

Apprendimento di reti bayesiane da database di esempi

Enrico Papalini

papalini@biancaneve.ing.unifi.it

Michele Piccini

mpiccini@biancaneve.ing.unifi.it

elaborato per l'esame di "intelligenza artificiale"

corso di laurea in Ingegneria Informatica

Università di Firenze

a.a. 1996/97

Sommario

Le reti di Bayes sono modelli grafici di probabilità in cui i nodi rappresentano variabili aleatorie e gli archi le dipendenze causali fra variabili. Questa relazione descrive come apprendere la struttura grafica e la distribuzione di probabilità connesse alla rete a partire da un database di realizzazioni delle variabili. Viene presentata una tecnica basata su due passi: nel primo passo viene appresa la struttura selezionando il modello grafico che massimizza una appropriata funzione di costo, nel secondo vengono apprese le tabelle di probabilità associate a ciascun nodo tramite un approccio Bayesiano. Sono illustrati i fondamenti teorici della metodologia e i risultati ottenuti applicandola a database provenienti sia da processi simulati e che reali.

1 Introduzione alla stima bayesiana

Supponiamo di avere un insieme di variabili aleatorie $\mathbf{X} = \{X_1, \dots, X_n\}$ ed un insieme di loro realizzazioni $D = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, detto database di esempi. Supponiamo che sia ignoto il modello \mathbf{m} che descrive tale insieme di variabili e sia ignota la distribuzione di probabilità congiunta di questo insieme. Denotiamo questa distribuzione con $p(\mathbf{X} | \theta_{\mathbf{m}}, \mathbf{m})$, dove $\theta_{\mathbf{m}}$ rappresenta i parametri del modello \mathbf{m} . Il problema da risolvere è quello di stimare la distribuzione, noto il database D .

L'approccio bayesiano per risolvere il problema di stima proposto prevede di definire una variabile aleatoria discreta \mathbf{M} i cui stati sono i possibili \mathbf{m} , che codifica l'incertezza sulla struttura del modello tramite la distribuzione di probabilità a priori $p(\mathbf{M} = \mathbf{m})$. Inoltre, per ogni modello \mathbf{m} viene definita una variabile aleatoria continua $\mathbf{T}_{\mathbf{m}}$ che codifica i possibili valori che i suoi parametri possono assumere, con un'incertezza a priori data dalla densità di probabilità $p(\mathbf{T}_{\mathbf{m}} = \theta_{\mathbf{m}} | \mathbf{m})$.

Dato un database di esempi D , il teorema di Bayes permette di calcolare le distribuzioni a posteriori per le due variabili aleatorie \mathbf{M} e $\mathbf{T}_{\mathbf{m}}$:

$$p(\mathbf{m} | D) = \frac{p(\mathbf{m})p(D | \mathbf{m})}{\sum_{\bar{\mathbf{m}}} p(\bar{\mathbf{m}})p(D | \bar{\mathbf{m}})} \quad (1)$$

$$p(\theta_{\mathