenza in un collider (B nell'immagine) o in un discendente di un collider (D nell'immagine).

$$\mathbb{P}(A, B, C) = \sum_D \mathbb{P}(A, B, C, D) = \sum_D \mathbb{P}(B | A, C) \cdot \mathbb{P}(D | B) \cdot \mathbb{P}(A) \cdot \mathbb{P}(C) = \mathbb{P}(B | A, C) \cdot \mathbb{P}(A) \cdot \mathbb{P}(C)$$

L'ultima uguaglianza si ottiene tirando fuori dalla somma tutto ciò che non dipende da D ed infine notando che la somma che resta fa uno. Osserviamo quindi, prendendo il primo e l'ultimo membro della precedente catene di uguaglianze, che si ottiene la stessa fattorizzazione del caso (iii) e quindi si ha ancora indipendenza marginale tra A e C . Verifichiamo che inoltre, mettendo evidenza in D , si perde l'indipendenza tra queste due analogamente al caso precedente.

$$\mathbb{P}(A, C | D = d) = \sum_B \mathbb{P}(A, B, C | D = d) \stackrel{rete}{=} \sum_B \frac{\mathbb{P}(B | A, C) \cdot \mathbb{P}(D | B) \cdot \mathbb{P}(A) \cdot \mathbb{P}(C)}{\mathbb{P}(D = d)}$$

Possiamo a questo punto tirare fuori dalla sommatoria tutto ciò che non dipende da B , ad esempio il fattore $\mathbb{P}(A) \cdot \mathbb{P}(C)$ da cui si ottiene che

$$\mathbb{P}(A, C | D = d) = \left[\sum_B \frac{\mathbb{P}(B | A, C) \cdot \mathbb{P}(D | B)}{\mathbb{P}(D = d)} \right] \cdot \mathbb{P}(A) \cdot \mathbb{P}(C)$$

Anche qui non abbiamo nessuna garanzia che sommando su tutti i valori di B si ottenga risultato pari ad uno e quindi di conseguenza non è garantita l'indipendenza condizionale di A da C se si osserva D , analogamente a quanto accadeva nel caso precedente con evidenza nel collider.

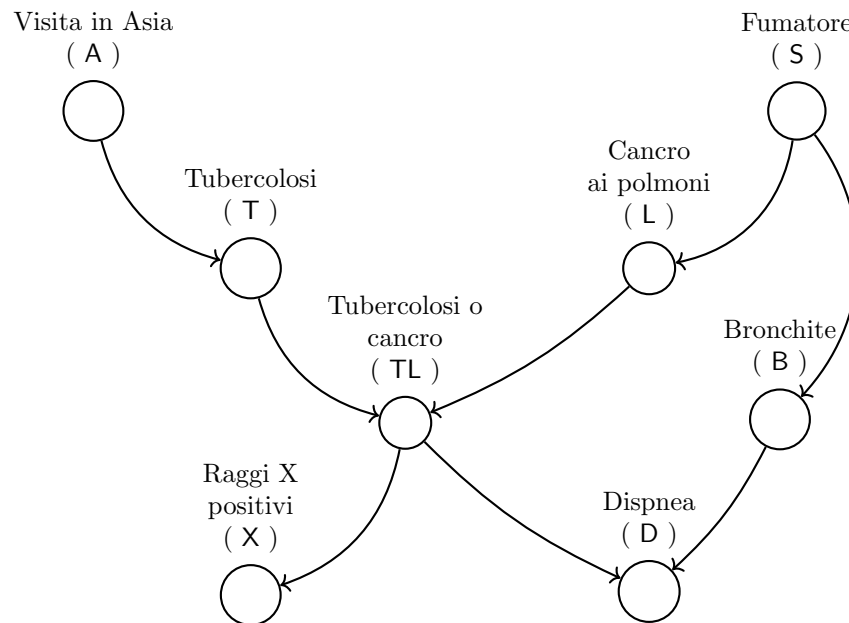
Definizione di D-separazione

Abbiamo discusso cosa succede nei quattro casi precedenti siamo pronti a dare una definizione di D-separazione:

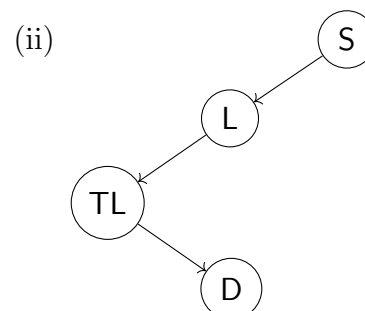
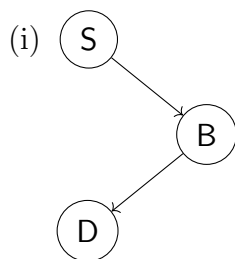
Definizione 12.2 (D-separazione). Siano X, Y, Z tre insiemi di variabili aleatorie, allora si dice che $X \perp_D Y \mid Z$ se ogni cammino tra X e Y è “bloccato” da Z , ovvero è verificata almeno una di queste due condizioni per ciascuno di questi:

- (i) Il cammino contiene un nodo in Z che non è un collider;
- (ii) Il cammino contiene un collider C tale che $C \notin Z$ e nessun suo discendente sta in Z

Esempio 12.4: Applicazione pratica del criterio



Ci chiediamo se nella precedente rete di Bayes $\{D\} \perp_D \{S\} \mid \{L, B\}$. Si deve immaginare di aver messo evidenza in B ed in L e ci si chiede se il belief può passare da D a S . Ci sono due cammini tra i due nodi in questione:



Ma il primo cammino è bloccato da \mathbf{B} in quanto è un elemento di \mathbf{Z} che sta nel cammino e non è un collider, mentre il secondo è bloccato per la stessa ragione da \mathbf{L} e quindi tutti i cammini da \mathbf{D} a \mathbf{S} sono bloccati: possiamo concludere che in effetti $\{\mathbf{D}\} \perp_D \{\mathbf{S}\} \mid \{\mathbf{L}, \mathbf{B}\}$.

Se avessimo preso invece $\mathbf{Z} = \{\mathbf{B}\}$ il secondo cammino non sarebbe stato bloccato e quindi avremmo concluso che $\{\mathbf{D}\} \not\perp_D \{\mathbf{S}\} \mid \{\mathbf{B}\}$. Altri esempi con riferimento alla rete di partenza sono che $\{\mathbf{A}\} \perp_D \{\mathbf{S}\} \mid \emptyset$ e $\{\mathbf{A}\} \not\perp_D \{\mathbf{S}\} \mid \{\mathbf{X}\}$ ed il lettore è invitato a verificarlo per esercizio. ■

9/11/2020

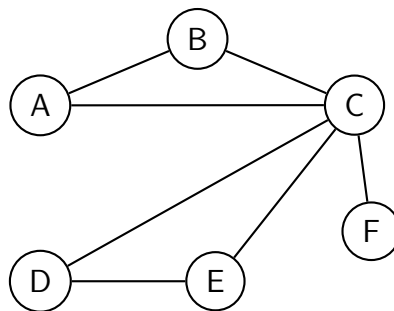
13.1 Limiti della D-separazione e Reti di Markov

Teorema 13.1.1. Supponendo di avere una distribuzione di probabilità che fattorizza in accordo ad un grafo orientato, vale la seguente relazione tra insiemi di variabili aleatorie:

$$X \perp_D Y \mid Z \implies \mathbb{P}(X \mid Y, Z) = \mathbb{P}(X \mid Z)$$

Ovvero se X è D-separato da Y data Z allora X è anche condizionalmente indipendente da Y dato Z . Il viceversa non è generalmente vero.

13.1.1 Modelli grafici non orientati: reti di Markov



Definizione 13.1 (U-separazione). Assegnato un grafo non orientato (rete di Markov), un universo di variabili aleatorie $U = \{X_1, \dots, X_n\}$ e tre insiemi di variabili X , Y e Z si dice che X è **U-separato** da Y data Z se e solo se il sottografo indotto da $U \setminus Z$ consiste di due componenti, una delle quali contiene X e l'altra contiene Y .

Equivalentemente alla precedente definizione, possiamo richiedere acciocché si abbia U-separazione che ogni cammino da un nodo in X ad uno in Y contenga un nodo in Z . Per esempio, con riferimento alla rete di Markov sopra disegnata, possiamo affermare che $\{A, B\} \perp_U \{D, E\} \mid \{C\}$ perché tutti i cammini tra A, B e D, E devono passare per C .

Il criterio di separazione per reti di Markov è più semplice da verificare rispetto a quello in reti di Bayes; il motivo per cui tuttavia le seconde sono più popolari è che per le reti di Markov è più difficile spiegare ad un esperto umano com'è fatta la fattorizzazione. Formalmente questa è fatta in questo modo:

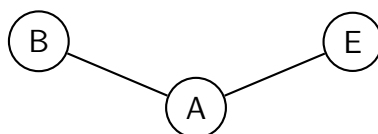
$$\mathbb{P}(X_1, \dots, X_n) = \frac{1}{z} \cdot \prod_c \phi(X_c)$$

Dove qui sopra la produttoria è fatta su tutte le cricche massimali del grafo in questione ed il fattore z^{-1} funge da normalizzatore e viene chiamato “funzione di partizione”; infine la ϕ è una cosiddetta “funzione potenziale”. Quest’ultime sono particolarmente difficili da assegnare perché non sono nemmeno interpretabili come delle probabilità, infatti l’unico vincolo a cui devono sottostare è di essere positive e pertanto sono più difficili da assegnare interrogando esperti umani.

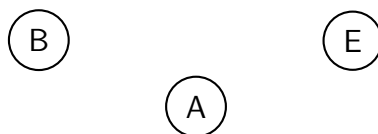
Grazie alle reti di Markov possiamo farci un'idea del perché il teorema 13.1.1 sia soltanto condizione sufficiente ma non necessaria: vedremo infatti adesso con dei contro-esempi che ci sono alcune indipendenze condizionali che possono essere catturate solo con reti di Markov ed altre solo con reti di Bayes.

Esempio 13.1: Fallimento di reti di Markov

Con riferimento al modello 12.1 si vuole catturare il fatto che la rapina ed il terremoto sono indipendenti tra di loro, ma smettono di esserlo se si osserva l'allarme (*explaining away*). Questo non si riesce a fare attraverso una rete di Markov. Una possibile idea potrebbe essere quella di collegarli così:



Con questi collegamenti si riuscirebbe a catturare l'assenza di indipendenza condizionale ma non la presenza di indipendenza marginale: in questo modo $B \not\perp E$, infatti il cammino da B a E passa da A che non contiene evidenza e quindi il belief può passare. Un'idea alternativa potrebbe essere quella di usare una rete senza archi:



In questo caso si riesce a catturare l'indipendenza marginale ma non l'assenza di quella condizionale e quindi nuovamente non funziona. Più in generale si possono provare tutti i possibili modi di collegare i nodi ed in nessun caso si riesce a catturare contemporaneamente che:

- (i) $\{B\} \perp \{E\}$
- (ii) $\{B\} \not\perp \{E\} \mid \{A\}$

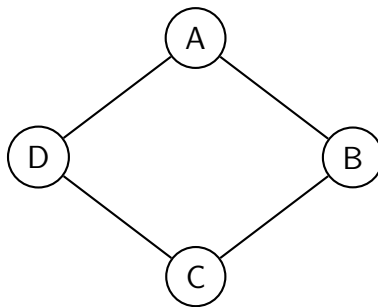
D'altra parte con una rete di Bayes catturare questa situazione è semplice, è infatti quello che si ha nel caso (iii) della definizione di D-separazione. ■

Esempio 13.2: Fallimento di reti di Bayes

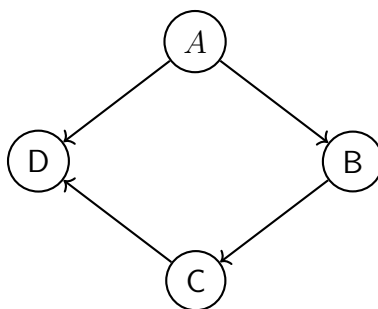
Sono assegnate quattro variabili: A, B, C e D e si vuole che due variabili siano condizionalmente indipendenti date le due rimanenti, ma allo stesso tempo le due rimanenti devono essere condizionalmente indipendenti date le prime:

- (i) $\{A\} \perp \{C\} \mid \{B, D\}$
- (ii) $\{B\} \perp \{D\} \mid \{A, C\}$

Con una rete di Markov è semplice catturare questa situazione: possiamo infatti collegare i nodi come segue:



In questo modo se si mette evidenza in B e D le variabili A e C risultano separate in accordo alla definizione 13.1, e se si mette evidenza in A e C le B e D risultano a loro volta separate. Invece non c'è modo di fare la stessa cosa con una rete di Bayes: questa per definizione deve essere un grafo orientato **aciclico** e l'unico modo per non ottenere cicli è quello di formare almeno un collider, come qui sotto:



Con questa rappresentazione B e D sono separate da $Z = \{A, C\}$, infatti ogni cammino da B a D contiene un nodo in Z che non è un collider e quindi è bloccato. Invece A non è separata da C dato $Z' = \{B, D\}$, infatti il cammino $\{(A, D), (D, C)\}$ non è bloccato da Z' e quindi si riesce a catturare l'indipendenza (ii).

Comunque si cambi l'orientazione delle frecce per non creare un ciclo deve sempre comparire un collider e pertanto si ripresenta ogni volta una situazione analoga a questa. ■

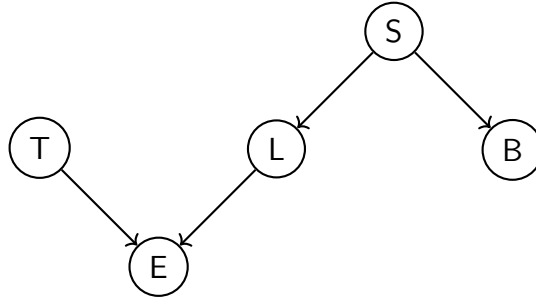
13.2 Inferenza in reti di Bayes

È assegnato un universo U di variabili aleatorie ed un grafo orientato \mathcal{G} con insieme di nodi U . Si assume che la probabilità congiunta $\mathbb{P}(U)$ fattorizzi secondo \mathcal{G} e che siano date una query $Q \subseteq U$ ed un'evidenza $E \subseteq U$. L'obiettivo è quello di calcolare $\mathbb{P}(Q | E)$ in modo efficiente, nonostante il problema sia NP-completo in generale. Ci sono almeno due strategie percorribili:

- (i) La prima strategia consiste in uno scambio di messaggi sul grafo \mathcal{G} , fattibile soltanto se la versione non orientata di \mathcal{G} è un albero, ossia \mathcal{G} è un cosiddetto **polytree**.
- (ii) La seconda strategia, che si può applicare anche quando la precedente non è adatta, consiste in uno scambio di messaggi su un junction tree di \mathcal{G} .

La prima tecnica è più generale ma è più complessa da raccontare, rimandiamo di conseguenza a [B12], *propagation on factor graphs*, mentre trattiamo in maniera esplicita la seconda, che può essere invece approfondita in [J97]. Il primo passaggio è quello di definire alcune operazioni sulle tabelle.

13.2.1 Algebra su tabelle di probabilità



Cerchiamo innanzitutto cosa intendiamo per tabella di probabilità: sia $V = \{T, E, L\}$, allora la tabella corrispondente deve possedere un'entrata per ciascuna diversa configurazione delle variabili T, E, L ed in corrispondenza di ciascuna di queste, la rispettiva probabilità congiunta. Assumiamo per fissare le idee che tutte le precedenti variabili siano di Bernoulli, allora una possibile tabella di probabilità per l'insieme V , indicata di seguito con la notazione t_V , è la seguente:

T	E	L	$\mathbb{P}(T, E, L)$
0	0	0	0.05
0	0	1	0.1
0	1	0	0
0	1	1	0.1
1	0	0	0.2
1	0	1	0.4
1	1	0	0.05
1	1	1	0.1

In realtà almeno temporaneamente non è richiesto che i valori nell'ultima colonna formino una distribuzione di probabilità e potrebbero essere normalizzati solo in seguito. In fin dei conti questo non rappresenta un ostacolo, infatti abbiamo già visto almeno in un'occasione che è sufficiente che tali numeri siano proporzionali a delle probabilità per poterli utilizzare. Le operazioni che vogliamo definire sulle tabelle sono le seguenti:

- (i) Prodotto tra due tabelle $t_V \cdot t_W$, con $V, W \subseteq U$
- (ii) Marginalizzazione rispetto ad un sottoinsieme $S \subseteq V$
- (iii) Inserimento di evidenza in una tabella t_V , con $V \subseteq U$

Prodotto tra tabelle

Il prodotto tra due tabelle dà luogo ad una nuova tabella definita sull'unione delle variabili presenti nei due fattori, ed in particolare se v^* e w^* sono una possibile configurazione per le variabili in V e W rispettivamente, si pone per definizione

$$t_V \cdot t_W(v^*, w^*) \stackrel{def}{=} t_V(v^*) \cdot t_W(w^*)$$

Si osservi che a priori non è richiesto che V e W siano due insiemi di variabili disgiunti; ovviamente in caso di variabile duplicata sono valide soltanto le coppie (v^*, w^*) in cui la variabile (o le variabili) in comune assume il medesimo valore.

Riportiamo di seguito un accenno di esempio in cui $V = \{T, E, L\}$ e $W = \{L, S\}$

T	E	L	$\mathbb{P}(T, E, L)$	L	S	$\mathbb{P}(L, S)$	T	E	L	S	$\mathbb{P}(T, E, L, S)$
0	0	0	0.05	0	0	0.1	0	0	0	0	$0.05 \cdot 0.1$
0	0	1	0.1	0	1	0.5	0	0	0	1	$0.05 \cdot 0.5$
0	1	0	0	1	0	0.2	0	0	1	0	$0.1 \cdot 0.2$
0	1	1	0.1	1	1	0.2	\vdots	\vdots			\vdots
1	0	0	0.2				1	1	1	1	$0.1 \cdot 0.2$
1	0	1	0.4								
1	1	0	0.05								
1	1	1	0.1								

Da sinistra verso destra, incontriamo rispettivamente t_V , t_W e $t_V \cdot t_W$.

Marginalizzazione

Sia assegnato un insieme di variabili aleatorie V ed un suo sottoinsieme $S \subseteq V$, allora la marginalizzazione è quell'operazione che ci consente eliminare le variabili in S da t_V . Nello specifico, se $v^* \in V$ è una configurazione delle variabili in V ed s^* è la corrispondente configurazione delle variabili in S , poniamo per definizione

$$t_{V \setminus S}(v^*) \stackrel{\text{def}}{=} \sum_S t_V(v^*)$$

Per esempio se prendiamo $V = \{T, E, L\}$ ed $S = \{L\}$ allora il risultato della marginalizzazione sarà la seguente tabella (assumendo t_V uguale a quella vista sopra):

T	E	$\mathbb{P}(T, E)$
0	0	0.15
0	1	0.1
1	0	0.6
1	1	0.15

Volendo vedere esplicitamente come opera la marginalizzazione in un caso particolare, se $v^* = (0, 0) \in V \setminus S$ allora $t_{V \setminus S}(v^*)$ si calcola sommando su tutti i possibili valori di S le righe di t_V in cui $T = 0$ e $E = 0$, ottenendo appunto come riportato sopra $0.05 + 0.1 = 0.15$.

Inserimento di evidenza

È assegnato un insieme di variabili aleatorie $V \subseteq U$ e la tabella di probabilità corrispondente t_V . Inserire evidenza in t_V significa porre a 0 la probabilità di tutte le righe la cui configurazione corrispondente è in disaccordo con l'evidenza.

Per esempio, mantenendo $V = \{T, E, L\}$ e t_V come negli esempi precedenti e prendendo come evidenza $\{T = 1, L = 0\}$, il risultato di questa operazione sarebbe questo:

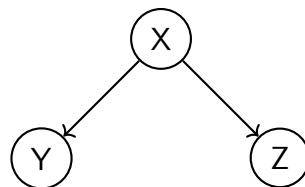
T	E	L	$\mathbb{P}(T, E, L)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0.2
1	0	1	0
1	1	0	0.05
1	1	1	0

13.2.2 Junction tree e inizializzazione

A partire dal grafo orientato si costruisce un secondo grafo i cui nodi sono delle aggregazioni di variabili di \mathcal{U} . Le intersezioni, i.e. variabili in comune tra nodi, vengono chiamate “separatori” e deve valere la running intersection: per ogni coppia di nodi V e W nel junction tree $V \cap W$ deve essere presente in tutti i nodi del cammino che connette V a W . Ad ogni nodo di questo nuovo grafo è associata una tabella di probabilità, ed infine per ogni variabile $X_i \in \mathcal{U}$ deve esistere un nodo che contiene X_i e $\text{PARENTS}(X_i)$.

Esempio 13.3: Esempio elementare di junction tree

Prendiamo un universo composto dalle variabili $\mathcal{U} = \{X, Y, Z\}$ che fattorizza secondo questa rete di Bayes, ed assumiamo le tabelle di probabilità riportate sotto:

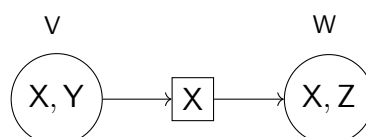


X	$\mathbb{P}(X)$
0	0.7
1	0.3

X	Y	$\mathbb{P}(Y X)$
0	0	0.2
0	1	0.8
1	0	0.4
1	1	0.6

X	Z	$\mathbb{P}(Z X)$
0	0	0.1
0	1	0.9
1	0	0.5
1	1	0.5

Un junction tree si può ottenere scegliendo come nodi $V = \{X, Y\}$ e $W = \{X, Z\}$



Una volta che siamo passati da una rete di Bayes ad uno dei junction trees associati, e vedremo a fine lezione come fare, possiamo procedere alla fase di inizializzazione, che consiste di tre passaggi:

- (i) Ad ogni cluster di variabili si associa una tabella di probabilità in cui tutti i valori sono inizializzati ad 1.
- (ii) Si ripete il passo precedente per tutti i separatori.
- (iii) Per ogni variabile del dominio si sceglie un cluster che contiene questa insieme ai suoi padri, e se il gruppo scelto è V , si aggiorna la tabella ad esso associato in accordo alla seguente regola:

$$t_V = t_V \cdot \mathbb{P}(X_i \mid \text{PARENTS}(X_i))$$

Ossia moltiplichiamo la tabella esistente per la tabella di probabilità della rete di Bayes associata a X_i e $\text{PARENTS}(X_i)$.

Riprendiamo l'esempio (13.3) e facciamo vedere passo per passo come opera l'inizializzazione su quel grafo. Per quanto riguarda (i) e (iii), i passaggi sono banali ed il risultato è il seguente:

X	Y	1	X	Z	1	X	1
0	0	1	0	0	1	0	1
0	1	1	0	1	1	1	1
1	0	1	1	0	1		
1	1	1	1	1	1		

Adesso vediamo invece come opera il passaggio (iii): per quanto riguarda la variabile X potremmo scegliere come cluster che la contiene insieme ai suoi padri sia V che W ; qui scegliamo di associarlo al cluster V e quindi si aggiorna t_V così:

X	Y	$\mathbb{P}(X)$	
0	0	$1 \cdot 0.7$	$\longrightarrow 0.7$
0	1	$1 \cdot 0.7$	$\longrightarrow 0.7$
1	0	$1 \cdot 0.3$	$\longrightarrow 0.3$
1	1	$1 \cdot 0.3$	$\longrightarrow 0.3$

La variabile Y ha come padre X e quindi dobbiamo individuare un cluster che contenga insieme X e Y ; ovviamente c'è un unico cluster con questa proprietà che è ancora una volta V , e quindi si aggiorna nuovamente t_V con il seguente risultato:

X	Y	$\mathbb{P}(X)\mathbb{P}(Y X)$	
0	0	$0.7 \cdot 0.2$	$\longrightarrow 0.14$
0	1	$0.7 \cdot 0.8$	$\longrightarrow 0.56$
1	0	$0.3 \cdot 0.4$	$\longrightarrow 0.12$
1	1	$0.3 \cdot 0.6$	$\longrightarrow 0.18$

Infine per quanto riguarda la variabile Z , essa ha come padre il solo X , e quindi va trovato un cluster che contenga entrambi; l'unico candidato è W e quindi aggiorniamo la tabella

t_W secondo la regola stabilita:

X	Z	$\mathbb{P}(Z X)$	
0	0	$1 \cdot 0.1$	$\rightarrow 0.1$
0	1	$1 \cdot 0.9$	$\rightarrow 0.9$
1	0	$1 \cdot 0.5$	$\rightarrow 0.5$
1	1	$1 \cdot 0.5$	$\rightarrow 0.5$

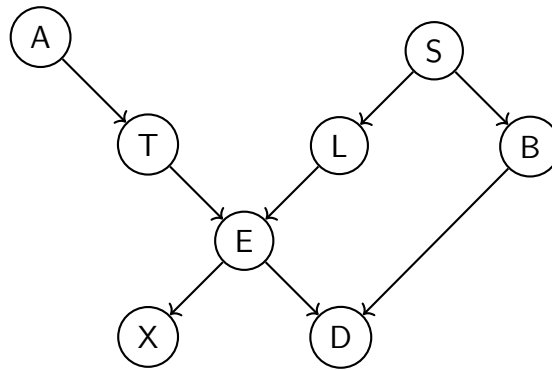
Alla fine di questa fase la tabella t_V corrisponde a $\mathbb{P}(X) \cdot \mathbb{P}(Y | X)$ mentre t_W corrisponde a $\mathbb{P}(Z | X)$. Inoltre si osserva che vale la seguente relazione almeno a questo punto:

$$\mathbb{P}(U) = \frac{t_V \cdot t_W}{t_S} = \frac{\mathbb{P}(X) \cdot \mathbb{P}(Y | X) \cdot \mathbb{P}(Z | X)}{1}$$

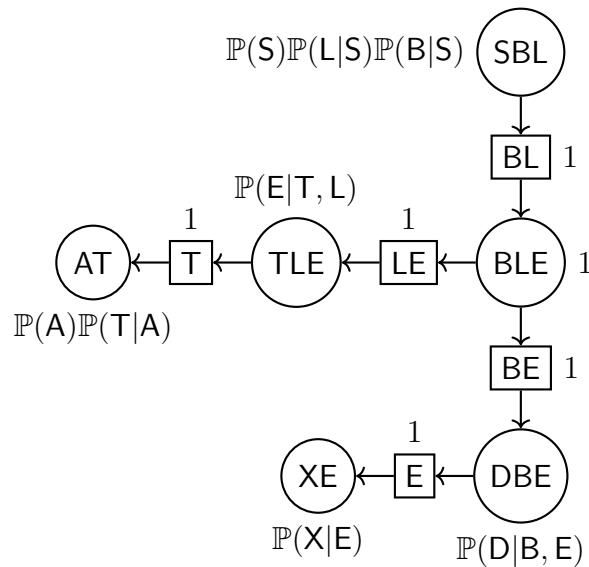
In realtà questa si può far vedere che resta un'invariante per tutto l'algoritmo.

Esempio 13.4: Esempio di junction tree

Si considera il seguente universo di variabili: $U = \{A, S, T, L, B, E, X, D\}$, e supponiamo che la distribuzione congiunta fattorizzi secondo il seguente grafo:

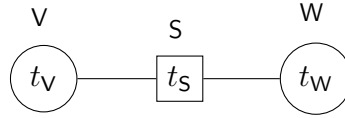


A seguito della fase di inizializzazione si potrebbe ottenere il seguente junction tree con questa fattorizzazione:



13.2.3 Scambio di messaggi su junction tree

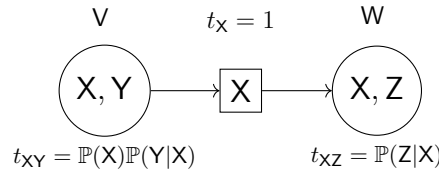
Sul junction tree associato alla rete di Bayes c'è una operazione fondamentale che si chiama **assorbimento** (*absorption*). Supponiamo per semplicità di avere un albero composto da due soli nodi come in figura:



Questa operazione consiste nell'esecuzione dei seguenti tre passaggi:

- (i) Si calcola un messaggio da inviare da V a W nel modo seguente: $t_S^* = \sum_{V \setminus S} t_V$
- (ii) Si aggiorna la tabella di W in accordo alla seguente regola: $t_W^{new} = \frac{t_S^*}{t_S} \cdot t_W$
- (iii) Si pone $t_S \leftarrow t_S^*$; non possiamo farlo prima perché il vecchio valore di t_S serve per aggiornare la tabella di W.

Vediamo come questa fase opera sul grafo dell'esempio (13.3), che dopo la fase di inizializzazione conterrà le seguenti tabelle:

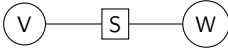


- (i) $t_X^* = \sum_Y t_{XY} = P(X)P(Y|X) = \sum_Y t_{XY} = P(X, Y) = P(X)$
- (ii) $t_{XZ}^{new} = \frac{P(X)}{1} \cdot P(Z|X) = P(Z|X) \cdot P(X) = P(X, Z)$
- (iii) $t_X = P(X)$

Lo scambio di messaggi funziona allo stesso modo se decidiamo di effettuarlo da W verso V. Inoltre è facile notare che succede questo:

Osservazione 13.2.1. In una rete con due nodi come questa, se componiamo un assorbimento da V verso W con uno nel verso opposto, il secondo assorbimento non produce nessun effetto.

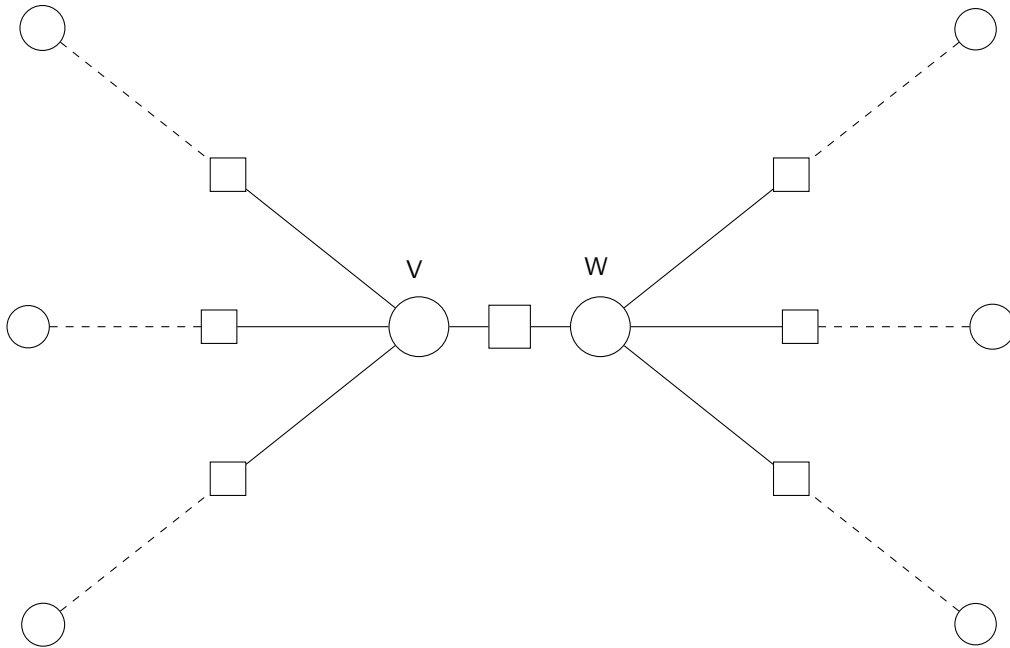
Immaginiamo, dopo aver effettuato il primo assorbimento, di volerlo fare anche nell'altra direzione: allora dobbiamo calcolarci $t_X^{**} = \sum_Z t_{XZ} = \sum_Z P(X, Z) = P(X)$ ed aggiornare la tabella t_{XY} moltiplicandola per $\frac{t_X^{**}}{t_X^*} = \frac{P(X)}{P(X)} = 1$ e quindi in effetti t_{XY} non viene modificata, come da osservazione. Ne approfittiamo per introdurre una definizione importante che è quella di consistenza di un collegamento:

Definizione 13.2 (Link consistency). Un collegamento  si dice essere **consistente** se $\sum_{V \setminus S} t_V = t_S = \sum_{V \setminus S} t_W$

Quando un collegamento è consistente significa che i due nodi “estremi” contengono la medesima informazione rispetto al separatore, e l’operazione di assorbimento è ciò che ci permette di rendere consistente un collegamento. In tale ottica è naturale che comporre due assorbimenti non abbia effetti diversi da assorbire una sola volta: eseguire l’assorbimento su un collegamento già consistente non comporta cambiamenti.

La nozione di consistenza di collegamento è di carattere locale, mentre noi vorremmo una nozione analoga di carattere globale, ovvero assegnati due nodi V e W tali che $I = V \cap W \neq \emptyset$ vorremmo che marginalizzando rispetto ad I le due tabelle si ottenesse la stessa quantità. Se prendiamo come riferimento la rete dell’esempio (13.4), si vorrebbe che $\sum_{T,L} t_{TLE} = \sum_X t_{XE}$ il che significherebbe che le due tabelle conterrebbero la stessa informazione su $\mathbb{P}(E)$, come è naturale richiedere. Se sono localmente consistenti i collegamenti $TLE - BLE$, $BLE - DBE$ e $DBE - XE$ allora tale richiesta è sicuramente soddisfatta, i.e. abbiamo anche consistenza globale nel collegamento $TLE - XE$, e qui si vede tutta l’importanza di avere un junction tree, per il quale deve per definizione valere la running intersection.

Per rendere consistente secondo questa nuova nozione un collegamento non è più sufficiente eseguire l’assorbimento se al posto di un singolo collegamento abbiamo una catena; si rende necessario invece uno scambio di messaggi da un nodo verso tutti gli altri.



Siamo davanti ad un albero, quindi non è possibile che ci siano dei cicli, ad esempio non ci può essere un cammino che unisce il primo nodo in alto a sinistra con il primo in alto a destra, il cammino tra questi due deve passare necessariamente da V e W , e così in generale. L’idea è che per poter fare l’assorbimento da V verso W per poter avere consistenza globale si debba attendere che siano stati fatti gli assorbimenti dai nodi a sinistra di V verso V nel nostro esempio; in generale prima che esso possa mandare l’informazione avanti deve averla ricevuta dai propri vicini. Una volta concluso l’assorbimento da V a W , il secondo può a sua volta procedere a propagare il messaggio ai suoi vicini, finché non raggiungeranno i nodi “estremi”. Differentemente dal caso di un collegamento singolo, dove un secondo passaggio di assorbimento non produceva effetti, in questo caso si rende necessario effettuarlo, infatti, ad esempio, il primo nodo in alto a destra ed il primo in

alto a sinistra potrebbero contenere informazioni su variabili diverse; in altre parole ogni nodo non solo deve ricevere informazioni da tutti gli altri ma anche mandarle verso tutti gli altri.

Essendo il grafo in questione un albero si può eleggere un nodo a caso come radice ed inviare l'evidenza dalle foglie verso tale radice (*collect evidence*) e poi da questa verso le foglie (*distribute evidence*). Così facendo abbiamo garanzia che ogni nodo abbia propagato la propria informazione verso tutti gli altri ed abbia ricevuto i messaggi da ogni altro nodo al termine dell'operazione, e quindi tutti gli archi saranno consistenti, i.e. si ha consistenza globale.

Il meccanismo descritto sopra può anche essere utilizzato per calcolare delle probabilità dopo aver inserito evidenza. Sempre usando la rete dell'esempio (14.4), immaginiamo di voler calcolare $\mathbb{P}(\mathbf{X}|\mathbf{S})$: allora possiamo inserire evidenza su \mathbf{S} in t_{SLB} , i.e. azzerare le righe della tabella che sono in disaccordo col valore osservato di \mathbf{S} , dopodiché si effettuano le operazioni di collect e distribute ed alla fine ci ritroveremo in t_{XE} delle informazioni relative alla distribuzione congiunta di \mathbf{X} e \mathbf{E} (a meno di un fattore di normalizzazione). Se vogliamo informazioni su \mathbf{X} è sufficiente marginalizzare sommando la tabella su \mathbf{E} , e quello che si ottiene è una tabella su \mathbf{X} . Poiché siamo partiti dall'aver azzerato tutte le righe che sono in disaccordo con l'evidenza su \mathbf{S} , si ha la garanzia che tale tabella rappresenti $\mathbb{P}(\mathbf{X}|\mathbf{S})$.

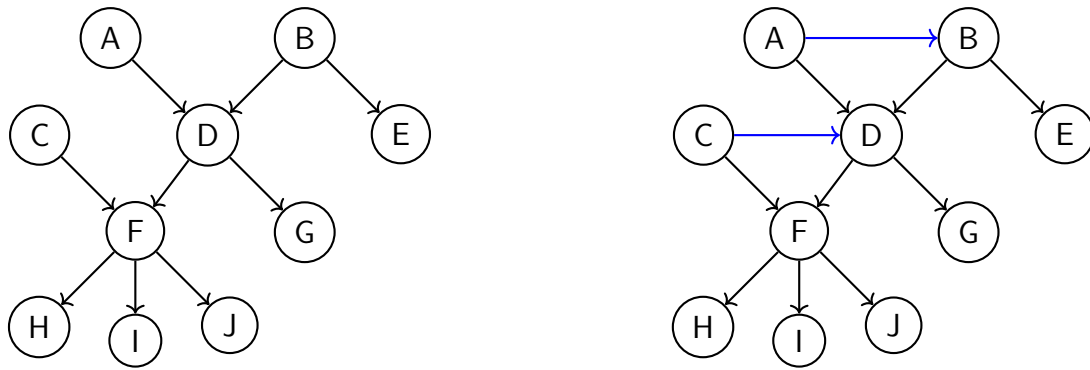
Quanto trattato qui chiude il problema dell'inferenza. Osserviamo a tal proposito che l'assorbimento ha un costo che dipende dalle dimensioni delle tabelle, che a loro volta contengono un numero di elementi esponenziale nella cardinalità del cluster corrispondente, quindi l'algoritmo proposto è efficace quando si riescono a produrre dei junction trees i cui clusters non sono troppo grandi (non c'è modo di sapere a priori se si riescono a generare alberi con nodi piccoli).

13.2.4 Costruzione di junction trees su grafi orientati

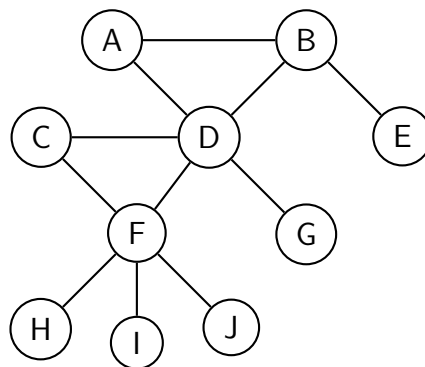
Abbiamo già visto nella trattazione dei problemi di soddisfacimento di vincoli come si costruisce un junction tree a partire da un grafo non orientato. Per costruirne uno sulla base di un grafo orientato non è sufficiente considerare la sua versione non orientata ed applicare la triangolazione, perché questo non garantirebbe l'esistenza di un cluster che contenga la variabile e tutti i suoi genitori per ogni possibile $\mathbf{X}_i \in \mathbf{U}$.

Il procedimento è il seguente: a partire dal grafo originario \mathcal{G} si effettua una procedura, illustrata di seguito, per ottenere un cosiddetto **grafo morale**, che è non orientato e quindi è possibile applicare su di questo la triangolazione. Il risultato di questa operazione è un cosiddetto **junction graph**, che contiene le cricche del grafo triangolato e dal quale, estraendo un albero ricoprente massimo, si ottiene un junction tree.

L'operazione che ci consente di ottenere un grafo morale a partire da \mathcal{G} si chiama **moralizzazione**; il termine deriva dal fatto che due genitori di uno stesso nodo debbano essere collegati (come se due genitori per essere considerati tali dovessero essere sposati, sennò sarebbe immorale). L'operazione è in realtà molto semplice: per ogni nodo in \mathbf{U} , se i suoi genitori non sono "sposati" li colleghiamo, ed una volta terminata questa prima fase si trasformano tutti i collegamenti orientati in collegamenti non orientati. Vediamo come la moralizzazione opera su un grafo esemplificativo:

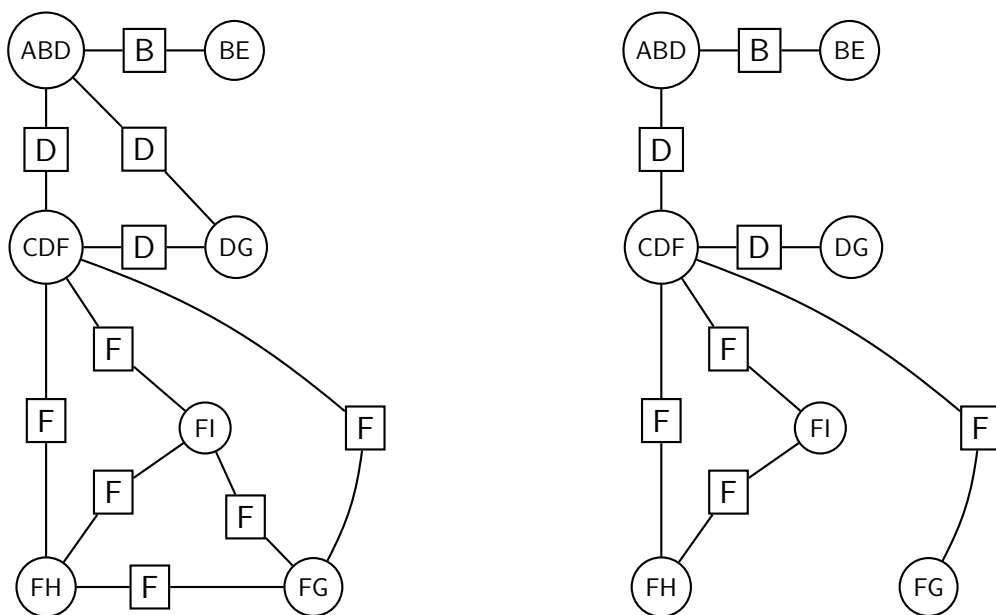


Nell'immagine di sinistra si vede il grafo originale, in quella di destra invece troviamo il risultato parziale della moralizzazione. È richiesto come specificato sopra di sbarazzarsi delle frecce per ottenere il grafo morale. Esso sarà quindi fatto così:



Questo grafo è già triangolato e si possono facilmente individuare le cricche: queste sono

$$\{\{A, B, D\}, \{C, D, F\}, \{D, G\}, \{B, E\}, \{F, I\}, \{F, H\}, \{F, J\}\}$$



Nella figura di sinistra è riportato il junction graph i cui nodi corrispondono alle cricche individuate sopra, mentre nella figura di sinistra è ritratto uno dei possibili junction trees che si possono estrarre da esso: ciascuno di questi dovrebbe essere un albero ricoprente

massimo del grafo di destra, in cui il peso di ciascun collegamento è dato dalla cardinalità dell'intersezione. Poiché tutte queste hanno lo stesso numero di elementi, i.e. 1, un qualsiasi albero ricoprente è adatto: è sufficiente eliminare un collegamento per ciascun ciclo e si ottiene un junction tree con tutte le proprietà richieste.

12/11/2020

14.1 Apprendimento dei parametri in Reti di Bayes

Avendo a disposizione dei dati vogliamo estrarre due elementi: una struttura (indipendenze condizionali) e dei parametri (tabelle di probabilità condizionate). Almeno momentaneamente assumeremo di lavorare nell'apprendimento dei soli parametri e che quindi qualcuno abbia già assegnato la struttura. Si parla in questo caso di **parameter learning**. Un elemento di una tabella, che rappresenta uno dei parametri, lo connoteremo nel modo seguente:

$$\theta_{\ell jk} = \mathbb{P}(X_{\ell} = j \mid \text{PARENTS}(X_{\ell}) = k)$$

Dove qui sopra $j \in \text{DOM}(X_{\ell})$ mentre k è una delle possibili configurazioni dei genitori di X_{ℓ} .

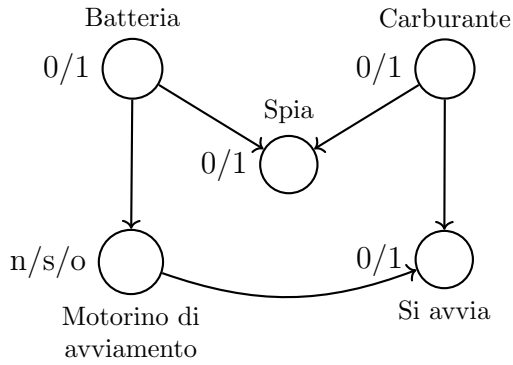
Definizione 14.1 (Dataset). Chiamiamo **dataset** una tabella in cui ciascuna riga corrisponde ad un esempio, i.e. un'assegnazione delle variabili coinvolte. Tutti gli esempi sono assunti indipendenti e provenienti dalla stessa distribuzione.

Se il dataset riguardasse una singola variabile aleatoria di Bernoulli X , taluno sarebbe della forma $\mathcal{D} = \{0, 1, 1, 1, 0, 0, 1\}$ (i valori sono assegnazioni a caso). In questo caso semplice la tabella su X coinvolgerebbe un singolo parametro θ che rappresenta la probabilità di successo, i.e. che $X = 1$, ed uno dei modi possibili di stimare il suo valore è quello di utilizzare la tecnica della massima verosimiglianza: l'idea è quella di definire una funzione di verosimiglianza (*likelihood*) $\mathcal{L}(\theta)$ che rappresenta la probabilità del dataset in funzione del parametro θ , che in questo caso particolare sarebbe

$$\mathcal{L}(\theta) = \mathbb{P}(\mathcal{D}|\theta) = \theta^4 \cdot (1 - \theta)^3$$

E quindi scegliere il valore θ^* che massimizza questa funzione. Normalmente $\mathcal{L}(\theta)$ si scrive come prodotto delle densità delle variabili coinvolte e quindi per calcolare il valore di θ^* occorrerebbe trovare il punto di massimo di un prodotto di funzioni; poiché questo richiede il calcolo di una derivata che diventa laborioso molto facilmente, si decide invece di massimizzare $\ln(\mathcal{L}(\theta))$, forti del fatto che la funzione logaritmo è monotona strettamente crescente e quindi realizza il proprio massimo quando è massimo il suo argomento. Nel caso di una variabile di Bernoulli dal calcolo risulta che lo stimatore di massima verosimiglianza del parametro θ è \bar{X} , ovvero la media aritmetica dei dati; altri risultati ed approfondimenti sul tema possono essere trovati in [PDR20].

Idealmente se il dataset fosse composto da un numero infinito di righe avremmo la garanzia che lo stimatore di massima verosimiglianza restituirebbe il vero valore del parametro da stimare.

Esempio 14.1: Stima in massima verosimiglianza in pratica

Esempio	Motorino	Carburante	Si avvia
1	Normale	Sì	Sì
2	Lento	Sì	Sì
3	Normale	Sì	Sì
4	No	Sì	No
5	Normale	Sì	Sì
6	Normale	Sì	No
7	Lento	Sì	No
8	Normale	Sì	Sì
9	Normale	No	No
10	Normale	Sì	No
11	Lento	No	No
12	Normale	No	No
13	No	Sì	No
14	Normale	Sì	Sì
15	Normale	Sì	Sì
16	No	No	Sì

Nella figura di sinistra abbiamo anche riportato la proiezione su tre colonne di un dataset. Dovremo assumere in ogni caso che i dati siano **completi**, ovvero tutte le variabili siano sempre osservate (assunzione forte). La funzione di verosimiglianza in questo caso dipenderà da 16 parametri, e vogliamo stimarli usando il dataset:

$$\begin{aligned}
 \mathcal{L}(\theta) &\stackrel{(i)}{=} \mathbb{P}(\mathcal{D} \mid \theta) = \prod_{i=1}^n \mathbb{P}(\mathbf{X}^{(i)} \mid \theta) \\
 &= \prod_{i=1}^n \mathbb{P}(\mathbf{X}_1^{(i)}, \mathbf{X}_2^{(i)}, \dots, \mathbf{X}_N^{(i)} \mid \theta) \\
 &\stackrel{(ii)}{=} \prod_{i=1}^n \prod_{j=1}^N \mathbb{P}(\mathbf{X}_j^{(i)} \mid \text{PARENTS}(\mathbf{X}_j^{(i)}), \theta) \\
 &\stackrel{(iii)}{=} \prod_{j=1}^N \prod_{i=1}^n \mathbb{P}(\mathbf{X}_j^{(i)} \mid \text{PARENTS}(\mathbf{X}_j^{(i)}), \theta) \\
 &\stackrel{(iv)}{=} \prod_{j=1}^N \mathcal{L}_{ij}(\vec{\mathbf{X}}_j \mid \text{PARENTS}(\vec{\mathbf{X}}_j), \theta)
 \end{aligned}$$

L'uguaglianza (i) segue dal fatto che gli esempi sono indipendenti, la (ii) dalla fattorizzazione del grafo, la (iii) cambiando l'ordine delle produttorie, possibile in quanto queste sono finite, e nel secondo membro di (iv) $\vec{\mathbf{X}}_j$ rappresenta la colonna j -esima del dataset. Con questa semplice procedura matematica abbiamo trasformato un singolo problema di massima verosimiglianza su 16 parametri in N problemi indipendenti di massima verosimiglianza, ciascuno dei quali coinvolge solo una variabile insieme ai suoi padre. Così, se volessimo stimare la probabilità che il veicolo si avvii sapendo che il motorino di avviamento è normale e c'è carburante potremmo farlo considerando il rapporto tra numero di righe che hanno “Si avvia=Sì”, “Motorino=Normale”, “Carburante=Sì” e numero di quelle con “Motorino=Normale” e “Carburante=Sì”. Per portare alcuni esempi numerici:

$$(i) \hat{\mathbb{P}}(\text{Si avvia=Sì} \mid \text{Motorino=No, Carburante=Sì}) = \frac{6}{8}$$

- (ii) $\hat{\mathbb{P}}(\text{Si avvia}=\text{Sì} \mid \text{Motorino}=\text{Lento}, \text{Carburante}=\text{Sì}) = \frac{1}{2}$
- (iii) $\hat{\mathbb{P}}(\text{Si avvia}=\text{No} \mid \text{Motorino}=\text{No}, \text{Carburante}=\text{Sì}) = 1$



A partire da questo esempio si può estrapolare una regola più generale: possiamo infatti stimare il parametro generico nel modo seguente a partire da un dataset \mathcal{D} utilizzando la stima a massima verosimiglianza:

$$\hat{\theta}_{\ell jk} = \frac{\sum_{i=1}^n \mathbb{1}\{\mathbf{X}_{\ell}^{(i)} = j\} \cdot \mathbb{1}\{\text{PARENTS}(\mathbf{X}_{\ell}^{(i)}) = k\}}{\sum_{j=1}^{d_{\ell}} \sum_{i=1}^n \mathbb{1}\{\mathbf{X}_{\ell}^{(i)} = j\} \cdot \mathbb{1}\{\text{PARENTS}(\mathbf{X}_{\ell}^{(i)}) = k\}} \stackrel{\text{def}}{=} \frac{N_{\ell jk}}{N_{\ell k}}$$

Dove il numeratore conta il numero di esempi in cui \mathbf{X}_{ℓ} si trova nella configurazione j ed i suoi genitori si trovano nella configurazione k , mentre il denominatore conta gli esempi in cui i genitori di \mathbf{X}_{ℓ} sono nella configurazione k (il classico rapporto tra casi favorevoli e totali). In una singola passata si possono identificare tutte le configurazioni diverse e salvarle in una tabella hash associate ad un contatore, inizialmente nullo; tutte le volte che si trova una configurazione già presente in tabella si aggiorna il suo contatore incrementandolo di 1 ed in questo modo in un tempo $O(n)$ si riempie la tabella con tutti i parametri. Qui sopra i valori $N_{\ell jk}$ ed $N_{\ell k}$ possono essere memorizzati separatamente piuttosto di memorizzare il loro rapporto, e questo consente di aggiornare il valore di un parametro in tempo costante se si osservano nuovi esempi. Questi due valori vengono inoltre chiamati **statistiche sufficienti**, infatti se conosciamo il loro valore per tutte le configurazioni possiamo scartare il dataset, in quanto questi riassumono in sé tutta l'informazione necessaria. Questa strategia, tuttavia, non è sempre applicabile: è necessario infatti che sia n che $N_{\ell k}$ siano sufficientemente grandi, in modo da avere un numero significativo di esempi la stima di ciascun parametro; per esempio se il nostro dataset riguardasse il lancio di una moneta ed avessimo tre esempi, tutti e tre “testa” questo non sarebbe sufficiente a concludere che la probabilità di testa è pari ad 1, infatti il numero di esempi non sarebbe abbastanza grande.

Definizione 14.2 (Distribuzione categorica). Sia \mathbf{X} una variabile aleatoria discreta con immagine $\mathbf{X}(\Omega) = \{t_1, t_2, \dots, t_d\}$, allora diciamo che \mathbf{X} ha distribuzione **categorica** di parametri $\theta_1, \theta_2, \dots, \theta_d$ tali che $\theta_1 + \dots + \theta_d = 1$ se ha densità data da

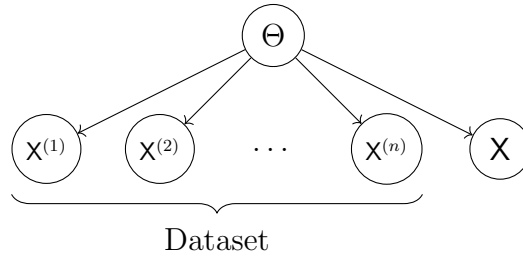
$$\mathbb{P}(\mathbf{X}) = \prod_{j=1}^d \theta_j^{\mathbb{1}\{\mathbf{X}=t_j\}}$$

Inoltre t_1, \dots, t_d vengono detti **livelli** di \mathbf{X} , e qui sopra $\theta_j \stackrel{\text{def}}{=} \mathbb{P}(\mathbf{X} = t_j)$.

14.2 Apprendimento bayesiano su reti di Bayes

Adesso vediamo una strategia alternativa e più raffinata rispetto a quella della massima verosimiglianza per l'apprendimento di parametri su reti di Bayes. L'idea fondamentale è che i parametri non vengono più considerati come dei numeri ma come variabili aleatorie ed il processo generativo degli esempi è il seguente: dapprima si campiona θ da una

data distribuzione di probabilità, chiamata **prior**, e solo successivamente campioniamo gli esempi $\mathbf{X}^{(i)}$ da una distribuzione condizionata dal parametro θ .



La rete qui sopra, rappresentante il processo generativo di un dataset, usa la semantica delle reti di Bayes, ossia gli esempi non sono indipendenti a priori, ma lo diventano appena si inserisce evidenza in Θ , i.e. si stabilisce il valore del parametro θ . Diversamente da quanto accadeva in precedenza, adesso in generale Θ è una variabile continua e quindi $\mathbb{P}(\Theta)$ non è più una tabella di probabilità, bensì una funzione; questo ha conseguenze immediate su come si effettuano alcune operazioni su di essa, per esempio nella marginalizzazione non avremo più una somma sulle variabili da eliminare ma piuttosto un integrale. La distribuzione di Θ può essere inoltre aggiornata sulla base del dataset osservato, ed in particolare ciò viene fatto applicando la formula di Bayes:

$$\mathbb{P}(\Theta \mid \mathcal{D}) = \frac{\mathbb{P}(\mathcal{D} \mid \Theta) \mathbb{P}(\Theta)}{\mathbb{P}(\mathcal{D})} \quad (14.1)$$

Osservare dei dati fa quindi cambiare la distribuzione di probabilità da cui si ritiene sia stato estratto il parametro.

Per quanto riguarda l'inferenza, immaginiamo di aver già osservato il dataset e di voler stabilire con che probabilità si incontra un generico **test point** \mathbf{X} (c.f.r rete riportata sopra). Possiamo procedere nel modo seguente:

$$\mathbb{P}(\mathbf{X} \mid \mathcal{D}) \stackrel{(i)}{=} \int_{\theta} \mathbb{P}(\mathbf{X}, \theta \mid \mathcal{D}) \cdot d\theta \stackrel{(ii)}{=} \int_{\theta} \mathbb{P}(\mathbf{X} \mid \theta) \cdot \mathbb{P}(\theta \mid \mathcal{D}) \cdot d\theta \quad (14.2)$$

L'uguaglianza (i) vale per marginalizzazione su θ , mentre la (ii) vale applicando la proprietà 11.9 e notando che \mathbf{X} è condizionalmente indipendente dal dataset se si osserva θ , come si può vedere dal modello grafico. Inoltre questa formula può essere ulteriormente espansa sostituendo all'ultimo membro l'equazione 14.1.

Qui sopra $\mathbb{P}(\Theta \mid \mathcal{D})$ è la distribuzione di probabilità dei parametri dato il dataset; per esempio nel caso di una Bernoulli si avrebbe, supponendo che il dataset riguardi il lancio di una moneta, h sia il numero di teste e t il numero di croci, e chiamando “successo” l'evento “esce testa”, che

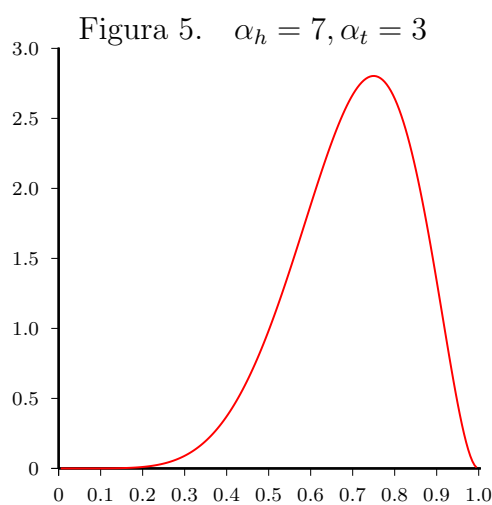
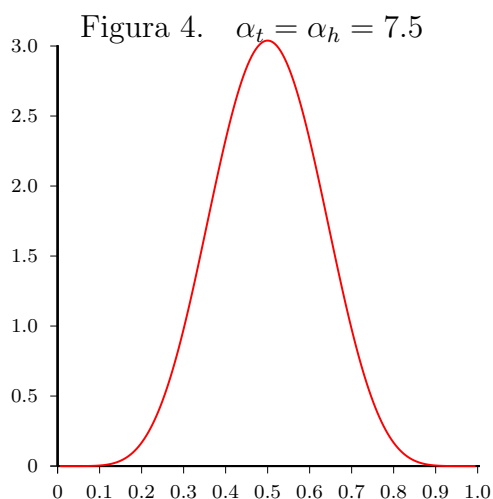
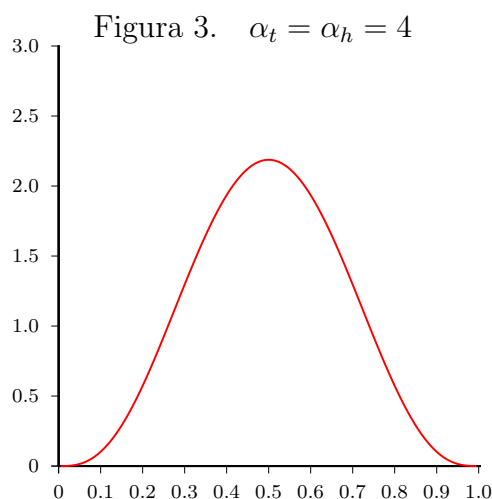
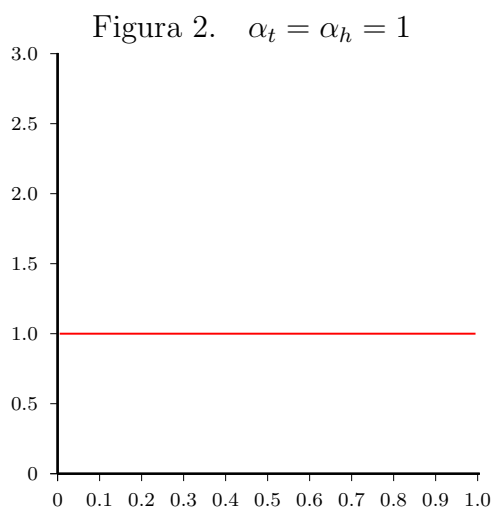
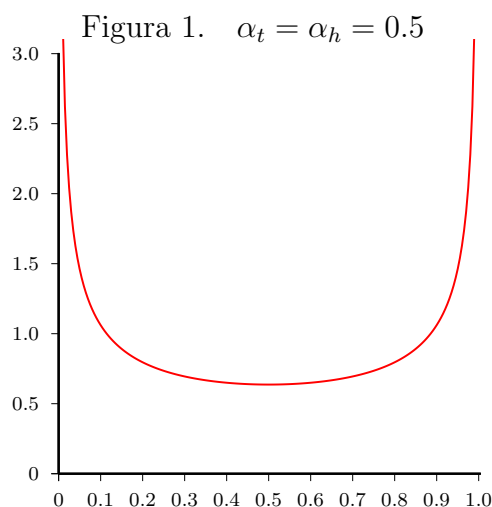
$$\mathbb{P}(\Theta \mid \mathcal{D}) \propto \theta^h \cdot (1 - \theta)^t \cdot \mathbb{P}(\Theta)$$

E per semplicità si vorrebbe che a seguito di un aggiornamento la distribuzione di probabilità da cui si ritiene sia stato estratto il parametro mantenesse la medesima forma analitica, perciò, portando avanti l'esempio, conviene prendere la distribuzione prior con questa forma:

$$\mathbb{P}(\Theta) = c \cdot \theta^{\alpha_h - 1} \cdot (1 - \theta)^{\alpha_t - 1}$$

Dove c è un fattore di normalizzazione che fa sì che l'integrale della densità su tutti i valori di θ faccia 1 come è necessario acciocché $\mathbb{P}(\Theta)$ sia in effetti una distribuzione. Si

può dimostrare che $c = \frac{\Gamma(\alpha_h + \alpha_t)}{\Gamma(\alpha_h) \cdot \Gamma(\alpha_t)}$ e la distribuzione risultante è nota come **Beta**. I due parametri α_h ed α_t vengono detti anche **iperparametri**, in quanto controllano la forma analitica della distribuzione. Inoltre si dice che questa è la distribuzione **coniugata bayesiana** della Bernoulli.



Qui sopra è riportato il grafico della distribuzione beta per alcuni valori dei parametri α_h ed α_t . Si osserva ad esempio che se $\alpha_t = \alpha_h$, più è grande il valore comune dei parametri e più la distribuzione tende a concentrarsi intorno a $\theta = 0.5$; in altre parole si

ha sempre meno possibilità di variazione per θ , riducendo l'incertezza su tale parametro; al limite la distribuzione tende ad una delta di Dirac. Se invece $\alpha_h \neq \alpha_t$ i valori tendono ad addensarsi intorno alla media della distribuzione, che si può dimostrare essere $\frac{\alpha_h}{\alpha_h + \alpha_t}$. Alla luce di quanto visto possiamo associare la seguente interpretazione ai valori di α_h ed α_t : il valore assoluto di questi iperparametri stabilisce quanto si crede nel prior e quanti esperimenti occorre fare per smuovere tale convinzione (\simeq convinzione virtuale nella mente di chi osserva l'esperimento). Per esempio se si stabilisce $\alpha_h = \alpha_t = 40$ è come se oltre al dataset si immaginasse di aver effettuato 80 esperimenti di cui 40 abbiano dato esito “testa” ed i rimanenti 40 siano risultati in “croce”.

Se adottiamo la distribuzione $\beta(\alpha_h, \alpha_t)$ come prior, allora la legge di aggiornamento della distribuzione da cui si crede sia stato estratto il parametro diventa la seguente:

$$\mathbb{P}(\Theta \mid \mathcal{D}) \propto \theta^{\alpha_h + h - 1} \cdot (1 - \theta)^{\alpha_t + t - 1}$$

Che è la distribuzione di una $\beta(\alpha_h + h, \alpha_t + t)$. Inoltre se $\mathbb{P}_\Theta = \beta(\alpha_h, \alpha_t)$ si può dimostrare che il valore atteso del parametro è

$$\mathbb{E}[\Theta] = \frac{\alpha_h}{\alpha_h + \alpha_t} \quad (14.3)$$

19/11/2020

Riprendiamo brevemente da dove abbiamo lasciato, con l'esempio della monetina; notiamo che se arriva un nuovo dato e vogliamo calcolare la probabilità che un certo lancio X prenda realizzazione h (i.e. testa) avendo osservato il dataset ed assegnato il valore degli iperparametri lo si può fare, in accordo all'equazione 14.2, nel modo seguente:

$$p = \int_{\theta} \mathbb{P}(X = h \mid \theta, \alpha_h, \alpha_t) \cdot \mathbb{P}(\theta \mid \mathcal{D}, \alpha_h, \alpha_t) d\theta$$

$$\stackrel{(i)}{=} \int_{\theta} \theta \cdot \mathbb{P}(\theta \mid \mathcal{D}, \alpha_h, \alpha_t) d\theta \stackrel{(ii)}{=} \mathbb{E}[\Theta]$$

L'uguaglianza (i) vale perché θ è per definizione la probabilità che esca testa, mentre la (ii) vale per la formula di composizione, dove il valore atteso è quello calcolato rispetto alla probabilità $\mathbb{P}(\cdot \mid \mathcal{D}, \alpha_h, \alpha_t)$. Ma sappiamo che se $\mathbb{P}_{\Theta} = \beta(\alpha_h, \alpha_t)$ il valore atteso di Θ è quello dato dalla relazione 14.3, e quindi tenendo conto anche del dataset \mathcal{D} in definitiva:

$$p = \frac{\alpha_h + h}{(\alpha_h + h) + (\alpha_t + t)}$$

15.1 Generalizzazione apprendimento dei parametri

Supponiamo adesso che invece di avere variabili di Bernoulli si abbia a che fare con delle categoriche (cfr definizione 14.2), allora si può far vedere che la coniugata bayesiana è la cosiddetta “distribuzione di Dirichlet”, che definiamo qui sotto:

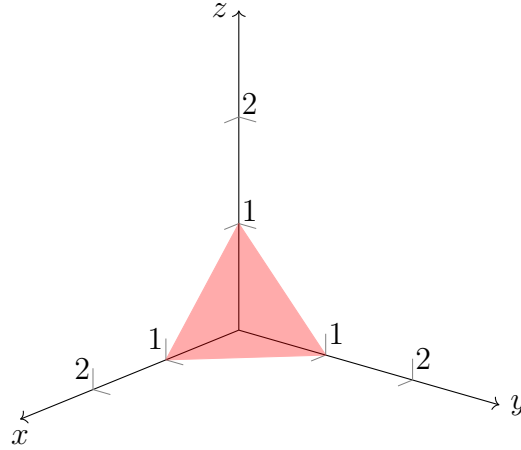
Definizione 15.1 (Distribuzione di Dirichlet). Sia Θ una variabile aleatoria discreta con immagine $\Theta(\Omega) = \{t_1, t_2, \dots, t_d\}$, allora si dice che Θ ha **distribuzione di Dirichlet** di parametri $\theta_1, \dots, \theta_d$ ed iperparametri $\alpha_1, \dots, \alpha_d$ se

$$\mathbb{P}(\Theta) = \frac{\Gamma(\alpha_1 + \dots + \alpha_d)}{\prod_{k=1}^d \Gamma(\alpha_k)} \cdot \prod_{k=1}^d \theta_k^{\alpha_k - 1}$$

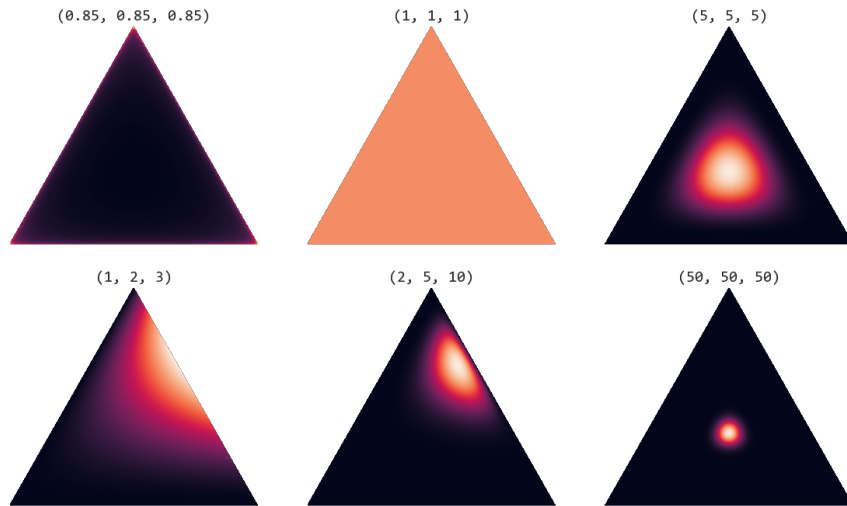
Dove il fattore che moltiplica la produttoria è semplicemente una costante utile a garantire che sia verificata la condizione seguente:

$$\int \dots \int_{\mathcal{V}} c \cdot \prod_{k=1}^d \theta_k^{\alpha_k - 1} d\theta_1 d\theta_2 \dots d\theta_d = 1$$

Dove $\mathcal{V} \stackrel{def}{=} \{(\theta_1, \dots, \theta_d) \in \mathbb{R}^d \mid \theta_1 + \dots + \theta_d = 1, \theta_1 \geq 0, \dots, \theta_d \geq 0\}$ è l'insieme di tutti i possibili valori che possono essere assunti dalla tupla di parametri della distribuzione.

Figura 15.1: \mathcal{V} nel caso in cui $d = 3$

L'insieme \mathcal{V} viene anche chiamato **simpleso**, o nel caso particolare illustrato in figura, **simpleso standard**. Si può utilizzare quest'ultimo per avere una intuizione grafica del significato degli iperparametri, facendo uso di una mappa di calore come la seguente:



Ad ogni punto sul simpleso corrisponde una diversa distribuzione di probabilità ed i valori di $\alpha_1, \dots, \alpha_d$, riportati sopra al vertice in alto di ciascun triangolo nella figura, stabiliscono quali distribuzioni sono più o meno probabili; in particolare le regioni più chiare sono quelle più probabili. Si nota per esempio che se gli iperparametri sono tutti più piccoli di 1 come nella figura in alto a sinistra le distribuzioni di probabilità che stanno sul bordo del simpleso sono leggermente più probabili, mentre più grandi sono i loro valori più la distribuzione tende a concentrarsi.

Vediamo adesso la legge di aggiornamento della distribuzione da cui si ritiene sia stato estratto il parametro, i.e. $\mathbb{P}(\Theta)$. Immaginiamo in tal senso di lavorare con un dataset di numeri interi, e supponiamo che ce ne siano d distinti con frequenze assolute n_1, n_2, \dots, n_d , allora

$$\mathbb{P}(\Theta \mid \mathcal{D}) \propto \theta_1^{n_1} \cdot \theta_2^{n_2} \cdot \dots \cdot \theta_d^{n_d} \cdot \text{DIR}(\Theta \mid \alpha_1, \dots, \alpha_d) = \text{DIR}(\alpha_1 + n_1, \dots, \alpha_d + n_d)$$

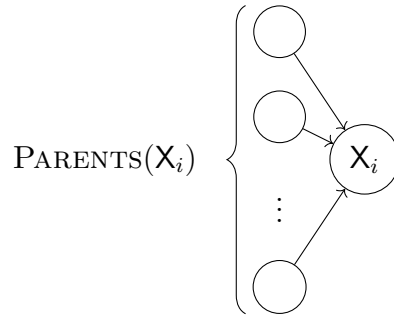
Ossia anche dopo aver osservato il dataset il posterior di una Dirichlet continua ad essere una Dirichlet, analogamente a quanto succedeva con la distribuzione Beta. Inoltre riguardo all'inferenza, immaginiamo di avere un esempio \mathbf{X} oltre al dataset, allora la probabilità che questa prenda realizzazione t_k è data da

$$\mathbb{P}(\mathbf{X} = t_k \mid \mathcal{D}) = \mathbb{E}[\Theta_k \mid \mathcal{D}] = \frac{\alpha_k + n_k}{\alpha + n}$$

Dove qui sopra n rappresenta la dimensione del dataset, mentre α è il cosiddetto “equivalent sample size”, che definiamo di seguito:

Definizione 15.2 (Equivalent sample size). Definiamo l'**equivalent sample size**, indicato col simbolo α , come la somma degli iperparametri di una Dirichlet. Intuitivamente il suo valore è tale per cui $\mathbb{P}(\mathbf{X} = t_k \mid \mathcal{D})$ è la stessa che si sarebbe ottenuto osservando l'unione di un dataset di dimensione α con effettivi $\alpha_1, \dots, \alpha_d$ e quello reale, di dimensione n , con statistiche sufficienti n_1, \dots, n_d

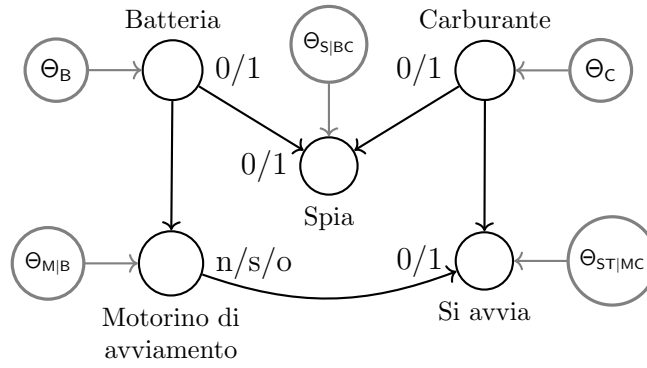
Abbiamo già avuto occasione di far vedere che se i dati sono completi allora piuttosto di risolvere un singolo problema di massima verosimiglianza per stimare il valore dei parametri, possiamo risolverne N indipendenti, ciascuno riguardante unicamente una sola variabile \mathbf{X}_i insieme ai suoi genitori. Assumiamo il seguente modello grafico:



La probabilità che la variabile i -esima prenda realizzazione t_k quando i suoi genitori stanno nella configurazione j , osservato il dataset, si può ancora scrivere come un posterior, ossia

$$\mathbb{P}(\mathbf{X}_i = t_k \mid \text{PARENTS}(\mathbf{X}_i) = j, \mathcal{D}) = \mathbb{E}[\Theta_{ijk}] = \frac{\alpha_{ijk} + n_{ijk}}{\sum_{k'} (\alpha_{ijk'} + n_{ijk'})}$$

Dove α_{ijk} è un prior mentre n_{ijk} è il conteggio del numero di esempi con $\mathbf{X}_i = t_k$ e $\text{PARENTS}(\mathbf{X}_i) = j$; i è un indice che può assumere un qualsiasi valore tra 1 ed N , numero di variabili, j invece prende valore tra 1 ed il prodotto del numero di livelli dei genitori di \mathbf{X}_i , ed infine k è da prendere tra 1 e d .

Esempio 15.1: Importanza dei dati completamente osservati

Con l'appoggio della rete disegnata sopra adesso facciamo vedere perché è importante l'ipotesi che i dati siano completi. Se così fosse potremmo mettere evidenza in tutte le variabili interne (i.e. esclusi i parametri) e quindi per come sono messe le frecce, il cammino tra un qualsiasi parametro ed ogni altro è bloccato dall'evidenza: ad esempio un cammino tra Θ_B e $\Theta_{S|BC}$ ha delle frecce seriali con evidenza sul nodo centrale B e quindi è bloccato, ed anche il secondo sarebbe bloccato per la medesima ragione da M (motorino di avviamento).

Supponiamo invece che M , B e C non siano osservate, allora in questo caso tutte le variabili che sono rimaste osservate sono dei colliders e quindi fanno passare l'evidenza: i cammini tra Θ_B e $\Theta_{S|BC}$ non sono più bloccati. Questo ci porta a notare che se ci sono delle variabili non osservate allora i parametri non sono più indipendenti tra di loro, date le variabili. Come conseguenza di questo fatto, lo schema di apprendimento che abbiamo visto, fondato sulla fattorizzazione di un problema di massima verosimiglianza in più sottoproblemi indipendenti, smette di funzionare. In questo caso si deve cercare di “riempire gli spazi vuoti” usando le variabili che si sono osservate, ma le tecniche di apprendimento da impiegare si complicano. ■

15.2 Apprendimento della struttura

Il problema che ci si pone adesso è quello di, assegnato un dataset \mathcal{D} , trovare il grafo orientato aciclico che rappresenta al meglio le indipendenze in \mathcal{D} . In questo senso si immagina che la struttura della rete sia a sua volta una variabile aleatoria \mathcal{S} le cui possibili realizzazioni sono i diversi possibili grafi orientati aciclici (che sono in quantità esponenziale rispetto al numero di variabili). Assegnata una candidata struttura s per stimare la sua probabilità, osservato il dataset, si applica banalmente la formula di Bayes:

$$\mathbb{P}(S = s \mid \mathcal{D}) = \frac{\mathbb{P}(\mathcal{D} \mid S = s) \cdot \mathbb{P}(S = s)}{\mathbb{P}(\mathcal{D})}$$

Invece per quanto riguarda l'inferenza, avendo osservato il dataset ed assegnata una nuova tupla \mathbf{X} , si procede in questo modo: poiché non si conosce la struttura si prendono tutte quelle possibili e si scrive una somma pesata di come questa:

$$\mathbb{P}(\mathbf{X} \mid \mathcal{D}) = \sum_S \mathbb{P}(S \mid \mathcal{D}) \cdot \mathbb{P}(\mathbf{X} \mid \mathcal{D}, S) \stackrel{(i)}{=} \sum_S \mathbb{P}(S \mid \mathcal{D}) \cdot \int_{\theta} \mathbb{P}(\mathbf{X} \mid \theta, S) \cdot \mathbb{P}(\theta \mid \mathcal{D}, S) \cdot d\theta$$

Dove l'uguaglianza (i) segue dall'applicare l'equazione 14.2 aggiungendo la struttura al membro condizionante.

Se sapessimo che l'intera distribuzione a posteriori $\mathbb{P}(\mathbf{S} \mid \mathcal{D})$ fosse concentrata in un solo punto, ovvero che ci fosse un'unica struttura S^* con probabilità 1 e tutte le altre avessero probabilità nulla, allora l'inferenza, in accordo alla precedente equazione, si potrebbe fare in questo modo:

$$\mathbb{P}(\mathbf{X} \mid \mathcal{D}) = \int_{\theta} \mathbb{P}(\mathbf{X} \mid \theta, \mathbf{S} = S^*) \cdot \mathbb{P}(\theta \mid \mathcal{D}, \mathbf{S} = S^*) \cdot d\theta$$

E tutta la parte di stima dei parametri si continuerebbe a fare come abbiamo visto ad inizio di capitolo, con le distribuzioni di Dirichlet e con le medesime formule. Ovviamente questa assunzione è molto forte e quindi difficile da soddisfare, ma esiste una strategia, nota come **massimo a posteriori**, che prevede di far finta che l'intera distribuzione sia concentrata nel suo punto di massimo. In questo modo anche gli statistici Bayesiani possono accordarsi per usare una rete unica, e l'idea è alla base di un algoritmo, noto come *algoritmo di Cooper-Herskovitz* (1992). Questo si appoggia su due punti:

- (i) Si definisce una funzione di punteggio; la proposta avanzata dai due autori è la seguente (che non discutiamo):

$$\text{SCORE}(\mathbf{S}) \stackrel{\text{def}}{=} \prod_{i=1}^N \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + n_{ij})} \cdot \prod_{k=1}^{\alpha_i} \frac{\Gamma(\alpha_{ijk} + n_{ijk})}{\Gamma(\alpha_{ijk})}$$

Dove q_i è definito come il numero di genitori di \mathbf{X}_i ; si potrebbe far vedere che questa funzione rappresenta la bontà di adattamento dei dati alla particolare struttura \mathbf{S} .

- (ii) Si applica un algoritmo di ricerca locale per massimizzare $\text{SCORE}(\mathbf{S})$, ad esempio **HILL-CLIMBING**, in cui le possibili azioni sono le seguenti:
 - (a) Si aggiunge un arco orientato tra due variabili che non erano connesse, se questo non genera un ciclo.
 - (b) Si cancella un arco.
 - (c) Si rovescia la direzione di un arco, nuovamente prestando attenzione a non formare cicli.

Osservazione 15.2.1. Dopo aver effettuato una di queste azioni non è necessario ricalcolare da capo l'intero score in quanto una modifica ha effetto solo localmente, e quindi si può sfruttare il valore precedente.

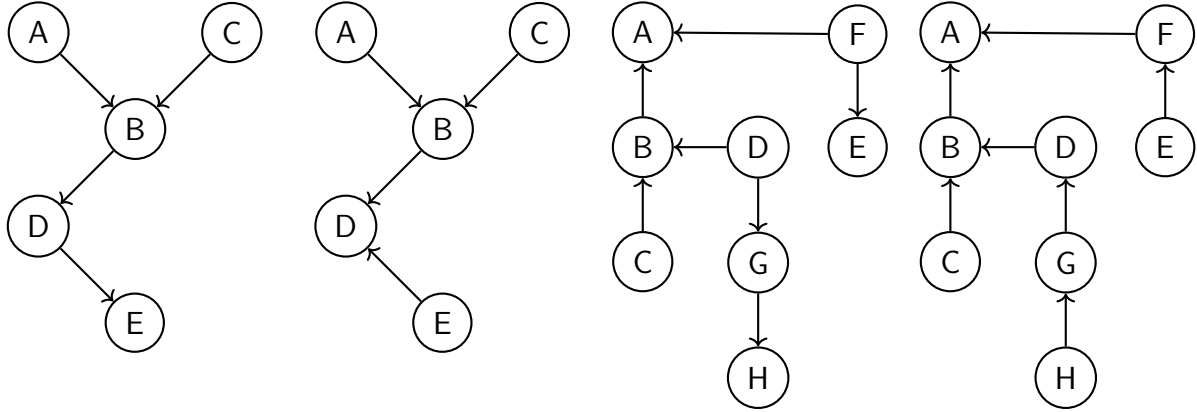
Osservazione 15.2.2. Applicare una di queste tre azioni può modificare la rete localmente ma in maniera da lasciare inalterate le indipendenze condizionali codificate. È quello che succede ad esempio se trasformiamo $\bigcirc \rightarrow \bigcirc \rightarrow \bigcirc$ in $\bigcirc \leftarrow \bigcirc \rightarrow \bigcirc$. Una mossa di questo tipo non modifica il punteggio e quindi è inutile effettuarla.

L'ultima osservazione è interessante ma fino a questo punto non abbiamo visto nessun risultato che ci dica come fare a stabilire se due reti di Bayes corrispondano o meno allo stesso insieme di indipendenze condizionali. Fortunatamente esiste un teorema che ci viene in soccorso:

Teorema 15.2.1. Siano $\mathcal{S}, \mathcal{S}'$ due reti di Bayes, allora sono equivalenti i seguenti fatti:

- (i) $\mathcal{S}, \mathcal{S}'$ corrispondono allo stesso insieme di indipendenze condizionali
- (ii) $\mathcal{S}, \mathcal{S}'$ hanno gli stessi colliders in tutti i cammini.

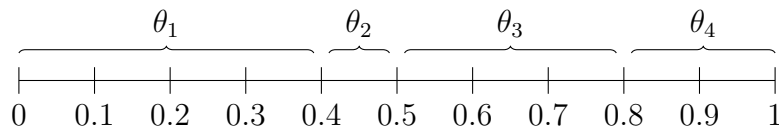
Se è vera l'ipotesi (i) allora le due reti corrispondono alle stesse famiglie di distribuzioni, e se è verificata la (ii) si dice che le due reti hanno la stessa **struttura a V**. Di seguito riportiamo un esempio per chiarire il concetto:



Nella figura qui sopra la coppia di sinistra corrisponde a due reti che non hanno la stessa struttura a V, infatti in quello di destra è presente un collider in più, ovvero D. La coppia di destra ha invece gli stessi colliders in tutti i cammini, ovvero A e B, e pertanto corrispondono alla medesima famiglia di distribuzioni per il teorema appena enunciato e quindi hanno anche lo stesso punteggio. Si può immaginare che una struttura a V definisca una classe di equivalenza, ed ha senso effettuare solo mosse che ci fanno cambiare classe quando si esegue HILL-CLIMBING (stessa classe \implies stesso score).

15.2.1 Campionamento di un dataset

Se volessimo mettere alla prova un algoritmo di apprendimento della struttura potremmo generare un dataset a partire da una struttura nota e verificare se l'algoritmo riesce o meno a ritrovarla. A tal fine illustriamo qui di seguito come procedere. Supponiamo di avere una rete composta da un singolo nodo associato ad una variabile categorica a 4 livelli, con parametri $(\theta_1, \theta_2, \theta_3, \theta_4) = (0.4, 0.1, 0.3, 0.2)$; per campionare questa rete possiamo predisporre l'intervallo $[0, 1]$ ed assegnare a ciascun parametro uno spicchio di questo in maniera proporzionale al suo valore, come si vede nell'immagine:



A questo punto possiamo generare un numero casuale nell'intervallo $[0, 1]$ con probabilità uniforme ed in base a dove cade il numero generato, restituire il valore associato al parametro corrispondente. Di seguito è illustrata la procedura nel caso particolare illustrato sopra:

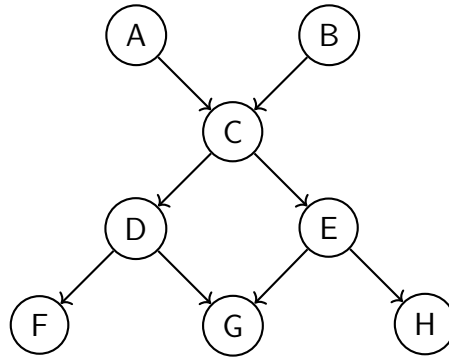
Algoritmo 22 Campionamento di un dataset

```

1:  $r \leftarrow \text{RANDOM}(0,1)$ 
2: if  $r < 0.4$  then
3:   return  $t_1$ 
4: else
5:   if  $r < 0.5$  then
6:     return  $t_2$ 
7:   else
8:     if  $r < 0.8$  then
9:       return  $t_3$ 
10:    else
11:      return  $t_4$ 

```

Assegnato una rete di Bayes generica, possiamo utilizzare questa procedura per campionare i nodi che non hanno genitori, e a partire da questi campionare poi l'intera rete. Prendiamo come esempio la seguente rete per illustrare il ragionamento:



Possiamo campionare A e B usando l'algoritmo 22 e a quel punto, conoscendo queste sarà nota la distribuzione di C, che quindi potrà essere campionato sempre nella stessa maniera. Ciò fatto sarà nota sia la distribuzione di D che di E che potranno essere campionati usando la procedura riportata sopra, ed infine una volta effettuato il campionamento di questi sarà nota anche la distribuzione di F, G e H, che potranno a loro volta essere campionati così; in questo modo si è campionata l'intera rete. In sostanza è sufficiente scegliere un nodo che non abbia genitori come radice e campionare con la procedura riportata sopra seguendo un ordinamento topologico. Il campionamento può essere ripetuto un numero arbitrario di volte per produrre un dataset con un numero di esempi a piacimento.

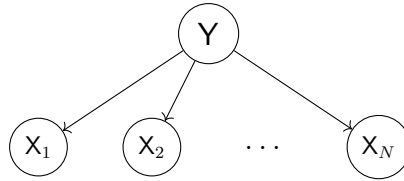
15.3 Classificatore Naive Bayes

Si tratta del primo esempio che vediamo di **algoritmo con supervisione**; tra tutte le variabili di un dataset, X_1, \dots, X_N prevediamo la presenza di una¹ speciale, connotata come Y, che ci interessa predire assumendo che tutte le altre variabili siano note. Un'applicazione tipica è quella di un filtro antispam: le variabili in questo caso sono le parole del vocabolario ($X_i = 1$ se la parola i -esima compare nel messaggio, $X_i = 0$ altrimenti) ed Y è una variabile che vale 1 se il dato messaggio è spam, 0 altrimenti. In

¹In generale potrebbero essercene di più, per semplicità atteniamoci al caso in cui ce ne sia una sola.

questo contesto ci interessa di calcolare la probabilità che un messaggio sia spam, ovvero $\mathbb{P}(Y = 1 \mid X_1, \dots, X_N)$. In termini della rappresentazione del messaggio, se applichiamo l'idea proposta ciascuno di questi sarà un vettore binario estremamente sparso e quindi conviene memorizzarlo come l'insieme delle posizioni in cui esso vale 1. Questa rappresentazione però non riesce a tenere conto del numero di volte in cui le parole compaiono nel messaggio, né in quale ordine; per la prima problematica, una possibile soluzione è quella di memorizzare il messaggio come multiset, ovvero tenere in memoria le coppie (i, n_i) , con i la posizione nel vettore ed n_i la molteplicità della parola. Questo tipo di rappresentazione è noto come **bag of words**.

Il modello grafico che si può usare per risolvere questo problema, noto come **naive bayes**, è il seguente:



Questo esprime il fatto che X_1, \dots, X_N , detti **attributi**, sono tutti condizionalmente indipendenti l'uno dall'altro data Y , detta **classe**. Quest'ipotesi tuttavia è palesemente falsa: nell'esempio del filtro antispam, è chiaro che possono esistere due parole che tendono a comparire insieme nei tentativi di spam e quindi la probabilità di trovare una di questa data la classe ed osservata l'altra parola aumenta. Nonostante tutto il modello tende a comportarsi decentemente bene ed ha alcuni vantaggi rispetto ad altri più sofisticati e ciò lo rende non obsoleto. Predire la classe è inoltre molto semplice, infatti applicando la formula di Bayes viene che

$$\mathbb{P}(Y \mid X_1, \dots, X_N) \propto \mathbb{P}(X_1, \dots, X_N \mid Y) \cdot \mathbb{P}(Y) \stackrel{(i)}{=} \left[\prod_{j=1}^N \mathbb{P}(X_j \mid Y) \right] \cdot \mathbb{P}(Y)$$

Dove l'uguaglianza (i) vale per la fattorizzazione del grafo. In questo modo effettuare la classificazione è estremamente veloce, ed anche la stima dei parametri si può fare in un tempo buono; in questo caso i parametri sono $\theta_{jk} := \mathbb{P}(X_j = 1 \mid Y = k)$ ed $\eta_k := \mathbb{P}(Y = k)$, che possiamo stimare in massima verosimiglianza come segue:

$$\hat{\theta}_{jk} = \frac{\sum_{i=1}^n x_j^{(i)} \cdot \mathbb{1}\{y^{(i)} = k\}}{\sum_{i=1}^n \mathbb{1}\{y^{(i)} = k\}}$$

Dove $x_j^{(i)}$ vale 0 oppure 1 e quindi il numeratore conta quanti esempi nel dataset hanno classe k ed $x_j^{(i)} = 1$, mentre il denominatore conta gli esempi di classe k . Si noti qui che invece di memorizzare il rapporto si memorizzassero separatamente numeratore e denominatore potremmo aggiornare il modello in tempo $O(1)$ quando si osservano degli esempi nuovi aggiungendo +1 nei posti giusti ed usando una tabella hash; questa è una caratteristica estremamente desiderabile, ad esempio se si contrassegna un nuovo messaggio come spam non si dovrebbe aspettare il tempo di addestrare un modello per averlo aggiornato. Invece per quanto riguarda il secondo tipo di parametro, lo si stima

come la proporzione di esempi di classe k :

$$\hat{\eta}_k = \frac{1}{n} \cdot \sum_{i=1}^n \mathbb{1} \{y^{(i)} = k\}$$

Il modello, nonostante tutti questi vantaggi, presenta due notevoli problematiche, che presentiamo di seguito e discutiamo più in dettaglio nella prossima lezione:

- (i) La predizione $\mathbb{P}(Y | \mathbf{X}_1, \dots, \mathbf{X}_N)$ non è affidabile, i.e. non possiamo fidarci del valore ottenuto per questa probabilità. In ogni caso il modello è abbastanza affidabile per decidere se $\mathbb{P}(Y | \mathbf{X}_1, \dots, \mathbf{X}_N) > 0.5$ o meno;
- (ii) Esistono dei casi in cui il modello non può predire perché nello stimare i parametri escono delle forme $\frac{0}{0}$

Il motivo di (i) è piuttosto evidente, il fatto di considerare gli attributi come condizionalmente indipendenti data la classe porta a sovrastimare o sottostimare brutalmente quella probabilità, ad esempio nel contesto del filtro antispam, se ci sono due parole che co-occorrono frequentemente nei messaggi spam, il considerarle indipendenti spinge più fortemente verso la classe $Y = 1$ perché il modello non è in grado di tenere conto della loro co-occorrenza. Per quanto riguarda invece il punto (ii), vedremo in quali situazioni si verifica e che si può porre rimedio nella lezione successiva.

23/11/2020

16.1 Laplace smoothing per Naive Bayes

Introduciamo un breve esempio per mostrare una situazione in cui un classificatore Naive Bayes non riesce a predire per la comparsa di forme $\frac{0}{0}$.

Esempio 16.1: Fallimento di Naive Bayes

Sia X lo spazio dei documenti e siano $\{1, 2, \dots, k\}$ le possibili categorie, allora la funzione di predizione che si usa con Naive Bayes è la $f: X \mapsto \{1, 2, \dots, k\}$ fatta in questo modo:

$$f(x) \stackrel{\text{def}}{=} \text{MAX}_k \left\{ \eta_k \cdot \prod_{j=1}^n \theta_{jk}^{x_j} \cdot (1 - \theta_{jk})^{1-x_j} \right\}$$

L'argomento del massimo è la probabilità di avere un documento di classe k con la data assenza o presenza di parole. Può capitare talvolta che tutti gli argomenti siano nulli ed in quel caso non si riesce a predire: per esempio assumiamo $k = 2$ e possibili categorie $\{\text{scienza}, \text{musica}\}$ e supponiamo di sapere che la parola *chitarra* non compaia mai nei documenti di categoria *scienza* mentre la parola *segnale* non presenzi mai nei documenti a tema *musica*. Allora:

$$(i) \quad \theta_{\text{chitarra}, \text{scienza}} = 0$$

$$(ii) \quad \theta_{\text{segnale}, \text{musica}} = 0$$

Se arrivasse da classificare un documento x contenente sia *chitarra* che *segnale* tutti e due gli argomenti del MAX sarebbero nulli e questo porterebbe al fallimento della predizione. ■

Fortunatamente è possibile porre rimedio a questa situazione critica adottando una semplice strategia nota come **Laplace smoothing**, che consiste nell'assumere di aver osservato almeno una volta ogni parola in ciascuna categoria di documenti ed almeno un documento per ogni classe¹. Con questa semplice modifica la stima del parametro θ_{jk} cambia in questo modo:

$$\hat{\theta}_{jk} = \frac{\left(\sum_{i=1}^n x_j^{(i)} \cdot \mathbb{1} \{y^{(i)} = k\} \right) + 1}{\left(\sum_{i=1}^n \mathbb{1} \{y^{(i)} = k\} \right) + 2}$$

In generale al denominatore avremmo un $+k$ piuttosto di $+2$ se k è il numero di classi. La stima del parametro η_j invece diventa la seguente:

$$\hat{\eta}_j = \frac{1 + \left(\sum_{i=1}^n \mathbb{1} \{y^{(i)} = j\} \right)}{k + n}$$

¹In altre parole, per ogni categoria si finge di aver visto un documento che vi appartiene contenente tutte le parole del dizionario.

Notasi adesso che con Laplace smoothing $\hat{\eta}_k > 0$ e $\hat{\theta}_{jk} > 0$ quindi la situazione vista nell'esempio e che portava al fallimento non si può più verificare: gli argomenti del MAX sono tutte prodotti di quantità positive ed in quanto tali saranno positivi.

16.2 Classificatore Naive Bayes multinomiale

Il modello visto fino a questo momento è detto anche “di Bernoulli” in quanto ogni parola è una variabile di Bernoulli per ciascuna classe. Il processo generativo di un esempio sarebbe stato il seguente:

Algoritmo 23 Processo generativo modello di Bernoulli

```

1:  $y \leftarrow \text{CATEGORICA}(\eta)$ 
2: for all  $j \in \{1, \dots, N\}$  do
3:    $x_j \leftarrow \text{BERNOULLI}(\theta_{jy})$ 

```

In questo modello i documenti erano considerati come insiemi di parole e quindi non si riusciva a tenere conto del numero di occorrenze di queste. Per superare questo limite si abbandona il modello di Bernoulli a favore di uno basato su multisets di parole, detti *bags*: la sostanziale differenza rispetto a prima è che ogni elemento dell'insieme adesso viene contato con la propria molteplicità, e si parla di **multinomial naive Bayes**. Riportiamo di seguito il processo generativo per un esempio:

Algoritmo 24 Processo generativo modello multinomiale

```

1:  $y \leftarrow \text{CATEGORICA}(\eta)$ 
2:  $L \leftarrow \text{POISSON}(\lambda)$ 
3: for all  $t \in \{1, \dots, L\}$  do
4:    $x_t \leftarrow \text{CATEGORICA}(\theta_y)$ 

```

Dapprima si campiona la categoria del documento y da una distribuzione categorica e poi la sua lunghezza L espressa in numero di parole, infine per ogni possibile posizione t nel documento si campiona la parola che la occuperà da una categorica che ha come possibili realizzazioni le parole del vocabolario. Talvolta piuttosto di campionare x_j da una distribuzione categorica lo si prende da una *multinomiale*, i.e. $x_j \leftarrow \text{MULTINOMIALE}(L, \theta_{1y}, \dots, \theta_{Ny})$, che è una distribuzione fatta in questo modo:

Definizione 16.1 (Distribuzione Multinomiale). Sia \mathbf{X} una variabile aleatoria discreta con $\mathbf{X}(\Omega) = \{t_1, t_2, \dots, t_L\}$, allora si dice che \mathbf{X} ha distribuzione **multinomiale** di parametri $L, \theta_1, \dots, \theta_L$, e si scrive che $\mathbb{P}_{\mathbf{X}} \sim \text{MULT}(L, \theta_1, \dots, \theta_N)$, se ha densità

$$\mathbb{P}(\mathbf{X}) = \left[\mathbb{1}_{\{|x|=L\}} \cdot \frac{L!}{x_1!x_2! \dots x_L!} \right] \prod_{j=1}^N \theta_j^{x_j}$$

Nella definizione soprastante la funzione indicatrice si legga come “quando la lunghezza del documento è L ”, e se L si considera fissata allora il prodotto tra parentesi quadre è semplicemente un fattore costante che non impatta nella predizione. Nel modello

multinomiale la stima dei parametri θ_{jk} in massima verosimiglianza si fa così:

$$\hat{\theta}_{jk} = \frac{\sum_{i=1}^n x_j^{(i)} \cdot \mathbb{1}\{y^{(i)} = k\}}{\sum_{r=1}^N \sum_{i=1}^n x_r^{(i)} \cdot \mathbb{1}\{y^{(i)} = k\}}$$

Dove $x_j^{(i)}$ a differenza di prima non vale più 0 oppure 1 ma ha per valore la molteplicità della parola j nel documento i -esimo. Il denominatore invece conta il numero di parole nei documenti di classe k (il conteggio è cumulativo) e c'è come vincolo aggiuntivo che per ogni categoria k si abbia $\sum_{j=1}^N \theta_{jk} = 1$. Questa è una notevole differenza rispetto alla rappresentazione come *bags of words*, infatti diversamente a quanto accadeva in precedenza adesso, per effetto di questa condizione, se aumenta la probabilità di una parola deve calare quella di qualcun'altra. Grazie all'acquisita capacità di contare le parole con le rispettive molteplicità il modello risultante è più accurato.

Il Laplace smoothing agisce in questo modo sul Naive Bayes multinomiale:

$$\hat{\theta}_{jk} = \frac{1 + \sum_{i=1}^n x_j^{(i)} \cdot \mathbb{1}\{y^{(i)} = k\}}{N + \sum_{r=1}^N \sum_{i=1}^n x_r^{(i)} \cdot \mathbb{1}\{y^{(i)} = k\}}$$

Non cambia invece la stima di η_k .

16.2.1 Punto di break-even

Adesso vedremo una metrica che consente di quantificare la bontà di un classificatore. Una volta addestrato uno di questi, dato un dataset di prova detto *training set*, si possono calcolare un certo numero di parametri, ad esempio, immaginando un classificatore binario si potrebbe contare il numero di esempi classificati correttamente come di classe positiva, ed allo stesso modo si possono contare anche i veri negativi, falsi positivi e falsi negativi per stilare una matrice come questa:

		Classe vera	
		1	0
Classe	1	Veri positivi	Falsi positivi
Predetta	0	Falsi negativi	Veri negativi

Questa viene chiamata anche **matrice di contingenza**. A partire da questa si possono definire varie metriche, e la prima che vediamo è la cosiddetta **accuratezza** (*accuracy*), definita come segue:

$$A \stackrel{def}{=} \frac{\text{Veri positivi} + \text{Veri negativi}}{\text{Veri positivi} + \text{Veri negativi} + \text{Falsi positivi} + \text{Falsi negativi}}$$

Idealmente un classificatore non dovrebbe sbagliare e pertanto tutti gli elementi fuori dalla diagonale principale della tabella di contingenza dovrebbero essere nulli, con l'accuratezza che varrebbe quindi 1. In casi non ideali tuttavia commettere errori è inevitabile e quindi

si cercherà di ottenere una accuracy quanto più vicina possibile a tale valore. Un'altra metrica che può essere definita è la **precisione** (*precision*), data da

$$P \stackrel{def}{=} \frac{\text{Veri positivi}}{\text{Veri positivi} + \text{Falsi positivi}}$$

Ovvero è il rapporto tra veri positivi e predetti positivi. Il valore di questa metrica è indicativo di quanto ci si possa fidare del classificatore quando predice positivo, con il caso ideale nuovamente dato da $P = 1$. Esiste tuttavia un modo semplice per ottenere una precision elevata senza che il classificatore si comporti in maniera desiderabile: è sufficiente fare in modo che predica poco la classe positiva e lo faccia in maniera conservativa, ovvero solo quando è particolarmente sicuro. Questo exploit però fa aumentare il numero di falsi negativi rendendo inaffidabile il classificatore quando esso predice questa classe, e quindi piuttosto di considerare il solo valore P si considera questo in coppia con un'altra metrica detta **richiamo** (*recall*), definita come segue:

$$R \stackrel{def}{=} \frac{\text{Veri positivi}}{\text{Veri positivi} + \text{Falsi negativi}}$$

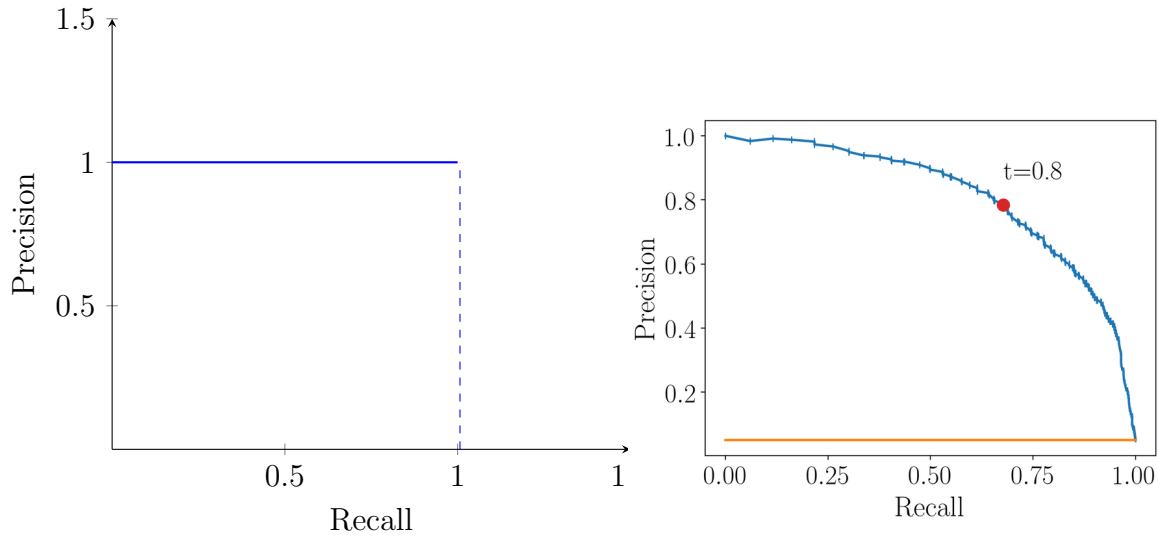
Come prima, anche in questo caso è facile costruire un classificatore capace di mantenere una recall elevata, i.e. vicina ad 1, infatti basta che questo predica poco la classe negativa e lo faccia solo quando è piuttosto sicuro, quindi ancora una volta il solo valore di R non è molto significativo per misurare la bontà di un classificatore. Ciò che è difficile invece è creare un classificatore che abbia sia P che R elevati.

Per discriminare tra classe positiva e negativa possiamo definire una funzione di predizione fatta in questo modo:

$$f(x) := \ln \left(\frac{\mathbb{P}(Y = 1 \mid X)}{\mathbb{P}(Y = 0 \mid X)} \right) \quad (16.1)$$

Si può facilmente mostrare che se $f(x) > 0$ allora $\mathbb{P}(Y = 1 \mid X) > 0.5$ e quindi se f è maggiore di zero si predice la classe positiva, altrimenti si sta predicando la classe negativa. Più in generale $f(x)$ può essere modificata in maniera tale che, fissato un parametro $t \in \mathbb{R}$, si predica la classe positiva quando $f(x) > t$ e quella negativa quando $f(x) \leq t$ e si definisce così una famiglia di predittori, ciascuno dei quali avrà un suo valore di precision e recall. In linea di principio possiamo immaginare che se t è piccolo sia facile che $f(x) > t$ e quindi il predittore corrispondente abbia una precision alta ma una bassa recall, mentre con t grandi sarà più facile che $f(x) < t$ e quindi al contrario si otterrà un classificatore con alta recall ma bassa precision.

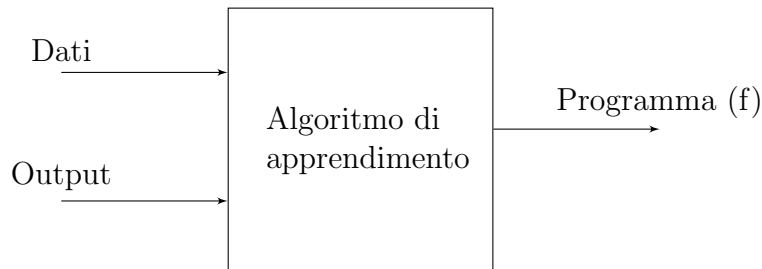
A questo punto è possibile tracciare un grafico in cui si riporta la cosiddetta **curva precision-recall** della famiglia: questa è una curva parametrica che dipende da $t \in \mathbb{R}$: per ciascuno di questi si considera il predittore corrispondente e si calcolano i suoi valori di precision e recall, e si riportano sul grafico. A priori per tracciarla dovremmo far variare t in tutto \mathbb{R} , che è un insieme non numerabile; in realtà possiamo farlo variare anche sui soli valori di $f(x)$ sul test set, ma in quel caso il grafico ottenuto non è più una curva continua. Il **punto di break-even** viene definito come l'intersezione tra la curva precision-recall e la retta bisettrice del primo quadrante. Di seguito riportiamo due immagini: quella di sinistra rappresenta la curva nel caso ideale, mentre quella di destra in un caso pratico, con il punto di break-even evidenziato in rosso. Si noti che a priori non è garantita la monotonicità della curva, diversamente da quanto si potrebbe pensare.



Se tracciamo la curva facendo variare t nell'immagine di $f(x)$ sul test set tuttavia come già detto si ottiene un grafico composto da punti e non c'è garanzia che in effetti la bisettrice del primo quadrante intersechi uno di questi. Si accetta come approssimazione di prendere come punto di break-even l'intersezione tra la retta e la curva che interpola i due punti ad essa più vicini. Infine, specifichiamo che il valore associato al break-even è tipicamente l'ordinata del break-even point, e tanto più il valore è vicino ad 1 quanto più il classificatore è buono, essendo più vicino al caso ideale.

Rimandiamo a [MN98] per un confronto di prestazioni tramite punto di break-even tra modello naive Bayes multinomiale e di Bernoulli.

16.3 Apprendimento supervisionato



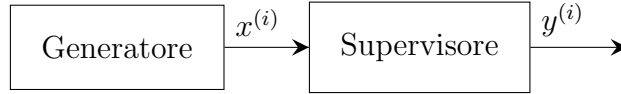
Un algoritmo di apprendimento è una funzione che mappa datasets in funzioni, dove un dataset è un insieme \mathcal{D} della forma seguente:

$$\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid i = 1, 2, \dots, n\}$$

Ogni coppia si dice essere un **esempio**, il cui primo elemento viene detto **input** ed il secondo **output**, e quest'ultimo rappresenta la vera classe dell'input; la sua presenza è ciò che motiva il fatto che questi algoritmi vengano chiamati “supervisionati”.

Assumeremo che esista un insieme \mathcal{X} chiamato **spazio delle istanze** che contenga tutti i possibili input, i.e. $x^{(i)} \in \mathcal{X}$ per ogni i valido e questo può essere un insieme qualsiasi. Riguardo all'output, se prende valori in $\{0, 1\}$ si parla di *classificazione binaria*,

ma più in generale anche $y^{(i)}$ potrebbe stare in un insieme qualunque e se esso prende valori in \mathbb{R} si parla di *regressione*. Dobbiamo immaginare che gli esempi provengano da un processo generativo come quello dello schema:



Il generatore produce inputs da una certa distribuzione di probabilità \mathbb{P}_X ed il supervisore assegna un'etichetta all'input secondo una ulteriore distribuzione di probabilità $\mathbb{P}_{Y|X}$; si viene a formare in questo modo una distribuzione di probabilità congiunta $\mathbb{P}_{X,Y} = \mathbb{P}_X \mathbb{P}_{Y|X}$ che è fissata e sconosciuta. Un algoritmo di apprendimento è una mappa $\mathcal{D} \mapsto f$ con f una funzione con la seguente firma: $f: X \mapsto Y$. Lo spazio \mathcal{F} delle possibili funzioni però può essere enormemente grande anche in casi semplici, per esempio le $f: \{0, 1\}^n \mapsto \{0, 1\}$ sono 2^{2^n} e quindi molto spesso ha senso cercare di limitare le possibili funzioni imponendo dei vincoli; si ottiene quindi una classe di funzioni $\mathcal{H} \subseteq \mathcal{F}$ detta *classe d'ipotesi*.

Per portare un esempio, nel caso della classificazione binaria si può cercare di prendere la funzione che minimizza il numero atteso di errori di classificazione, i.e.

$$f = \underset{h \in \mathcal{H}}{\text{MIN}} \left\{ \mathbb{E} \left[\mathbb{1}_{\{h(x) \neq y\}} \right] \right\} \stackrel{\text{def}}{=} \underset{h \in \mathcal{H}}{\text{MIN}} \left\{ \int_{X \times Y} \mathbb{1}_{\{h(x) \neq y\}} \cdot \rho(x, y) dx dy \right\}$$

L'idea però non è applicabile nella pratica in quanto calcolare il valore atteso richiederebbe di conoscere la distribuzione $\mathbb{P}_{X,Y}$ che abbiamo già detto essere non nota. Il problema di apprendimento con supervisione così formulato è un problema di ottimizzazione di una funzione non osservata e quindi il problema non può essere risolto. Come suo surrogato si decide invece di risolvere questo:

$$f = \underset{h \in \mathcal{H}}{\text{MIN}} \left\{ \frac{1}{n} \cdot \sum_{i=1}^n \mathbb{1}_{\{h(x^{(i)}) \neq y^{(i)}\}} \right\}$$

Si parla allora di **minimizzazione del rischio empirico**: in sostanza tra tutte le possibili funzioni di classificazione si cerca quella che minimizza l'errore sul training set nella speranza che questo porti sempre a fare bene; se \mathcal{H} è un insieme finito si possono enumerare tutte le possibili funzioni ed una per una calcolare l'argomento corrispondente nel MAX, invece se così non fosse si cerca di ricondursi ad un problema di ottimizzazione nel continuo con la speranza che abbia un minimo assoluto facile da identificare. I vari algoritmi di apprendimento supervisionato si possono distinguere per come viene calcolato il minimo, per come viene scelto \mathcal{H} o per come viene deciso se $h(x^{(i)}) \neq y^{(i)}$.

16.3.1 Formalizzazione di Naive Bayes

Quando si è introdotto Naive Bayes abbiamo già accennato che si trattasse di un algoritmo di apprendimento supervisionato. Adesso vogliamo entrare nel dettaglio su come funziona la sua predizione, che abbiamo già riportato nell'equazione 16.1. Per esplicito:

$$f(x) = \ln \left(\frac{\mathbb{P}(Y = 1) \cdot \prod_{j=1}^N \mathbb{P}(X_j | Y = 1)}{\mathbb{P}(X = 0) \prod_{j=1}^N \mathbb{P}(X_j | Y = 0)} \right) = \ln \left(\frac{\hat{\eta}_1 \cdot \prod_{j=1}^N \hat{\theta}_{j1}^{x_j} \cdot (1 - \hat{\theta}_{j1})^{1-x_j}}{\hat{\eta}_0 \cdot \prod_{j=1}^N \hat{\theta}_{j0}^{x_j} \cdot (1 - \hat{\theta}_{j0})^{1-x_j}} \right)$$

Sfruttando le proprietà elementari dei logaritmi è possibile scomporre tutto questo nella somma di vari logaritmi naturali con coefficienti opportuni; facendo quindi dei calcoli elementari si arriva a riscrivere la funzione di predizione nel modo seguente:

$$f(x) = \ln \left(\frac{\hat{\eta}_1}{\hat{\eta}_0} \right) + \sum_{j=1}^N x_j \cdot \ln \left(\frac{\hat{\theta}_{j1}}{\hat{\theta}_{j0}} \right) + \sum_{j=1}^N (1 - x_j) \cdot \ln \left(\frac{1 - \hat{\theta}_{j1}}{1 - \hat{\theta}_{j0}} \right) = b + \langle \underline{w}, \underline{x} \rangle$$

Dove nell'ultima uguaglianza b è uno scalare mentre \underline{w} è un vettore che vive in \mathbb{R}^N , ed i due oggetti matematici sono così definiti:

$$b \stackrel{\text{def}}{=} \ln \left(\frac{\hat{\eta}_1}{\hat{\eta}_0} \right) + \sum_{j=1}^N \ln \left(\frac{1 - \hat{\theta}_{j1}}{1 - \hat{\theta}_{j0}} \right) \quad w_j \stackrel{\text{def}}{=} \ln \left(\frac{\hat{\theta}_{j1}}{\hat{\theta}_{j0}} \right) - \ln \left(\frac{1 - \hat{\theta}_{j1}}{1 - \hat{\theta}_{j0}} \right)$$

Di conseguenza se $\underline{x} \in \mathbb{R}^N$ la funzione di predizione $f(\underline{x}) = b + \langle \underline{w}, \underline{x} \rangle$ è lineare e quindi rappresenta un iperpiano; precisiamo in ogni caso che non tutti gli iperpiani possono essere prodotti come funzioni di classificazione, infatti c'è da precisare che b e \underline{w} non possono essere qualsiasi ma devono sottostare al vincolo che $\hat{\eta}_k$ e θ_{jk} siano tutte delle probabilità. Utilizzando questa riformulazione per $f(x)$ il problema di minimizzazione del rischio empirico si può riscrivere così: si prende come classe d'ipotesi $\mathcal{H} = \{f: f(\underline{x}) = b + \langle \underline{w}, \underline{x} \rangle \mid \underline{w} \in \mathbb{R}^N, b \in \mathbb{R}\}$ e si sceglie f come indicato sotto:

$$\hat{\underline{w}}, \hat{b} = \underset{\underline{w}, b}{\text{MIN}} \left\{ \frac{1}{n} \cdot \sum_{k=1}^n \mathbb{1} \{ S(b + \langle \underline{w}, \underline{x}^{(i)} \rangle) \neq y^{(i)} \} \right\}$$

Dove S è la funzione gradino di Heaviside, data da $S(t) = \frac{1}{2}(\text{SGN}(t) + 1)$. In generale il problema è intrattabile ma diventa facile in un caso fortunato, che è quello in cui il dataset sia *linearmente separabile*.

Definizione 16.2 (Lineare separabilità). Un dataset \mathcal{D} si dice essere **linearmente separabile** se esiste un iperpiano che mantiene tutti gli esempi positivi in un semispazio e tutti quelli negativi nel rimanente.

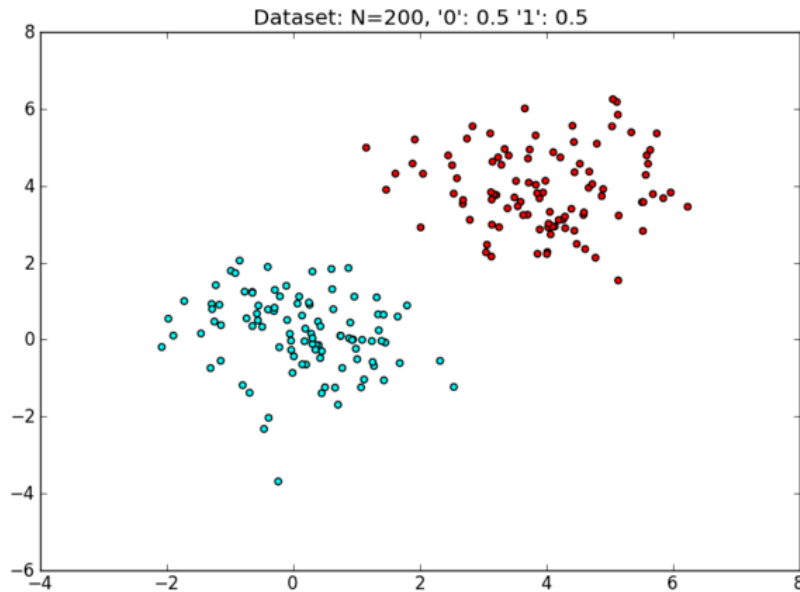


Figura 16.1: Un dataset linearmente separabile

26/11/2020

17.1 Perceptron

Perceptron è un algoritmo di apprendimento supervisionato che risolve il problema della classificazione binaria; il formato con cui si aspetta i dati in ingresso è il seguente:

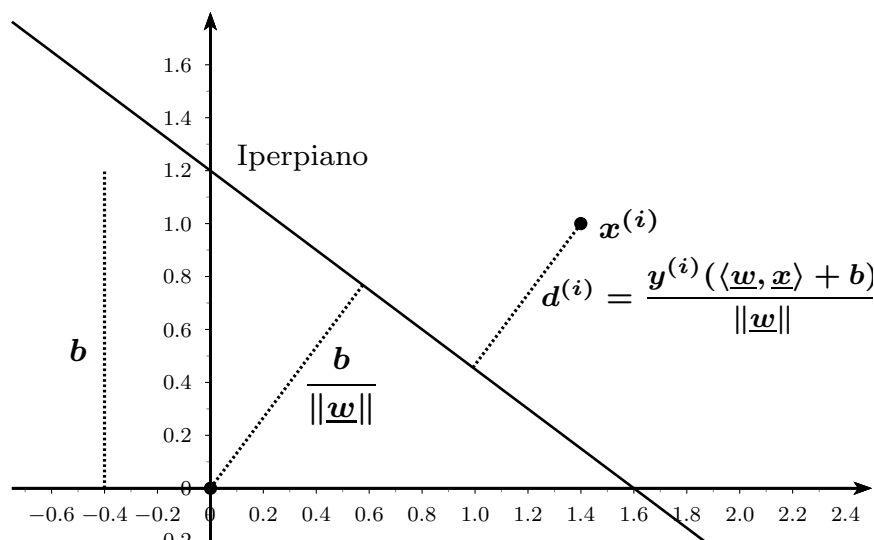
$$\mathcal{D} = \{ (x^{(i)}, y^{(i)}) , i = 1, 2, \dots, n \}$$

Dove $x^{(i)} \in \mathbb{R}^p$ mentre diversamente da quanto fatto sino ad ora, solo per ottenere delle semplificazioni dal punto di vista della matematica, poniamo $y^{(i)} \in \{-1, 1\}$ piuttosto che $\{0, 1\}$. La classe d'ipotesi di Perceptron è composta da iperpiani, i.e.

$$\mathcal{H} = \left\{ f: \mathbb{R}^p \mapsto \{-1, 1\} \mid f(x) = \text{SGN}(\langle \underline{w}, \underline{x} \rangle + b), \underline{w} \in \mathbb{R}^p, b \in \mathbb{R} \right\}$$

E l'algoritmo ha alcuni importanti vantaggi, come la semplicità, il fatto che ci sia un solo iperparametro da impostare ed è interpretabile; quest'ultima caratteristica è estremamente importante, tanto da essere in alcune applicazioni un requisito di legge, e consiste nella facilità di comprendere il motivo della scelta operata da parte dell'algoritmo.

Nel caso ideale si vorrebbe che $y^{(i)}(\langle \underline{w}, \underline{x} \rangle + b)$ fosse positivo per ogni esempio i , infatti $y^{(i)} \in \{-1, 1\}$ ed il segno di $\langle \underline{w}, \underline{x} \rangle + b$ è proprio la classe predetta, per cui la positività di quel prodotto riflette il fatto che la classe predetta coincida con quella effettiva.



Assegnato un punto $(x^{(i)}, y^{(i)})$ definiamo il suo **margin geometrico** rispetto all'iperpiano $b + \langle \underline{w}, \underline{x} \rangle$, connotato come $d^{(i)}$ nel modo indicato in figura, e rappresenta la distanza con segno del punto rispetto all'iperpiano. Oltre a questo definiamo anche un **margin funzionale** del punto, indicato come $\gamma^{(i)}$, nel modo seguente

$$\gamma^{(i)} \stackrel{def}{=} y^{(i)} (b + \langle \underline{w}, \underline{x} \rangle)$$

Questo concetto si può estendere facilmente anche ad un intero dataset nel modo ovvio, definendo il margine funzionale di un dataset rispetto ad un iperpiano come il più piccolo tra i margini in \mathcal{D} , ed ancora più in generale astraendo dal particolare iperpiano:

$$\gamma := \text{MAX}_{\underline{w}, b} \left\{ \text{MIN}_i \{ y^{(i)} (\langle \underline{w}, \underline{x}^{(i)} \rangle + b) \} \mid \|\underline{w}\| = 1 \right\}$$

Il margine funzionale di un dataset dà una misura di quanto siano distanziati come minimo gli esempi positivi da quelli negativi, e quindi anche quanto siano facili da classificare (tanto più γ è grande, quanto più è facile). Se esiste un iperpiano rispetto al quale γ è positivo il dataset è linearmente separabile e questo garantisce che la classificazione sia un problema facile; se non è verificata questa condizione si potrebbe comunque cercare l'iperpiano che tra tutti commette il minimo numero di errori, ma il problema di individuarlo è NP-hard.

Uno pseudocodice per PERCEPTRON può essere trovato di seguito. Si precisa inoltre che l'algoritmo assume che il dataset sia linearmente separabile.

Algoritmo 25 Classificazione binaria con dataset linearmente separabile

```

1: function PERCEPTRON( $\mathcal{D}$ )
2:    $\underline{w}_0 \leftarrow \underline{0}$ 
3:    $b_0 \leftarrow 0$ 
4:    $k \leftarrow 0$  ▷ Numero di errori accumulati
5:    $R \leftarrow \text{MAX}_i \|\underline{x}^{(i)}\|$ 
6:   do
7:      $\text{n\_err} \leftarrow 0$ 
8:     for  $i = 1, 2, \dots, n$  do ▷ Un'epoca
9:       if  $y^{(i)} (b_k + \langle \underline{w}_k, \underline{x}^{(i)} \rangle) \leq 0$  then
10:         $\underline{w}_{k+1} \leftarrow \underline{w}_k + y^{(i)} \underline{x}^{(i)}$  ▷ Legge di aggiornamento parametri
11:         $b_{k+1} \leftarrow b_k + y^{(i)} R^2$ 
12:         $k \leftarrow k + 1$ 
13:       $\text{n\_err} \leftarrow \text{n\_err} + 1$ 
14:   while  $\text{n\_err} \neq 0$ 
15:   return  $(\underline{w}_k, b_k)$ 

```

Qui sopra k rappresenta il numero di errori accumulati nel corso di tutto l'algoritmo mentre n_err rappresenta quelli nell'*epoca* attuale, ed R è il raggio della più piccola sfera con centro nell'origine che contiene tutti i punti del dataset. L'algoritmo è strutturato in **epoche**, ciascuna delle quali è una intera passata sul dataset e termina dopo che nel corso di un epoca il classificatore ha commesso 0 errori di classificazione. Durante ciascuna epoca si provano a classificare i vari punti del dataset e se su un particolare punto non si sbaglia classificazione allora non si fa niente, mentre se il classificatore commette un errore si cerca di aggiustare l'iperpiano. Nello specifico, se un esempio era positivo ($y^{(i)} = 1 \implies x^{(i)} y^{(i)} = x^{(i)}$) ed è stato classificato come negativo, allora si cerca di far diventare più grande il valore di $\langle \underline{w}, \underline{x} \rangle + b$ sommando $x^{(i)}$ al precedente valore di \underline{w} e sommando R^2 al vecchio valore di b ; analogamente se l'esempio era negativo e lo si è classificato come positivo si cerca di far calare il prodotto scalare. Si noti che aggiornare i parametri in questo modo può portare a sbagliare predizione su un esempio che prima veniva classificato bene, ed è per questo che è richiesto di eseguire più di un'epoca.

L'algoritmo però in questa forma non dà garanzia di terminare in tempo finito, e quindi si può prevedere come ulteriore criterio di arresto l'introduzione di una soglia massima k_{\max} di iterazioni da eseguire prima di terminare, aggiungendo la condizione $k < k_{\max}$ al do-while della riga 14.

Per una intuizione geometrica supponiamo di avere come primo esempio $x^{(1)} = (0.4, 0.8)$ ed $y^{(1)} = -1$ e per semplicità assumiamo che $R = 1$, allora si ottiene applicando la legge di aggiornamento dei parametri di PERCEPTRON:

$$w_1 = 0 - (0.4, 0.8)$$

$$b_1 = 0 - 1$$

E l'iperpiano corrispondente sarà $i_1 : -0.4x - 0.8y - 1 = 0$ che è raffigurato in rosso nella figura sottostante; si noti che questo è perpendicolare al vettore applicato nell'origine con secondo estremo in $x^{(1)}$, e che adesso lo stesso $x^{(1)}$ si trova nel semispazio negativo definito dall'iperpiano i_1 ¹. Supponiamo adesso che arrivi un secondo punto $x^{(2)} = (0.4, -0.8)$ con $y^{(2)} = +1$: usando l'iperpiano i_1 si classificherebbe taluno come di classe negativa sbagliando, pertanto si riapplica la legge di aggiornamento ed ottiene un iperpiano che è perpendicolare ad $\underline{x}_1 + \underline{x}_2$; un iperpiano plausibile è riportato in viola nella figura. Nota che con questa mossa si rischia di sbagliare a classificare $x^{(1)}$, non c'è garanzia di classificare bene entrambi dopo l'aggiornamento come già suggerito in precedenza.

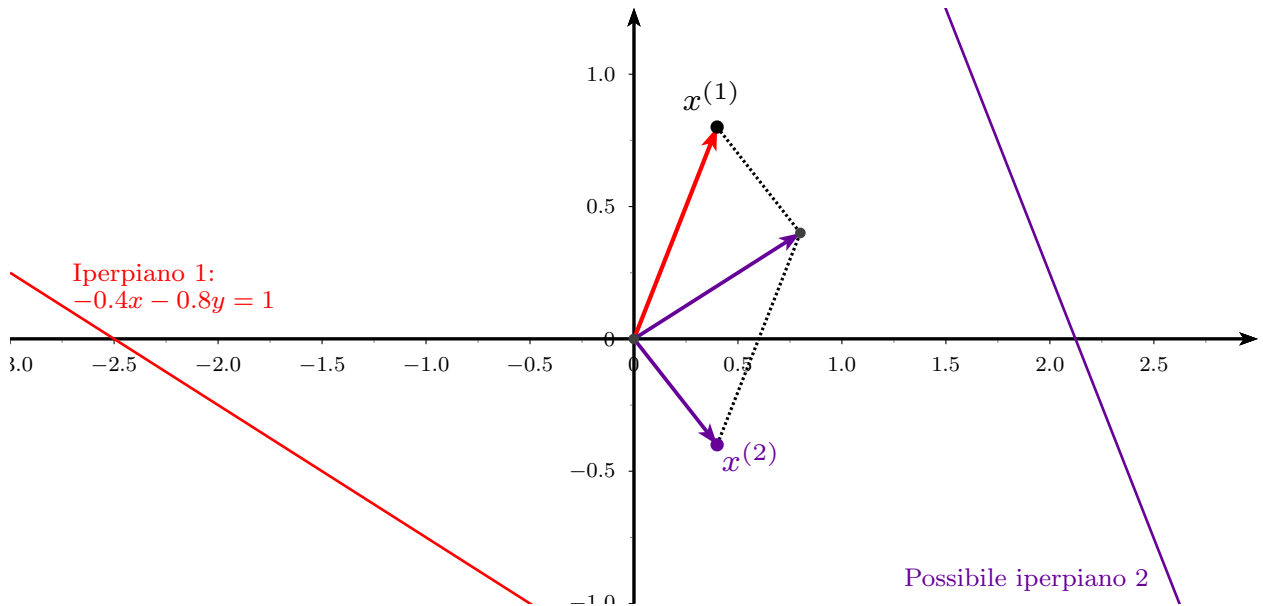


Figura 17.1: Intuizione geometrica di PERCEPTRON

17.1.1 Convergenza

A priori non c'è garanzia di raggiungere prima o poi l'istruzione di return, e di conseguenza nessuna garanzia che l'algoritmo riesca a produrre una funzione di predizione. Fortunatamente sotto alcune ipotesi che sono spesso verificate esiste un teorema che ci dà delle garanzie in questo senso. Tale teorema fu formulato in maniera indipendente da due matematici russi, Block e Nobikoff, nel 1962, e riportiamo il suo enunciato di seguito:

¹Per verificarlo basta constatare che sostituendo $(0, 0)$ nell'equazione dell'iperpiano si ottiene un numero negativo.

Teorema 17.1.1 (Teorema di Block-Nobikoff). Sia \mathcal{D} un dataset non banale, i.e. contenente almeno un esempio positivo ed uno negativo, e sia $R := \max_i \|x^{(i)}\|$, allora se sono verificate le seguenti ipotesi:

- (i) $\exists \underline{w}^* \mid \|\underline{w}^*\| = 1$
- (ii) $\exists b^* \in \mathbb{R}, \gamma > 0 \mid \forall i = 1, 2, \dots, n \quad y^{(i)} (\langle \underline{w}^*, \underline{x}^{(i)} \rangle + b^*) \geq \gamma$

Si dimostra che il numero k di aggiornamenti dei parametri verifica questa disuguaglianza:

$$k < \left(\frac{2R}{\gamma} \right)^2$$

Osservazione 17.1.1. La condizione (i) non è restrittiva, infatti se \underline{w} è un vettore a cui l'iperpiano ottimo è perpendicolare possiamo dividerlo per la propria norma senza cambiare la sua direzione in modo da verificare (i), mentre la (ii) è una condizione di lineare separabilità del dataset, con γ che sarà il suo margine funzionale.

Dimostrazione. Per semplificare la matematica introduciamo una nuova notazione:

$$\bar{x}^{(i)} = \begin{bmatrix} x^{(i)} & R \end{bmatrix}^t \qquad \bar{w} = \begin{bmatrix} \underline{w} & \frac{b}{R} \end{bmatrix}^t$$

Ci si convince facilmente che in questi termini $\langle \bar{w}, \bar{x}^{(i)} \rangle = \langle \underline{w}, \underline{x}^{(i)} \rangle + b$ e quindi l'intera legge di aggiornamento dei parametri di PERCEPTRON può essere riscritta in questa forma:

$$\bar{w}_k \stackrel{def}{=} \begin{bmatrix} w_k \\ \frac{b_k}{R} \end{bmatrix} = \begin{bmatrix} w_{k-1} \\ \frac{b_{k-1}}{R} \end{bmatrix} + y^{(i)} \cdot \begin{bmatrix} x^{(i)} \\ R \end{bmatrix}$$

Adesso sviluppiamo il prodotto $\langle \bar{w}_k, \bar{w}^* \rangle$ per trovare una relazione che vale ad ogni passo dell'algoritmo:

$$\langle \bar{w}_k, \bar{w}^* \rangle = \langle \bar{w}_{k-1} + y^{(i)} \bar{x}^{(i)}, \bar{w}^* \rangle = \langle \bar{w}_{k-1}, \bar{w}^* \rangle + y^{(i)} \langle \bar{x}^{(i)}, \bar{w}^* \rangle \stackrel{(a)}{\geq} \langle \bar{w}_{k-1}, \bar{w}^* \rangle + \gamma$$

La prima uguaglianza si ottiene sostituendo la definizione di \bar{w}_k mentre la seconda distribuendo il prodotto scalare. Infine la disuguaglianza (a) vale in quanto il secondo addendo del membro di sinistra è proprio il margine funzionale del punto $\bar{x}^{(i)}$ che per ipotesi è più grande di γ . Procedendo per induzione su k ed usando questo come passo base, si può dimostrare che

$$\langle \bar{w}_k, \bar{w}^* \rangle \geq +k\gamma \quad \forall k \tag{17.1}$$

Adesso facciamo un'altra volta un procedimento simile, usando invece $\|\bar{w}_k\|$:

$$\|\bar{w}_k\|^2 = \|\bar{w}_{k-1} + y^{(i)} \bar{x}^{(i)}\|^2 = \|\bar{w}_{k-1}\|^2 + |y^{(i)}|^2 \cdot \|\bar{x}^{(i)}\|^2 + 2y^{(i)} \langle \bar{w}_{k-1}, \bar{x}^{(i)} \rangle$$

Notiamo che il termine $2y^{(i)} \langle \bar{w}_{k-1}, \bar{x}^{(i)} \rangle$ all'ultimo membro è l'iperpiano precedente applicato al punto $\bar{x}^{(i)}$ moltiplicato per la sua vera classe, ma il fatto che stiamo passando da $k-1$ a k significa che si stanno aggiornando i valori dei parametri, i.e. PERCEPTRON prima sbagliava a classificare, quindi tutto questo deve essere negativo. Di conseguenza

$$\|\bar{w}_k\|^2 \leq \|\bar{w}_{k-1}\|^2 + \|\bar{x}^{(i)}\|^2 \stackrel{(b)}{=} \|\bar{w}_{k-1}\|^2 + \|\underline{x}^{(i)}\|^2 + R^2$$

L'uguaglianza (b) si ottiene ricordando come si è definito $\bar{x}^{(i)}$ ad inizio dimostrazione in termini di $\underline{x}^{(i)}$ ed R . Ancora una volta questo risultato è generalizzabile, e per induzione su k usando questo come passo base si può far vedere che vale la seguente disuguaglianza ad ogni passo dell'algoritmo:

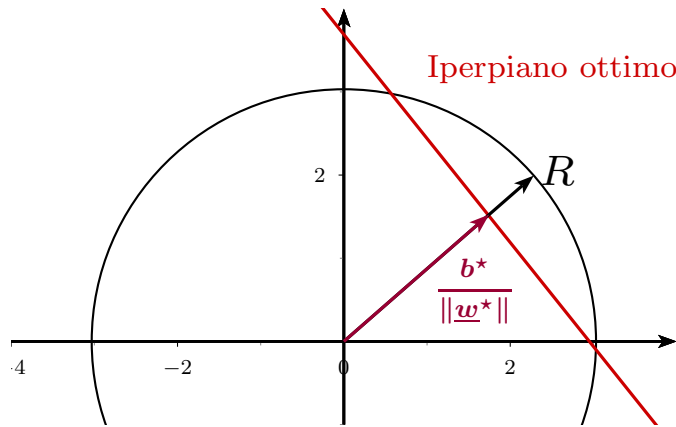
$$\|\bar{w}_k\|^2 \leq 2kR^2 \quad (17.2)$$

Usiamo le due relazioni appena dimostrate per trovare una limitazione superiore per k :

$$k\gamma \stackrel{(i)}{\leq} \langle \bar{w}_k, \bar{w}^* \rangle \stackrel{(ii)}{\leq} \|\bar{w}_k\| \cdot \|\bar{w}^*\| \stackrel{(iii)}{\leq} \sqrt{2kR^2} \|\bar{w}^*\|$$

La disuguaglianza (i) vale per la relazione 17.1, la (ii) applicando la disuguaglianza di Cauchy-Schwarz ed infine la (iii) come conseguenza della relazione 17.2. Procediamo considerando il primo e l'ultimo membro della precedente catena di disuguaglianze ed elevando ambo i membri al quadrato:

$$k^2\gamma^2 \leq 2kR^2\|\bar{w}^*\|^2 \implies k\gamma^2 \leq 2R^2\|\bar{w}^*\|^2 \implies k \leq 2 \left(\frac{R}{\gamma} \right)^2 \|\bar{w}^*\|^2 \quad (17.3)$$



Utilizzando la figura precedente si osserva che l'iperpiano di separazione ottimo taglia la circonferenza di raggio R , infatti tutti i punti del dataset sono contenuti qui dentro e se l'iperpiano fosse esterno tutti i punti verrebbero classificati allo stesso modo, contro l'ipotesi che il dataset fosse non banale. Questo fatto, insieme all'ipotesi che $\|\underline{w}^*\| = 1$ fa sì che $b < R$ e pertanto $\frac{b}{R} < 1$. Questo fatto ci avvia a concludere: ricordandoci come si è definito \bar{w}_k si può riscrivere la sua norma così: $\|\bar{w}^*\|^2 = \|\underline{w}^*\|^2 + \left(\frac{b^*}{R}\right)^2 = 1^2 + 1^2 = 2$. Sostituendo questo risultato nella relazione 17.3 si conclude in definitiva che

$$k \leq 4 \left(\frac{R}{\gamma} \right)^2 \implies k \leq \left(\frac{2R}{\gamma} \right)^2$$

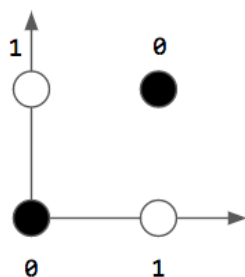
■

Se il dataset non è linearmente separabile la condizione $n_{err} \leq 0$ non è mai verificata e quindi PERCEPTRON cessa di funzionare, mancando la convergenza. Casi tipici di dataset non linearmente separabili sono quelli in cui il numero di esempi è alto ma la dimensione p è bassa, viceversa quando il dataset è in dimensione alta e ci sono pochi

esempi si ha lineare separabilità con probabilità 1. Se $p \gg n$ nonostante la quasi certa lineare separabilità si rischia una situazione indesiderata conosciuta come **overfitting**, cioè l'errore sul training set è piccolo ma sul test set è grande (fare bene sul training set non porta a generalizzare bene come invece si sperava).

17.1.2 Limiti e miglioramenti a Perceptron

PERCEPTRON, nonostante il suo iniziale successo e clamore suscitato, ha alcuni limiti molto evidenti dovuti al fatto che la sua classe di ipotesi sia composta da iperpiani. Esso riesce a fallire anche in casi banali come il cosiddetto **XOR problem**:



Data la situazione rappresentata in figura PERCEPTRON non riesce a generare una superficie di separazione in grado di classificare correttamente tutti gli esempi, a causa della non lineare separabilità del dataset. Di seguito esploriamo due possibili soluzioni a problemi di questo tipo.

Perceptron votato

Algoritmo 26 Classificazione binaria con dataset linearmente separabile

```

1: function PERCEPTRON-VOTATO( $\mathcal{D}$ ,  $T$ )
2:    $\underline{w}_0 \leftarrow 0$ 
3:    $b_0 \leftarrow 0$ 
4:    $k \leftarrow 0$ 
5:    $c_0 \leftarrow 0$ 
6:    $R \leftarrow \text{MAX}_i \|x^{(i)}\|$ 
7:   for epoca = 1, 2, ...,  $T$  do
8:     n_err  $\leftarrow 0$ 
9:     for  $i = 1, 2, \dots, n$  do
10:      if  $y^{(i)} (b_k + \langle \underline{w}_k, x^{(i)} \rangle) \leq 0$  then
11:         $\underline{w}_{k+1} \leftarrow \underline{w}_k + y^{(i)} x^{(i)}$ 
12:         $b_{k+1} \leftarrow b_k + y^{(i)} R^2$ 
13:         $k \leftarrow k + 1$ 
14:         $c_{k+1} = 1$ 
15:        n_err  $\leftarrow$  n_err + 1
16:      else
17:         $c_k \leftarrow c_k + 1$ 
18:   return  $\{(\underline{w}_j, b_j)\}_{j=1}^K, \{c_j\}_{j=1}^K$ 

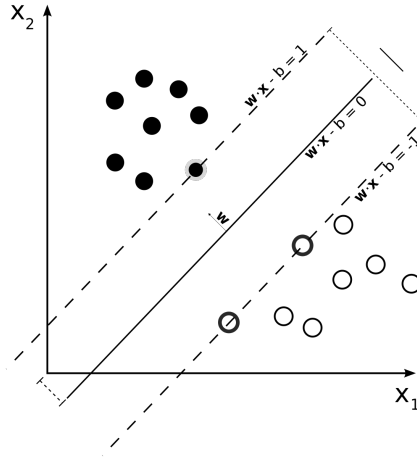
```

Si tratta di una revisione dell'algoritmo originale proposta da *Y. Freund* e *R. E. Schapire* nel 1999. In questa versione l'algoritmo si aspetta oltre al dataset un ulteriore ingresso T che rappresenta il numero massimo di iterazioni da poter eseguire prima di interrompersi. Oltre a ciò, nel ciclo delle epoche si mantiene un contatore c_k che rappresenta il numero di volte che l'iperpiano k è "sopravvissuto": tutte le volte che un iperpiano classifica male lo si reimposta ad 1, mentre se la classificazione è corretta c_k viene incrementato. L'output dell'algoritmo sarà l'intera sequenza di iperpiani generata insieme ai corrispondenti valori

di conteggio c_k , e la funzione di predizione che si andrà ad usare è la seguente:

$$f(x) = \text{SGN} \left[\sum_{j=1}^K c_j \cdot \text{SGN} (\langle \underline{w}_j, \underline{x} \rangle + b_j) \right]$$

Qui sopra K è il numero di iperpiani generati. L'algoritmo è piuttosto solido e va vicino a trovare il cosiddetto **iperpiano a massimo margine**: se il dataset è linearmente separabile, si chiama così quell'iperpiano che taglia a metà la separazione tra gli esempi, come si vede in figura:



Mentre se non sono linearmente separabili, ad esempio se ci fosse un punto nero tra i bianchi, l'algoritmo tende ad ignorarlo, intuitivamente perché presumibilmente l'iperpiano che lo classifica correttamente sopravvive molto poco ed il suo peso nella votazione (sommatoria) è basso.

Perceptron duale

Algoritmo 27 Classificazione binaria con dataset linearmente separabile

```

1: function PERCEPTRON-DUALE( $\mathcal{D}$ )
2:    $(\alpha^{(1)}, \dots, \alpha^{(n)}) \leftarrow (0, 0, \dots, 0)$ 
3:    $b_0 \leftarrow 0$ 
4:   do
5:      $n\_err \leftarrow 0$ 
6:     for  $i = 1, 2, \dots, n$  do
7:       if  $y^{(i)} \left( \sum_{j=1}^n \alpha^{(j)} y^{(j)} \langle \underline{x}^{(j)}, \underline{x}^{(i)} \rangle + b \right) \leq 0$  then
8:          $\alpha^{(i)} \leftarrow \alpha^{(i)} + 1$ 
9:          $b \leftarrow b + y^{(i)} R^2$ 
10:       $n\_err \leftarrow n\_err + 1$ 
11:   while  $n\_err \neq 0$ 
12:   return  $\alpha^{(1)}, \dots, \alpha^{(n)}, b$ 

```

Questa riformulazione si basa su una semplice osservazione che possiamo effettuare sul PERCEPTRON nella sua versione base, quella dell'algoritmo 25: si può infatti osservare, dato che ad ogni passo w_k resta invariato oppure gli si somma o sottrae l'esempio $x^{(i)}$, fine dell'esecuzione detto $\alpha^{(i)}$ il numero di volte che è stato sbagliato l'esempio i -esimo

dall'algoritmo, si può riscrivere w_k come segue:

$$\underline{w}_k = \sum_{i=1}^n \alpha^{(i)} y^{(i)} \cdot \underline{x}^{(i)}$$

Perciò la funzione di predizione di PERCEPTRON base può anche essere riscritta così:

$$f(\underline{x}) = \text{SGN} \left[\sum_{i=1}^n \alpha^{(i)} y^{(i)} \langle \underline{x}^{(i)}, \underline{x} \rangle + b \right] \quad (17.4)$$

In questo senso si vede che PERCEPTRON-DUALE fa esattamente la stessa cosa di PERCEPTRON ma semplicemente usando una diversa rappresentazione. Giocando su questa cosa possiamo dare anche una interpretazione di come opera questo algoritmo. Se la funzione di predizione fosse della forma $f(\underline{x}) = \text{SGN}(\sum_i y^{(i)})$ allora il classificatore predirebbe sempre la classe più frequente in quanto la somma è positiva solo se gli esempi positivi sono più di quelli negativi, mentre è negativa altrimenti. Se invece si guarda anche il prodotto scalare tra l'esempio $\underline{x}^{(i)}$ ed il punto di test \underline{x} , portando la funzione di predizione alla forma $f(\underline{x}) = \text{SGN}(\sum_i y^{(i)} \langle \underline{x}^{(i)}, \underline{x} \rangle)$, si classifica \underline{x} in base alla similitudine con gli esempi del training set, infatti tanto più i due sono simili (\simeq paralleli) quanto più il prodotto scalare è grande, e viceversa più sono diversi tanto più il prodotto è piccolo. Aggiungendo $\alpha^{(i)}$ come fa PERCEPTRON-DUALE si ottiene un classificatore che non solo guarda quanto il punto è simile a quelli del training set ma anche a quanto è stato difficile classificarlo; più errori sono stati commessi più è facile che il punto sia anomalo rispetto al resto della distribuzione e quindi gli si assegna un peso maggiore, perché altrimenti sarebbe difficile predire con buona accuratezza.

Notiamo infine che se sostituiamo in 17.4 il prodotto scalare con un'altra funzione in uno spazio diverso, ma che sia ancora interpretabile come un prodotto scalare, si può fare in modo che PERCEPTRON-DUALE produca delle superfici di separazione non lineari. Ad esempio sostituendolo con qualcosa come $(\langle \underline{x}^{(i)}, \underline{x} \rangle + 1)^2$ si può classificare anche il problema dello XOR presentato in apertura di sezione, e più in generale si potrebbe pensare ad una estensione per la quale invece di usare il prodotto scalare di sopra se ne usa uno della forma $(\langle \underline{x}^{(i)}, \underline{x} \rangle + 1)^p$, e si parla in questo caso di **polynomial preprocessing**.

30/11/2020

18.1 Precisazioni su Perceptron

Muoviamo di seguito due precisazioni per quanto riguarda PERCEPTRON. La prima di queste è riferita a cosa succede quando una delle variabili del dataset è di tipo categorico (cfr definizione 14.2). Fino a questo momento abbiamo assunto che i vari esempi fossero dei vettori, i.e. $x^{(i)} \in \mathbb{R}^p$ per qualche $p \in \mathbb{N}$, e quindi tutte le sue componenti sono dei numeri reali. Come fare per dare in pasto all'algoritmo dei dati provenienti da distribuzioni categoriche?

Esempio 18.1: Perceptron con variabili categoriche

Supponiamo che una delle variabili aleatorie a cui si riferisce il dataset sia **Workclass**, che assumiamo essere una variabile aleatoria categorica a 6 livelli, che sono i seguenti: {P,S,F,L,W,N}. Una prima brutale idea potrebbe essere di associare a ciascun livello un valore numerico, ad esempio P:1, S:2, F:3 e così via, per poi usare questi dentro ad $x^{(i)}$ al posto dei corrispondenti valori letterali. Non si tratta tuttavia di un'idea intelligente da applicare nella pratica, perché PERCEPTRON classifica in base alla similitudine con gli esempi e cambiando l'ordine dei livelli, e.g. assegnando L:1, P:2, N:3 e via dicendo, cambia il valore del prodotto scalare tra livelli distinti e quindi anche il grado di similitudine percepito dall'algoritmo. La scelta standard in questo caso è invece quella di istituire una codifica 1-hot dei valori, per esempio assegnare $P=(1, 0, 0, 0, 0, 0)$, $S=(0, 1, 0, 0, 0, 0)$, $W=(0, 0, 1, 0, 0, 0) \dots$. In questo il prodotto scalare tra due livelli distinti si annulla e contribuiscono soltanto i valori uguali. ■

Per quanto riguarda invece la seconda precisazione, abbiamo accennato all'idea di poter sostituire il prodotto scalare nella relazione 17.4 con uno generalizzato, detto **kernel**, in modo da ottenere superfici di separazione non lineari. Se $k(x^{(i)}, x)$ è un kernel, la funzione di predizione cambia come segue:

$$f(\underline{x}) = \text{SGN} \left(\sum_{i=1}^n \alpha^{(i)} y^{(i)} k(x^{(i)}, x) + b \right)$$

Definizione 18.1 (Kernel valido). Una funzione $k: \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ si dice essere un kernel **valido** se, detto \mathcal{F} uno spazio di Hilbert, esiste una mappa $\phi: \mathcal{X} \mapsto \mathcal{F}$ tale per cui

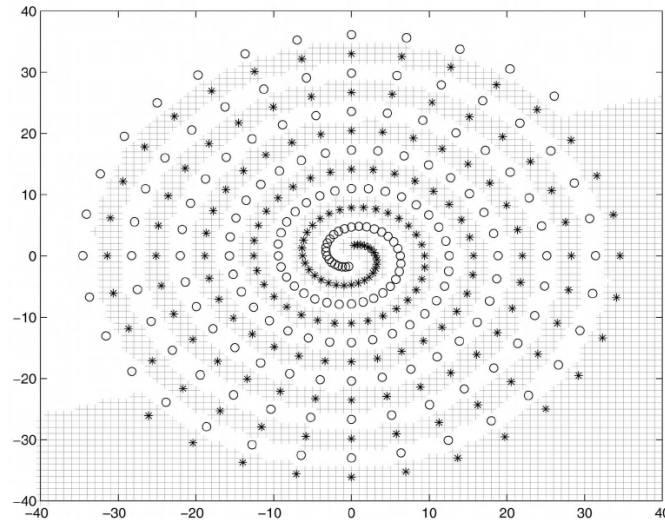
$$\forall x, z \in \mathcal{X} \quad \langle \phi(x), \phi(z) \rangle = k(x, z)$$

Quindi un kernel valido è un kernel che può essere usato come surrogato di un prodotto scalare. Ad esempio se poniamo $\mathcal{X} = \mathbb{R}^2$ la funzione $k(\underline{x}, \underline{z}) = (\langle \underline{x}, \underline{z} \rangle + 1)^2$ è uno di questi:

$$\begin{aligned} k(\underline{x}, \underline{z}) &= (\langle (x_1, x_2), (z_1, z_2) \rangle + 1)^2 = (x_1 z_1 + x_2 z_2 + 1)^2 \\ &= (x_1^2 z_1^2 + x_2^2 z_2^2 + 1 + 2x_1 z_1 + 2x_2 z_2 + 2x_1 z_1 x_2 z_2) = \langle \phi(x), \phi(z) \rangle \end{aligned}$$

Dove si pone $\phi(x) = [x_1^2, x_2^2, \sqrt{2}x_1 x_2, \sqrt{2}x_1, \sqrt{2}x_2, 1]^t$, ed è intuibile che sostituendo il 2 all'esponente con un generico $d \in \mathbb{N}$ si ottiene nuovamente un kernel valido; si definisce

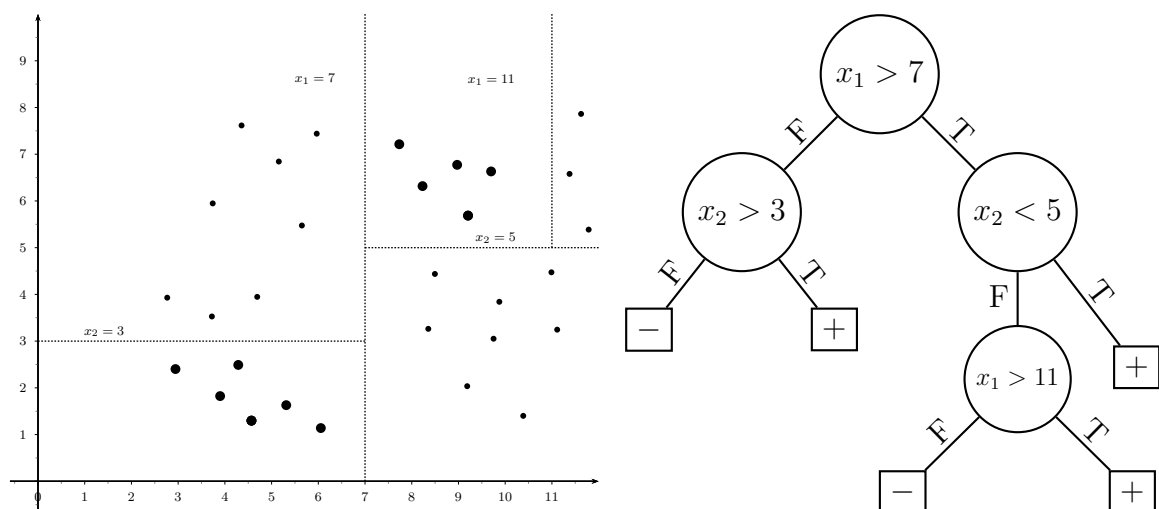
così una famiglia nota come **kernel polinomiali**. Esiste un'ulteriore classe nota come **Kernel RBF**, che sono della forma $k(x, z) = e^{-\gamma \|x-z\|^2}$, e sono estremamente espressivi, tanto da riuscire a classificare anche datasets fortemente non lineari come quello in figura.



Qui sopra \circ rappresenta gli esempi di classe positiva e $*$ quelli di classe negativa; inoltre le regioni più scure sono quelle i cui punti vengono classificati come negativi, in bianco invece le regioni i cui punti vengono classificati come positivi. Il grafico di $k(x, z)$ in questo caso ricorda la densità di una variabile gaussiana con l'iperparametro γ che controlla la velocità con cui la curva va a 0. Intuitivamente è come se l'algoritmo creasse intorno a ciascun punto una sfera più o meno grande in base al valore di γ e classificasse tutti i punti di quella sfera copiando la classe del punto "centrale".

18.2 Alberi di decisione

Un albero di decisione è un albero i cui nodi interni rappresentano dei test sugli attributi (tipicamente un singolo attributo). Ad ogni diverso valore che può essere assunto dall'attributo testato corrisponderà un ramo nel caso categorico, ed a ciascuna foglia corrisponderà una classe. Una decisione corrisponde ad un cammino completo sull'albero.



La figura mostra come si può utilizzare un albero di decisione per la classificazione binaria, e che la tecnica funziona benissimo senza nessun requisito di lineare separabilità del

dataset, basandosi essa su una partizione ricorsiva dello spazio delle istanze \mathcal{X} . Inoltre si intuisce che la classificazione tramite albero di decisione è estremamente interpretabile. Quando si lavora con attributi continui invece si stabilisce un valore di soglia e si ramifica rispetto a questo, e.g. dato un attributo **Altezza** possiamo scegliere come soglia 170cm e ramificare in base al fatto che **Altezza** ≥ 170 oppure no. Infine, per cercare di contenere il fenomeno dell'overfitting si cerca di prendere tra tutti i possibili alberi di decisione quello che ha il minor numero di nodi interni, ma anche questo problema è NP-Hard.

18.2.1 Algoritmo basato su albero di decisione

Algoritmo 28 Classificazione attraverso albero di decisione

```

1: function LEARN-DECISION-TREE(esempi, attributi, parent_examples)
2:   if esempi è vuoto then
3:     return CLASSE-MAGGIORANZA(parent_examples)
4:   else
5:     if tutti gli esempi hanno la stessa classe then
6:       return La classe comune degli esempi
7:     else ▷ Cuore dell'algoritmo
8:        $A \leftarrow \underset{a \in \text{attributi}}{\text{MAX}} \{ \text{GUADAGNO}(a, \text{esempi}) \}$ 
9:       albero  $\leftarrow$  Un albero di decisione con root-test su A
10:      for all valori  $v \in A$  do
11:        esempi_v  $\leftarrow \{ e \in \text{esempi} \mid e.A = v \}$ 
12:        sottoalbero  $\leftarrow$  LEARN-DECISION-TREE(esempi_v, attributi, esempi)
13:        Aggiungi un ramo ad albero con etichetta  $A = v$ 
14:        Aggiungi il sottoalbero sottoalbero al ramo " $A = v$ "
15:      return albero

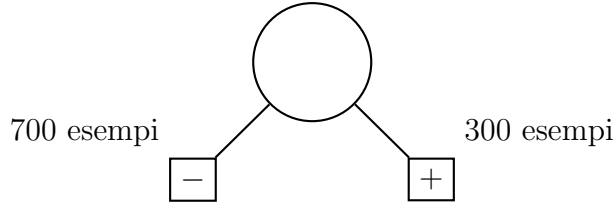
```

L'algoritmo funziona in questo modo: se sono finiti gli esempi da testare si restituisce la classe di maggioranza nel nodo genitore di quello correntemente testato, altrimenti si verifica immediatamente se tutti gli esempi hanno la stessa classe, nel qual caso si restituisce tale classe. Finiti di testare i casi banali si passa al cuore dell'algoritmo: tra tutti gli attributi possibili si sceglie quello che massimizza una funzione di importanza opportunamente scelta (vedremo a breve come) e si genera un sottoalbero che ha per radice un test su questo attributo. Taluno a sua volta conterrà un sottoalbero per ogni possibile valore v dell'attributo scelto, che verrà costruito ricorsivamente. Fatto ciò, tutto quanto viene agganciato all'albero di decisione.

Soffermiamoci adesso su come si prende la funzione GUADAGNO. Dato un dataset \mathcal{D} chiamiamo $\text{COST}(\mathcal{D})$ la sua impurità, che è minima (nulla) quando tutti gli esempi sono della stessa classe e massima quando tutti gli esempi sono di classi diverse. Se gli attributi sono X_1, \dots, X_p si vuole trovare l'attributo migliore X_{j^*} ed una scelta plausibile è questa:

$$j^* = \underset{1 \leq j \leq p}{\text{MAX}} \{ \text{GUADAGNO}(j, \mathcal{D}) \} = \underset{1 \leq j \leq p}{\text{MAX}} \{ \text{COSTO}(\mathcal{D}) - \text{Costi sottodatasets} \}$$

Immaginiamo per esempio di avere un dataset con $n = 1000$ esempi e con due soli classi, oltre ad avere uno split che classifica correttamente tutti gli esempi, come in questa situazione:



In questo caso i costi dei sottodatasets sono tutti nulli e siamo nel caso ideale. Nella pratica l'idea è che se i costi dei sottodatasets sono relativamente piccoli rispetto a quello del dataset di partenza \mathcal{D} allora lo split è vantaggioso. Nel definire i costi occorre tenere anche in considerazione la proporzione di esempi che sono finiti in ciascun sottodataset, per esempio il costo di $\boxed{+}$ verrà moltiplicato per 0.3 mentre quello di $\boxed{-}$ verrà pesato con un fattore 0.7. Con queste considerazioni la quantità da massimizzare sarà la seguente:

$$j^* = \text{MAX}_{1 \leq j \leq p} \left\{ \text{COSTO}(\mathcal{D}) - \sum_{v \in \mathcal{D}_j} \frac{|\mathcal{D}_v|}{|\mathcal{D}|} \cdot \text{COSTO}(\mathcal{D}_v) \right\}$$

Dove \mathcal{D}_j è il dominio dell'attributo X_j mentre \mathcal{D}_v è l'insieme degli esempi in cui X_j prende realizzazione v .

Misurare il costo di un dataset

Sia $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}, i = 1, 2, \dots, n\}$ ed assumiamo che le possibili classi siano $\{1, 2, \dots, K\}$. Per ogni classe k si definisce la seguente quantità:

$$\hat{p}_k = \frac{1}{n} \sum_{i=1}^n \mathbb{1}\{y^{(i)} = k\}$$

Questo valore rappresenta la frazione di esempi di classe k . Chiamiamo inoltre k^* la classe più frequente, ossia quella che realizza il massimo valore di \hat{p}_k . Di seguito riportiamo tre possibili misure di impurità che si possono usare per misurare il costo di un dataset:

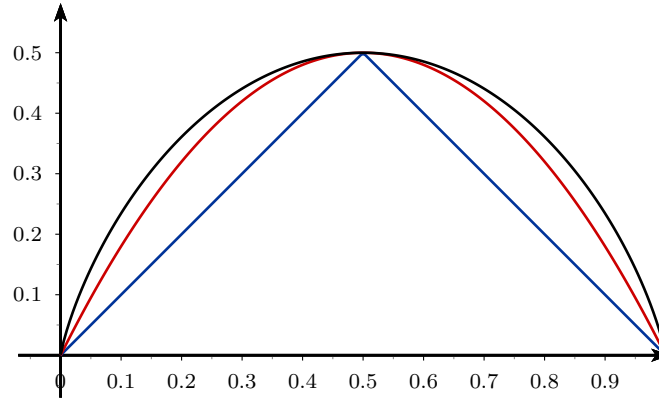
- (i) **Erroro di classificazione:** Si definisce l'errore di classificazione come $1 - \hat{p}_{k^*}$. Si attribuisce a tutti gli esempi la classe di maggioranza e pertanto il valore precedente è la frazione di esempi erroneamente classificati come di classe k^* .
- (ii) **Indice di Gini:** $\sum_{k=1}^K \hat{p}_k(1 - \hat{p}_k)$
- (iii) **Entropia:** è il numero atteso di bits necessari per codificare le diverse classi in una sorgente di informazione, assumendo che più una classe è frequente più è corta la sua rappresentazione. Si dimostra che tale valore atteso è il seguente:

$$S = - \sum_{k=1}^K \hat{p}_k \log_2(\hat{p}_k)$$

Assumiamo di avere due sole classi, una positiva ed una negativa, e che $p^+ = \mathbb{P}(\text{classe } +)$, $p^- = \mathbb{P}(\text{classe } -)$ siano le rispettive proporzioni di esempi:

- (i) L'errore di classificazione è $1 - \text{MAX}\{p^+, 1 - p^+\}$. Quando $p^+ < 0.5$ il massimo è realizzato da $1 - p^+$ e quindi l'errore di classificazione è p^+ , mentre al contrario quando $p^+ > 0.5$ il massimo è dato da p^+ e quindi l'errore di classificazione è $1 - p^+$. Nella figura sottostante è il grafico blu.

- (ii) L'indice di Gini è dato da $p^+(1 - p^+) + p^-(1 - p^-) = 2p^+(1 - p^+)$ che è una parabola con concavità rivolta verso il basso, massimo in $p^+ = 0.5$ e che interseca l'asse orizzontale in $p^+ = 0$ e $p^+ = 1$. Nella figura sottostante è il grafico rosso.
- (iii) L'entropia si calcola come $-p^+ \log_2(p^+) - (1 - p^+) \log_2(1 - p^+)$, che vale 0 in $p^+ = 0$ e $p^+ = 1$, mentre nel caso generale diversamente dai due casi precedenti non ha come valore massimo 0.5, sebbene si possa cambiare la base del logaritmo per far sì che questo avvenga. Nella figura sottostante è il grafico nero.



Sull'asse orizzontale del grafico soprastante è riportato p^+ mentre su quello verticale si riporta il costo corrispondente. Mostriamo con un esempio che tuttavia l'utilizzo dell'errore di classificazione non è particolarmente indicato.

Esempio 18.2: Inadeguatezza dell'errore di classificazione

Supponiamo di avere un dataset \mathcal{D} contenenti esempi su due classi diverse, indicate al solito come $+$ e $-$. Consideriamo i due seguenti split:



Lo split di destra è più promettente rispetto a quello di sinistra, infatti permette di introdurre una foglia e quindi potenzialmente porta a un albero di decisione più piccolo. Calcoliamo il termine da sottrarre nel calcolo del guadagno per i due split usando l'errore di classificazione:

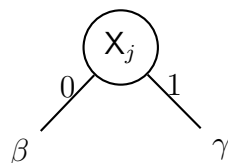
$$(i) \text{ GUADAGNO}_{\text{SX}} = \frac{400}{800} \cdot \frac{100}{100 + 300} + \frac{400}{800} \cdot \frac{100}{100 + 300} = 0.25$$

$$(ii) \text{ GUADAGNO}_{\text{DX}} = \frac{600}{800} \cdot \frac{200}{200 + 400} + \frac{200}{800} \cdot \frac{0}{100 + 300} = 0.25$$

Quindi usando questa misura di impurità non si riesce a cogliere il vantaggio dello split di destra. Invece usando l'indice di Gini si ottiene rispettivamente un costo di 0.333 per quello di destra e 0.375 per quello di sinistra, spingendoci pertanto verso il primo che in effetti è più promettente. ■

Sempre sulla falsariga dell'esempio proposto facciamo vedere di seguito il vantaggio di avere una funzione di costo concava come sono l'indice di Gini o la stessa entropia rispetto

all'errore di classificazione. Consideriamo la seguente situazione: abbiamo nell'albero di decisione un test sull'attributo binario X_j e nel sottoalbero sinistro finisce una proporzione di esempi positivi pari ad un certo $\beta = \mathbb{P}(Y = 1 | X_j = 0)$, mentre in quello destro finisce una frazione di esempi positivi pari a $\gamma = \mathbb{P}(Y = 1 | X_j = 1)$.

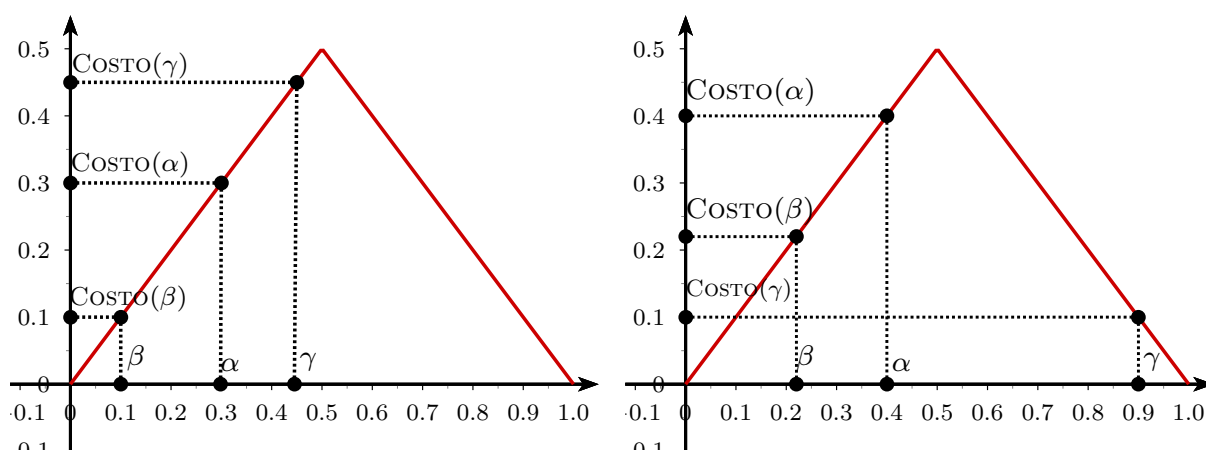


Chiamiamo inoltre $\mu = \mathbb{P}(X_j = 0) = \frac{|\mathcal{D}_0|}{|\mathcal{D}|}$ che rappresenta la frazione di esempi che hanno valore dell'attributo j pari a 0, ed $1 - \mu = \mathbb{P}(X_j = 1)$ la frazione di esempi con $X_j = 1$, allora dalla formula di probabilità totale, detta α la frazione di esempi positivi:

$$\alpha \stackrel{def}{=} \mathbb{P}(Y = 1) = \mathbb{P}(Y = 1 | X_j = 0) \mathbb{P}(X_j = 0) + \mathbb{P}(Y = 1 | X_j = 1) \mathbb{P}(X_j = 1) = \mu\beta + (1 - \mu)\gamma$$

Il guadagno associato alla scelta di X_j è espresso dalla seguente equazione:

$$\text{GUADAGNO}(X_j) = \text{COSTO}(\alpha) - \mu \text{COSTO}(\beta) - (1 - \mu) \text{COSTO}(\gamma)$$



Nella figura di sopra sono riportati due casi diversi, quello di sinistra è sfortunato mentre quello di destra è più fortunato. Focalizziamoci innanzitutto su quest'ultimo: osservando il grafico si riesce a notare che in effetti

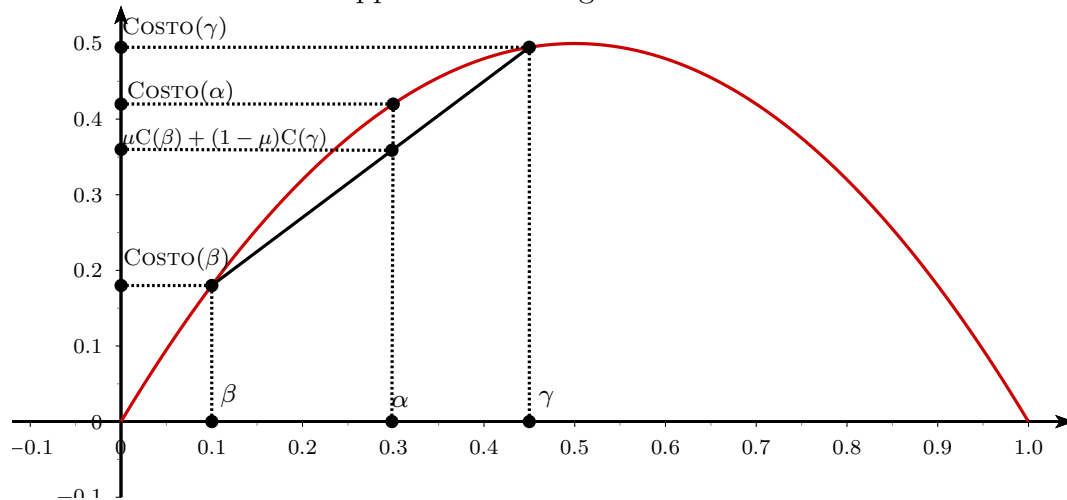
$$\text{COSTO}(\alpha) > \mu \text{COSTO}(\beta) + (1 - \mu) \text{COSTO}(\gamma)$$

Pertanto ci si accorge che lo split è vantaggioso in quanto prima, con una proporzione di esempi α , si pagava di più di quanto si pagherebbe in media pesata con i due sottodatasets aventi frazioni di esempi positivi pari a β e γ rispettivamente. Per quanto riguarda invece il caso sfortunato, figura di destra, viene fuori che il costo di α sta proprio lungo la stessa retta e quindi vale questa uguaglianza:

$$\text{COSTO}(\alpha) = \mu \text{COSTO}(\beta) + (1 - \mu) \text{COSTO}(\gamma)$$

In questo caso non ci si accorge che lo split è vantaggioso in quanto il costo rimane invariato e pertanto l'algoritmo non seleziona X_j , ed è quello che succede nell'esempio

18.2. Immaginiamo di essere sempre nel caso sfortunato ma di utilizzare invece l'indice di Gini piuttosto che l'errore di classificazione come misurazione del costo. Allora in questo caso si verifica la situazione rappresentata in figura:

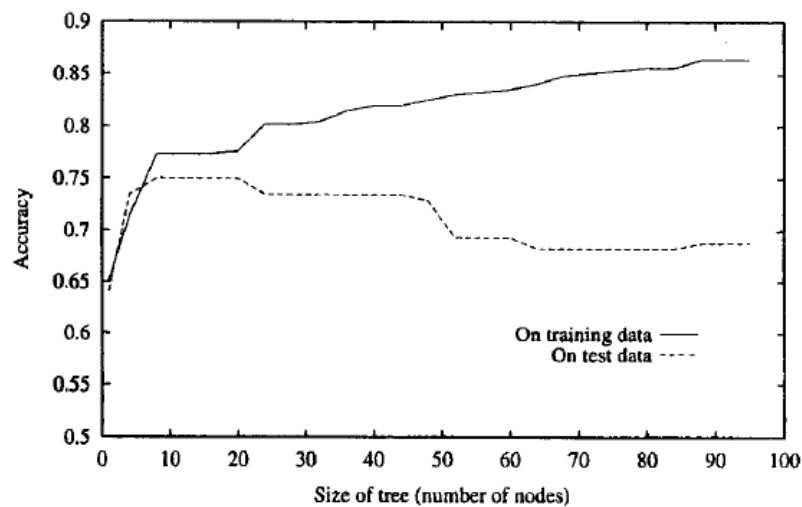


In questo caso la media dei costi dei sottodatasets con proporzioni β e γ è minore del costo del dataset di partenza e quindi, proprio grazie al fatto che la funzione è strettamente convessa si torna ad accorgersi della vantaggiosità. Il vantaggio è ancora più grande se si usa l'entropia in quanto il margine tra i costi aumenta, ma sia questa che Gini sono adeguati; conviene semplicemente evitare l'errore di classificazione perché può far perdere splits vantaggiosi e di conseguenza portare ad alberi di decisione innecessariamente grandi.

3/12/2020

19.1 Potatura di alberi di decisione

L'algoritmo LEARN-DECISION-TREE visto nello scorso capitolo fa crescere ciascun ramo dell'albero quanto basta per classificare perfettamente tutti gli esempi del training set. Sebbene la strategia sia ragionevole, può portare problematiche quando c'è del rumore nei dati o quando il numero di esempi nel training set è troppo piccolo per essere un campione rappresentativo della funzione obiettivo; in entrambi questi casi l'algoritmo può produrre alberi che soffrono del problema dell'overfitting.



La figura precedente ritrae il fenomeno descritto: la linea continua rappresenta la precisione delle predizioni effettuate dal classificatore sul training set al variare della dimensione dell'albero, mentre quella tratteggiata la sua precisione misurata su un insieme di esempi indipendente dal precedente detto **insieme di convalida** (*validation set*). Come ci si aspetterebbe la precisione cresce monotonicamente sul training set con l'aumentare del numero di nodi dell'albero, mentre sul validation set l'andamento è più irregolare e tendente alla decrescita.

L'overfitting è una problematica pratica significativa nell'apprendimento tramite alberi di decisione: uno studio sperimentale su dati affetti da rumore non deterministici ha mostrato che questo può portare ad un calo nella precisione di un 10 – 25% sulla maggior parte dei problemi. È importante perciò correre ai ripari e munirsi di strategie utili a contrastare il fenomeno, e qui se ne distinguono di due tipi:

- (i) **Pre-potatura.** Si tratta di un approccio che cerca di impedire all'albero di crescere eccessivamente già durante la fase di costruzione, arrestando il processo prima che l'albero arrivi a classificare perfettamente tutti gli esempi del training set.
- (ii) **Post-potatura.** Con questa strategia si permette all'albero di crescere arbitrariamente per poi rimuovere dei sottoalberi in seguito.

A causa della difficoltà nello stabilire adeguatamente quanto accrescere l'albero durante la fase di costruzione, la seconda strategia si è rivelata spesso più efficace nella pratica

rispetto alla prima. In entrambi casi dobbiamo stabilire come decidere se un albero di decisione è di una dimensione appropriata o meno. Uno dei criteri possibili in questo senso è di usare un insieme di esempi distinto da quello usato per addestrare il classificatore per valutare l'utilità di potare un nodo, mentre tecniche più raffinate possono impiegare dei test d'ipotesi per stimare se l'espansione di un nodo possa portare a miglioramenti nelle prestazioni su tutta la distribuzione degli esempi e non solo sul training set.

Il primo tipo di approccio è quello più comune: si decide di separare gli esempi a disposizione in due insiemi, uno da utilizzare per addestrare il modello, chiamato training set, ed un secondo usato per mettere alla prova il classificatore e verificare che non sia troppo aderente alle caratteristiche del training set. L'idea qui è che in quest'ultimo potrebbero esserci delle fluttuazioni casuali o delle regolarità accidentali che rischiano di portare fuori strada il classificatore, ma è improbabile che le stesse fluttuazioni casuali si ripropongano anche nel set di validazione. Come indicazione qualitativa, si dovrebbe cercare di mantenere un terzo degli esempi nell'insieme di convalida e due terzi in quello di addestramento. Vediamo adesso una tecnica di potatura basata su questa idea.

19.1.1 Potatura di regole

In questa sezione vediamo una tecnica di post-potatura basata sull'approccio training-validation set nota come **rule pruning**. Questa si articola nei passaggi seguenti:

- (i) Si ricava un albero di decisione dal training set, permettendogli di crescere finché l'albero non classifica nel miglior modo possibile gli esempi e permettendo quindi il verificarsi dell'overfitting.
- (ii) Si converte ogni possibile cammino dalla radice ad una delle foglie in un insieme di regole ad esso equivalente; per esempio con riferimento alla figura in apertura di sezione 19.2, il cammino che si ottiene scendendo sempre a sinistra è equivalente alla regola seguente:

if $\neg(x_1 > 7) \wedge \neg(x_2 > 3)$ **then**
 Classe = \square
- (iii) Si pota (generalizza) ciascuna regola rimuovendo delle sottocondizioni, se questo fa aumentare l'accuratezza stimata. Per esempio nella regola del punto precedente si considera di eliminare, una per volta, $\neg(x_1 > 7)$ e $\neg(x_2 > 3)$. Se eliminare una di queste porta ad un miglioramento nell'accuratezza attesa allora la si pota, e se più di una quando eliminata produce un miglioramento si pota quella che porta il più significativo. Se nessuna potatura porta ad un miglioramento dell'accuratezza stimata questo passaggio viene saltato.
- (iv) Si ordinano le regole in base alla loro accuratezza stimata, e quando si deve classificare una nuova istanza si considerano nell'ordine così stabilito.

Per quanto riguarda il punto (ii) il miglioramento nell'accuratezza della classificazione può essere stimato utilizzando l'insieme di convalida, ma alcune pubblicazioni, come [CQ93], utilizzano lo stesso training set, impiegando una stima pessimistica per compensare il fatto che questo produrrà una stima sbilanciata a favore delle regole. In particolare tale stima viene effettuata utilizzando una distribuzione binomiale con parametro n pari al numero di esempi a cui la regola si applica e parametro p non noto. Calcolando l'accuratezza media sul training set e la sua deviazione standard si può costruire un intervallo di

confidenza con un livello di fiducia assegnato, un valore tipico è $1 - \alpha = 0.95$, e si prende come stima pessimistica l'estremo inferiore dell'intervallo, i.e.

$$\bar{A} = \bar{A}_T + z_{\frac{\alpha}{2}} S$$

Con \bar{A} l'accuratezza stimata della regola, \bar{A}_T la stima di tale accuratezza eseguita sul training set, ed S la deviazione standard stimata.

19.2 Convalida incrociata K-fold

Assegnato un classificatore introduciamo delle metriche utili a valutare le sue prestazioni. Dato un input x chiamiamo $f(x)$ la funzione che predice il valore corrispondente di y , ed introduciamo una funzione di penalizzazione (*loss*) per gli errori di predizione; taluna dovrebbe essere nulla quando si predice correttamente e positiva altrimenti, per cui una scelta tipica è la seguente:

$$L(y, f(x)) = (y - f(x))^2$$

Legato a questo concetto, fissato un training set τ possiamo definire un **errore di predizione condizionato** che rappresenta l'errore medio in cui un modello addestrato su τ incorrerebbe quando applicato ad esempi provenienti dalla stessa distribuzione \mathbb{P}_{XY} da cui proviene il dataset, e viene indicato come ERR_τ . In pratica:

$$\text{ERR}_\tau = \mathbb{E} [L(y, f(x))]$$

Dove il valore atteso è calcolato rispetto alla distribuzione congiunta \mathbb{P}_{XY} . Per ottenere una metrica che non tenga conto della randomicità nell'aver selezionato un training set τ fortunato o sfortunato infine si generalizza questo concetto e si introduce l'**errore di predizione atteso**, indicato come ERR e definito nel modo seguente:

$$\text{ERR} = \mathbb{E} [\text{ERR}_\tau]$$

Dove il valore atteso è eseguito su tutti i possibili training set.

L'idea alla base della **convalida incrociata** (*cross-validation*) è la seguente: se avessimo molti dati potremmo partizionarli in due parti ed usare una di queste per addestrare il modello, mentre l'altra per stimare le prestazioni; in pratica tuttavia i dati molti spesso scarseggiano e quindi questa strada potrebbe non essere possibile. Come soluzione elegante a questo problema la tecnica proposta prevede di suddividere i dati in K parti approssimativamente delle stesse dimensioni, usarne $K - 1$ per addestrare il modello e la rimanente per valutare la bontà del modello ottenuto. Se indichiamo con $j \in \{1, 2, \dots, K\}$ l'indice della parte rimasta esclusa, il procedimento prevede di ripetere il tutto per $j = 1, 2, \dots, K$ ed alla fine combinare le K stime ottenute per l'errore di predizione. Sia \hat{f}_j la funzione di classificazione calcolata rimuovendo la j -esima parte e sia $j(i): \{1, 2, \dots, K\} \mapsto \{1, 2, \dots, K\}$ una funzione d'indicizzazione che mappa l'indice i nel numero d'ordine della tasca in cui è finito l'esempio i -esimo, e sia n la dimensione del dataset; allora la stima dell'errore di predizione ottenuto in questo modo è data dalla seguente formula:

$$\text{CV}(\hat{f}) = \frac{1}{n} \sum_{i=1}^n L(y^{(i)}, \hat{f}_{j(i)}(x^{(i)}))$$

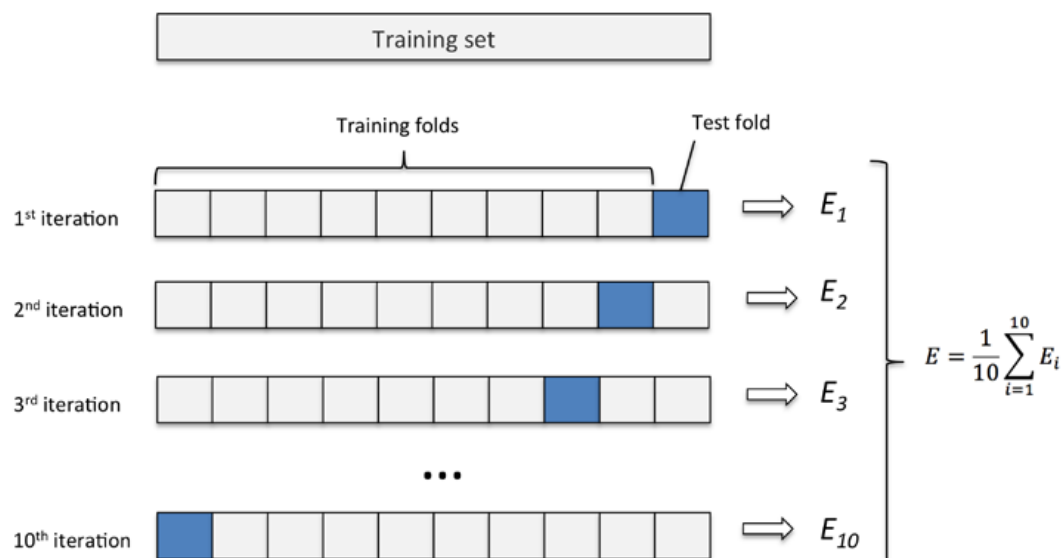


Figura 19.1: Rappresentazione di convalida incrociata 10-fold.

Quando $K = n$ ogni volta viene escluso un solo esempio e si parla in quel caso di *leave-one-out*. Potremmo chiederci quale errore stimiamo in effetti usando la convalida incrociata tra quelli descritti sopra. Per K abbastanza grande in effetti stiamo valutando molti training sets diversi e quindi potremmo congetturare di stare stimando l'errore di predizione atteso ERR , ma d'altra parte se si usa *leave-one-out* i training set sono molto simili tra di loro e quindi potremmo supporre di stare stimando l'errore di predizione condizionato ERR_{τ} . In pratica si può far vedere che solo il primo dei due può essere stimato efficacemente in questo modo.

Infine un'altra questione che tiene banco è quale valore scegliere per K . Se si sceglie $K = N$ allora si ottiene uno stimatore approssimativamente non distorto per il vero errore di predizione, tuttavia poiché gli N datasets utilizzati sono molto simili si potrebbe soffrire di alta varianza. Per K intermedi, come $K = 5$ o $K = 10$, la convalida incrociata produce una varianza minore ma la distorsione della stima potrebbe diventare un problema, a seconda di come la prestazione del metodo di apprendimento varia in funzione della dimensione del training set.

Come compromesso tra varianza e distorsione spesso sono raccomandati valori di K dell'ordine di cinque o dieci.¹

19.3 Bagging e Random Forest

Con **bagging** facciamo riferimento ad un insieme di algoritmi di apprendimento che per ottenere una migliore predizione fanno uso di più modelli. Il termine deriva dall'unione delle due parole inglesi *bootstrap*, riferito alla tecnica di campionamento utilizzata, ed *aggregation*, che invece fa riferimento a come viene ottenuta la predizione a partire da quella dei singoli modelli.

In teoria il **bootstrap** è una tecnica di ricampionamento utilizzata per stimare la distribuzione campionaria di uno stimatore assegnato che fa uso della reimmissione; in pratica nel nostro caso a partire da un dataset \mathcal{D} di dimensione n si possono ottenere

¹In parole povere, la convalida incrociata K -fold suddivide il dataset in K gruppi di numerosità simile ed escludendone uno per volta cerca di predirlo usando i rimanenti.

dei nuovi datasets $\mathcal{D}_1, \dots, \mathcal{D}_k$ ciascuno dei quali con una dimensione fissata $m < n$ ed ottenuto estraendo con reimmissione (di modo da simulare una popolazione infinita) esempi da \mathcal{D} . Se si assume che i dati nel dataset originale siano veramente rappresentativi della distribuzione $\mathbb{P}_{\mathbf{X}\mathbf{Y}}$ allora campionando con reinserimento da \mathcal{D} si simula questa stessa distribuzione e ciascuno dei nuovi dataset potrà essere utilizzato per addestrare un classificatore; l'errore di classificazione per il dataset \mathcal{D}_j potrà essere valutato sui punti in $\mathcal{D} \setminus \mathcal{D}_j$. Ripetendo il processo per k volte, una per ciascun dataset, si ottengono k stime dell'errore di classificazione e si può prendere come *bootstrap estimate* la loro media, e si dimostra che la stima così ottenuta ha una varianza più bassa rispetto a quella che avremmo ottenuto suddividendo arbitrariamente il dataset \mathcal{D} in training set ed insieme di convalida.

Una volta campionati i dataset ciascuno di questi può essere utilizzato per addestrare uno stesso tipo di classificatore, e questo produrrà k *votazioni* per la classe a cui appartiene un esempio che si vuole classificare; i voti vengono ricombinati mediante il processo di **aggregazione** che in questo caso consiste nel selezionare come classe predetta quella più frequente tra i voti. Quando si prende come classificatore da addestrare con i datasets $\mathcal{D}_1, \dots, \mathcal{D}_k$ un albero di decisione l'algoritmo che si ottiene è detto **random forest**. Talvolta per costruire i datasets derivati vengono scelte a caso delle **caratteristiche** (*features*)² e questa strategia permette di minimizzare l'overfitting del training set quando si usano alberi di decisione.

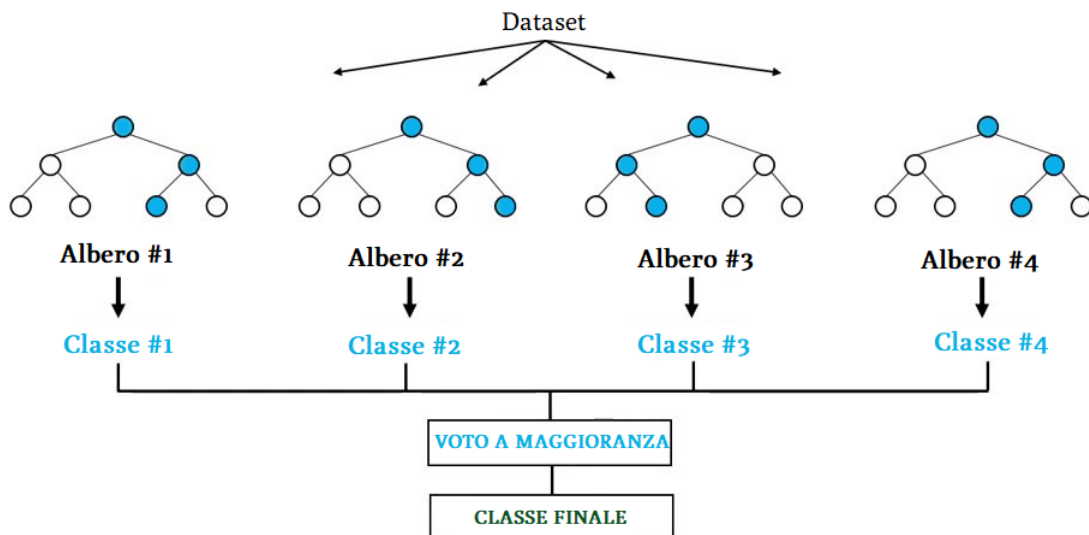


Figura 19.2: Illustrazione di Random Forest con $k = 4$ alberi.

²Così vengono dette le colonne di \mathcal{D}

1/12/2021

20.1 Multilayered Perceptron

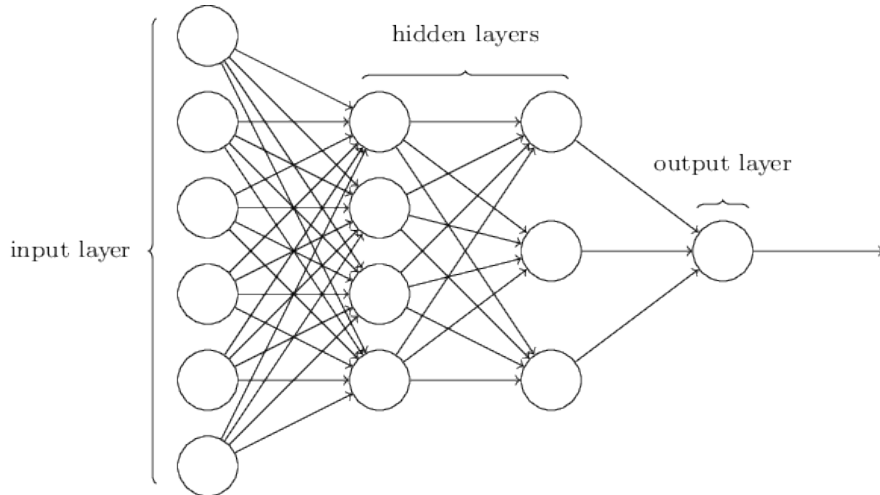


Figura 20.1: Perceptron multistrato tramite neurone di McCulloch e Pitts

Abbiamo già affrontato i limiti di Perceptron, su cui adesso torniamo brevemente. Anche i kernel RBF hanno alcune limitazioni significative, ed in particolare non sono adatti quando la dimensione del dataset è elevata.

Per risolvere problemi con dati ad alta dimensionalità come l'elaborazione di segnali una possibile idea è quella di utilizzare il dataset per apprendere anche la funzione $\phi(x, z)$ del kernel; essa stessa però dipende da dei parametri. Una possibile strategia si basa sul modello di **neurone di McCulloch e Pitts**: questo descrive il funzionamento di un neurone tramite una funzione della forma $\sigma(\underline{w}^t \cdot \underline{x} + b)$ con σ è una funzione *non* lineare (nelle prime pubblicazioni su questa idea σ ha la forma di un gradino di Heaviside).

Diverse unità il cui comportamento è descritto dal modello in questione possono essere combinate tra di loro come rappresentato nella Figura 20.1. Ciascun livello (*layer*) si associa ad un vettore in \mathbb{R}^ℓ con ℓ la numerosità dello strato, che si può interpretare come una rappresentazione del vettore d'ingresso \underline{x} . Per lo strato k -esimo tale rappresentazione si può pensare come una $\phi_k(x) = \sigma(\underline{w}_k \cdot \underline{x} + b_k)$ detta attivazione, mentre le σ si dicono *funzioni di attivazione*. Con la rappresentazione complessiva $\phi(x)$ si può costruire un Perceptron avente funzione di classificazione di questa forma:

$$f(x) = \text{SGN}(\underline{w}^t \cdot \phi(x) + b)$$

Quest'ultima è lineare rispetto alla rappresentazione, la quale se le funzioni di attivazione sono non lineari¹, fa sì che la mappa tra lo strato d'ingresso e quello di uscita sia non lineare.

¹È importante che le σ non siano lineari, perché in caso contrario la $\phi(x)$ sarebbe una composizione di funzioni lineari che è a sua volta una funzione lineare; così facendo non si avrebbe nessun guadagno rispetto al Perceptron classico.

Osserviamo che ciascuno strato insieme al successivo formano localmente un grafo bipartito completamente connesso; il numero di unità in un singolo livello viene detto *ampiezza* (*width*), mentre il numero di strati si chiama *profondità* (*depth*). I livelli intermedi tra quello d'ingresso e quello di uscita si dicono essere *nascosti*, e l'algoritmo che opera in questo modo è chiamato **multilayered Perceptron**, noto anche come *Feedforward Neural Network*.

Rispetto ai Kernel RBF si ha un vantaggio nell'uso di questa tecnica: apprendere il γ da usare come parametro del kernel non è un'opzione perché si dimostra causare un elevato overfitting del training set, mentre con questa nuova rappresentazione le funzioni σ ed i pesi degli archi \underline{w}_k possono essere appresi, e si parla di *representation learning*. Questo permette di avere una buona rappresentazione a seconda del problema da risolvere e di quello che i dati suggeriscano sia la cosa migliore, cioè senza dover usare conoscenza del problema. A seguire si riportano alcuni risultati rilevanti che supportano l'importanza della tecnica appena descritta.

Teorema 20.1.1. Se la larghezza di una Feedforward Neural Network è sufficientemente grande, questa può essere usata per approssimare qualsiasi funzione “abbastanza” regolare, ad esempio Lipschitziana.

Teorema 20.1.2. Una Feedforward Neural Network con un singolo strato nascosto può essere usata per realizzare una qualsiasi funzione booleana.

Dimostrazione. Una funzione booleana è una $f: \{0, 1\}^p \mapsto \{0, 1\}$, ed assumiamo che σ sia un gradino di Heaviside. Si consideri un'architettura formata da un singolo layer. In primo luogo dimostriamo che con una rete di questo tipo è possibile realizzare le due funzioni logiche elementari AND ed OR.

Assumiamo di voler rappresentare in \mathbb{R}^2 (ie $p = 2$) la funzione AND: questa avrà un grafico composto da quattro punti posizionati in $(x_1, x_2) = (0, 0)$, $(x_1, x_2) = (0, 1)$ e via dicendo. Questa presenterà solo un punto di classe positiva, che è quello in $(x_1, x_2) = (1, 1)$, mentre tutti gli altri saranno di classe negativa. L'insieme di punti così ottenuto è linearmente separabile e quindi si può sicuramente realizzare tramite PERCEPTRON. Analoghe considerazioni valgono per la funzione OR.

Ciascuna funzione booleana può essere scritta nella forma prodotto di somme (CNF), cioè come congiunzione di clausole, e si usa una unità nascosta che realizza una OR per ogni singola clausola, ed infine il livello di uscita è un insieme di unità che realizzano la AND. \square

20.2 Apprendimento in Feedforward Neural Networks

Indichiamo l'output dell'intera rete $\nu(x, \underline{w})$, che non necessariamente è la predizione, e dove \underline{w} sono tutti i pesi. Nell'esempio di Figura 20.2 si definiscono due matrici di pesi: una $\underline{W}_1 \in M(23 \times 4, \mathbb{R})$ associata agli archi dall'input layer verso il singolo strato interno, ed una seconda dall'hidden layer verso l'output layer, $\underline{W}_2 \in \mathbb{R}^4$. Ogni unità del livello nascosto calcola una rappresentazione $\phi(x) = \sigma(\underline{W}_1 \cdot \underline{x} + \underline{b}_1)$, con $\underline{b}_1 \in \mathbb{R}^4$ un vettore di offsets. Una scelta tipica per σ è la seguente:

$$\sigma(a) := \text{MAX}\{0, a\}$$

Invece $\nu(x) = \underline{W}_2^t \cdot \phi(\underline{x}) + b_2$ con b_2 uno scalare. Indicheremo a seguire con W l'insieme di tutti i pesi, ossia $W = \{W_1, b_1, W_2, b_2\}$. Il problema dell'apprendimento è quello di

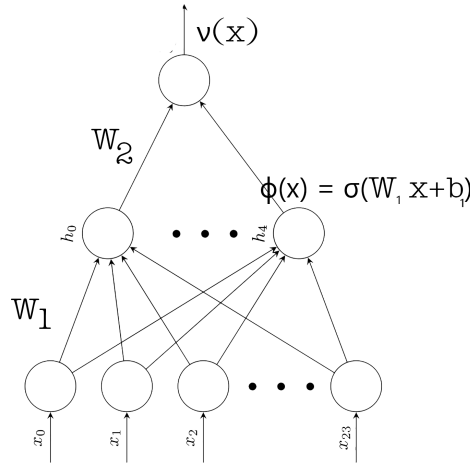


Figura 20.2: Notazione utilizzata apprendimento livelli

trovare W a partire dai dati, considerando come spazio delle ipotesi il seguente (sempre riferito all'esempio dell'immagine):

$$\mathcal{H} = \{ \Theta(\nu(x, W)) \mid W \in \mathbb{R}^{101} \}$$

Qui sopra Θ è una *funzione di risposta* che traduce l'output della rete in un parametro della distribuzione $\mathbb{P}(Y|X)$, e nel caso della classificazione binaria è sensato prendere $\Theta = \text{SGN}$.

Esempio 20.1: Funzione di risposta per regressione

La regressione è il problema di predire un numero reale a partire da un dataset. In questo caso è sufficiente assumere che $\Theta(\nu) = \nu$, cioè che Θ sia la funzione identità. Si assume così che $\mathbb{P}_{Y|X} \sim \mathcal{N}(\nu(x, W), 1)$. Si immagina quindi che ν , valore ottenuto dalla rete, una volta trasformato tramite la funzione di risposta in un parametro della distribuzione congiunta sopra menzionata. ■

Esempio 20.2: Funzione di risposta per classificazione binaria

Nel caso della classificazione binaria la distribuzione congiunta $\mathbb{P}_{Y|X}$ dev'essere una Bernoulli dato che Y assume solo due possibili valori, pertanto la Θ deve avere come immagine l'insieme $[0, 1]$. In questo caso una scelta comune per la funzione di risposta è la seguente:

$$\Theta(\nu) = \frac{1}{1 + e^{-\nu}}$$

Questa scelta di Θ viene chiamata anche *funzione logistica*, e permette di trasformare un qualunque numero reale in una probabilità, che nel caso in analisi si interpreta come $\mathbb{P}(Y = 1 | X)$. ■

Con questa trasformazione, cioè tramite l'introduzione di una funzione di risposta, permette di trattare il problema dell'apprendimento in uno di massima verosimiglianza. In tal senso si può dimostrare che la stima tramite questa tecnica è equivalente alla minimizzazione di una funzione di LOSS empirica, se taluna è scelta opportunamente, ie:

- **Regressione:** $\text{Loss}(\theta, y) = (\theta - y)^2$, detta anche *squared loss*.

- **Classificazione binaria:** $\text{LOSS}(\theta, y) = y \log(\theta) + (1 - y) \log(1 - \theta)$, detta anche *binary cross-entropy*.

Nel secondo caso, quando $y = 1$ si ha che $\text{LOSS}(\theta, y) = \log(\theta)$, mentre per $y = 0$ si ha $\text{LOSS}(\theta, y) = 1 - \log(\theta)$.

Minimizzazione della loss empirica

Per la scelta della funzione di risposta si può usare una strategia nota come *Minimizzazione della loss empirica*, che prevede di prendere tra tutte le possibili funzioni di risposta quella che minimizza la LOSS sul dataset, cioè trovare la quantità

$$\text{LOSS}(W) := \min_W \left\{ \frac{1}{n} \cdot \sum_{i=1}^n \text{LOSS}(\theta(x^{(i)}, W), y^{(i)}) \right\}$$

La funzione sopra definita in generale non è convessa, e nei casi d'interesse pratico non lo è nemmeno localmente per cui non si possono usare i metodi classici di ottimizzazione. Si usa invece una strategia nota come **discesa del gradiente**. Esiste un risultato che garantisce che se $W \in \mathbb{R}^m$ ed $m > n$ esiste un minimizzatore nullo, cioè $\exists W^* : \text{LOSS}(W) = 0$, e che il metodo del gradiente vi converge. Di tale metodo ne applicheremo una variante stocastica.

Metodo del gradiente stocastico

Similmente a quanto avveniva con Perceptron si costruisce una successione di pesi W_t , con $t = 0, 1, \dots, T$. Indipendentemente dal test l'aggiornamento dell'insieme dei pesi avviene secondo la seguente legge:

$$W_{t+1} = W_t - \gamma \nabla_W \text{LOSS}^{(i)}(W_t) \quad (20.1)$$

Dove $\text{LOSS}^{(i)} := \text{LOSS}(\theta(x^{(i)}, W), y^{(i)})$. Intuitivamente si guarda se modificando i pesi in una certa direzione si fa crescere oppure diminuire la LOSS. Ricordiamo che ∇_W è un operatore che esegue la derivata parziale della funzione a cui si applica rispetto a tutti i pesi. Il metodo è insignito dell'attributo di “stocastico” perché i è campionato uniformemente in $\{1, 2, \dots, n\}$. Un modo per realizzare tale campionamento è permutare l'insieme $[1, n] \cap \mathbb{N}$ e prendere i dall'insieme riordinato; una volta prelevate n di queste si costruisce una nuova permutazione e si ripete (simile ad epoche del PERCEPTRON).

A questo punto il problema principale diventa quello di calcolare il gradiente. Noi trattiamo un caso semplice, che è quello in cui non sono presenti unità nascoste (rete neurale priva di hidden layer). In questo ci torna utile la regola della catena per derivazione delle funzioni composte. Applicando questa si trova che

$$\frac{\partial \text{LOSS}}{\partial W} = \frac{\partial \text{LOSS}}{\partial \theta} \cdot \frac{\partial \theta}{\partial \nu} \cdot \frac{\partial \nu}{\partial W} \quad (20.2)$$

Qui i primi due termini sono scalari mentre l'ultimo è un vettore. Nel caso della classificazione binaria $\text{LOSS}(\theta, y) = y \log(\theta) + (1 - y) \log(1 - \theta)$ per quanto già osservato in

precedenza, mentre θ è la funzione logistica. Segue quindi che

$$\frac{\partial \text{Loss}}{\partial \theta} = \frac{y}{\theta} - \frac{1-y}{1-\theta} = \frac{y-\theta}{\theta(1-\theta)} \quad (20.3)$$

$$\frac{\partial \theta}{\partial \nu} = \frac{1}{1+e^{-\nu}} \cdot \left(1 - \frac{1}{1+e^{-\nu}}\right) \quad (20.4)$$

$$\frac{\partial \nu}{\partial W} = \frac{\partial}{\partial W} \cdot (W^t \cdot \underline{x} + b) = \underline{x} \quad (20.5)$$

Per quanto riguarda l'Equazione 20.4 si dimostri per esercizio che la funzione logistica è soluzione dell'equazione differenziale $\dot{\theta} = \theta \cdot (1 - \theta)$. Sostituendo quanto trovato in 20.2 si trova che

$$\frac{\partial \text{Loss}}{\partial W} = \frac{y-\theta}{\theta(1-\theta)} \cdot \theta(1-\theta) \cdot \underline{x} = (y-\theta)\underline{x} \quad (20.6)$$

Questo può adesso essere sostituito in 20.1, e da qui emerge ancora di più la somiglianza tra questo metodo e PERCEPTRON: ad esempio se la classe y è positiva e θ è vicino a 0, l'insieme dei pesi viene fatto calare di un passo $\gamma \underline{x}$. Il parametro γ viene chiamato *learning rate*, e questo può essere fissato oppure cambiare, nel qual caso diventa un γ_t .

Quando sono presenti dei layers nascosti come in MULTILAYERED-PERCEPTRON il calcolo del gradiente come qui sopra non è più valido e si usa invece un algoritmo noto come BACK-PROPAGATION, che cerca di determinare quanto ν sia sensibile alla variazione dei vari pesi della rete (nota che ogni volta che si attraversa un layer si incontra una non-linearità). Questo algoritmo basa il proprio funzionamento su uno scambio di messaggi sulla rete, dove ciascuno di questi è una derivata opportuna

Generalizzazione al caso multiclasse

Intuitivamente se per due sole classi gli esempi positivi e quelli negativi possono essere divisi tramite un iperpiano, quando se ne hanno tre o più occorrono almeno due iperpiani in modo da poter produrre una regione di classe 1, una di classe 2 ed una di classe 3.

Per semplicità quando in MULTILAYERED-PERCEPTRON si calcola $\nu(x)$ conviene far sì che questo non sia più uno scalare, bensì un vettore con una componente per ciascuna classe. L'idea è che la classe k avente ν_k più grande domina sulle altre.

9/12/2021

21.1 Backpropagation per Multilayered Perceptron

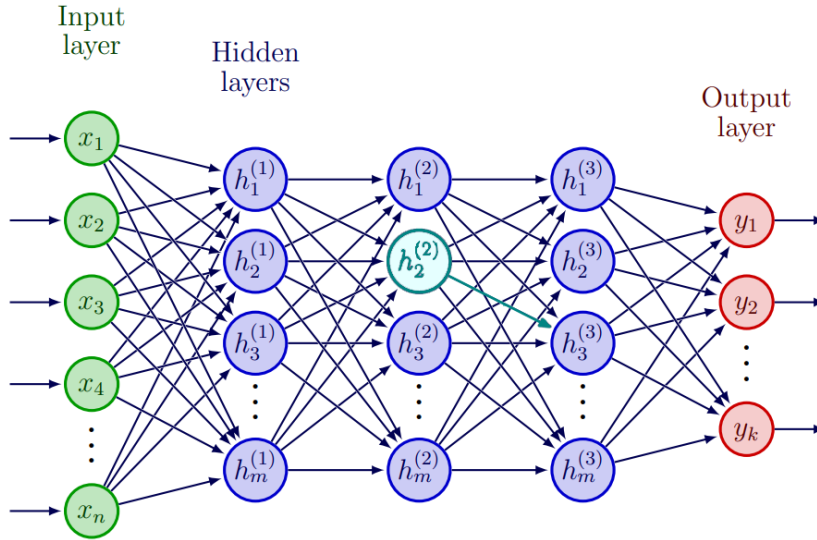


Figura 21.1: Riferimento per Backpropagation. Nodo ed arco generico evidenziato in blu
In questa sezione generalizziamo le tecniche viste in chiusura della scorsa lezione al caso in cui la rete possenga anche degli strati nascosti. Si consideri una generica unità nascosta ed uno qualsiasi dei pesi w_{jk} associati, cfr Figura 21.1. In questo contesto

$$a_j = \sum_{k \in \text{PA}(j)} w_{jk} \cdot \phi(k) + b_j \quad (21.1)$$

$$\phi(j) = \sigma(a_j) \quad (21.2)$$

Per quanto riguarda il gradiente, riprendiamo la funzione di LOSS nel caso binario definita la lezione precedente e calcoliamo la derivata parziale rispetto ad un certo peso; sia a_j l'attivazione del nodo a cui punta il secondo estremo dell'arco scelto, allora per la regola della derivazione di funzioni composte:

$$\frac{\partial \text{LOSS}}{\partial w_{jk}} = \frac{\partial \text{LOSS}}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{jk}} \stackrel{(a)}{=} \delta_j \cdot \frac{\partial a_j}{\partial w_{jk}} = \delta_j \cdot \phi_k$$

La relazione (a) segue definendo δ_j , detto *errore generalizzato*, come la derivata parziale al primo fattore del membro di sinistra. Osserviamo adesso che la non linearità dovuta all'arco scelto, quando varia, influenza anche tutte le unità dello strato successivo. Tenendo conto di questo, δ_j si può riscrivere nel modo seguente:

$$\delta_j = \frac{\partial \text{LOSS}}{\partial \phi_j} \cdot \frac{\partial a_j}{\partial w_{jk}} = \left(\sum_{r \in k.\text{CHILD}} \frac{\partial \text{LOSS}}{\partial a_r} \cdot \frac{\partial a_r}{\partial \phi_j} \right) \cdot \frac{\partial a_j}{\partial w_{jk}} \stackrel{(b)}{=} \left(\sum_{r \in k.\text{CHILD}} \delta_r \cdot w_{rj} \right) \sigma'(a_j)$$

Per quanto riguarda la relazione (b), questa segue dalla definizione data di δ_r , e dal fatto che la derivata parziale al secondo fattore nella sommatoria contiene a_r che si scrive in questo modo:

$$a_r = \sum_{j \in \text{PA}(r)} w_{rj} \phi_j + b_r$$

Si osservi che i δ si propagano dall'output verso lo strato d'ingresso, e quindi serve un ordine topologico inverso, mentre la ϕ si propaga nella direzione opposta, detta *forward*. Rimane adesso il problema di come calcolare il primo δ per far partire la ricorsione *backward*. Sia esso δ_o ; allora questo si può calcolare con lo stesso meccanismo che si usa per una rete priva di unità nascoste, i.e. $\delta_o = \Theta(\nu)$, con ν la somma pesata di ciò che arriva in output e Θ la funzione logistica, se si vuole fare classificazione binaria. Tramite calcoli analoghi a quelli dell'Equazione 20.6 si trova che $\delta_o = y - \theta$. Se si scelgono opportunamente θ e la forma della LOSS è garantito dalla teoria che δ_o assume sempre questo valore (ad esempio con θ =funzione logistica e LOSS=cross-entropy).

L'algoritmo basato sulla strategia descritta in questa sezione si chiama BACKPROPAGATION, e risale al 1986. Questo continua ad usare la regola di aggiornamento dei pesi data per STOCHASTIC-GRADIENT-DESCENT, Equazione 20.1, ma non si può inizializzare l'insieme dei pesi con tutti zeri, per far sì che il gradiente iniziale non sia subito nullo (sella per la LOSS). In tal senso un'idea plausibile è quello di inizializzare i pesi in maniera casuale per rompere le simmetrie della rete¹, e per motivi di efficienza temporale e per com'è fatta $\sigma(a)$ conviene che i pesi generati siano tutti vicini a zero inizialmente; infine il valore scelto per w_{jk} non solo dev'essere piccolo ma deve anche tenere conto del numero di padri e di figli del nodo j . Una convenzione in tal senso è quella di generare i pesi uniformemente in $[-d_j, d_j]$ dove $d_j := \sqrt{|\text{PA}(j)| + |\text{CHILDREN}(j)|}$ per ogni unità j della rete.

Generalizzazione al caso multiclasse

Per generalizzare quanto appena descritto da classificazione binaria al caso multiclasse occorre trovare θ e LOSS che presi insieme garantiscano come prima che $\delta_o = y - \theta$.

Il primo cambiamento è che ν passa dall'essere uno scalare ad un vettore con una componente (output) per ciascuna classe. Anche la risposta θ dev'essere generalizzata e si ha un cambiamento analogo rispetto a quello della ν : generalizziamo così la funzione logistica ponendo per definizione:

$$\theta_j := \frac{e^{\nu_j}}{\sum_{r=1}^K e^{\nu_r}}$$

Questo θ_j , detto *softmax*, si comporta come una stima di $\mathbb{P}(Y = j | X)$, ed il denominatore serve a garantire che la somma di tutti i θ_j faccia 1.

Per quanto riguarda la LOSS, anche questa non sarà che una generalizzazione della cross-entropy; si pone per definizione:

$$\text{LOSS}(\underline{\theta}, y) := - \sum_{j=1}^k \mathbb{1}\{y = j\} \log(\theta_j)$$

¹Nota che se in un livello si scambiano due unità ed i rispettivi archi entranti ed uscenti la W ottenuta è diversa ma la LOSS è la stessa.

Con questa scelta di funzione di risposta e LOSS si ottiene $\delta_{oj} = \mathbb{1}\{y = j\} - \theta_j$.

Osservazione 21.1.1. L'algoritmo di STOCHASTIC-GRADIENT-DESCENT può essere troppo rumoroso. La soluzione consiste nel considerare la seguente legge di aggiornamento: $w_{k+1} = w_k - \delta \nabla_W \text{LOSS}_t$ dove LOSS_t non è più riferita ad un singolo esempio ma ad un gruppo di questi; nello specifico dato un insieme di punti $\text{MB}(t)$ (sottoinsieme casuale di insiemi di dimensione B detta *minibatch size*) si pone:

$$\text{LOSS}_t := \sum_{i \in \text{MB}(t)} \text{LOSS}(\theta^{(i)}, y^{(i)})$$

Prendendo $B = 1$ si riottiene lo STOCHASTIC-GRADIENT-DESCENT, con $B = n$ si ottiene la GRADIENT-DESCENT mentre per $1 < B < n$ si possono ottenere algoritmi con una rumorosità intermedia.

21.2 Combinazione di classificatori semplici

Bibliografia

- [RN10] Stuart J. Russel, Peter Norvig, *Artificial Intelligence, A modern approach. Third Edition*, 2010.
- [PDR20] Laura Poggiolini, Niccolò Della Rocca, *Note di metodi matematici e probabilistici*, 2020.
- [MN98] Andrew McCallum, Kamal Nigam, *A Comparison of Event Models for Naive Bayes Text Classification*, pp. 6, 1998
- [J97] F. Jensen. *Introduction to Bayesian Networks*, 1997
- [B12] D. Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.
- [CQ93] J.R. Quinlan. *C4.5 Programs for Machine Learning*. Morgan Kaufmann Publishers, Inc., 1993.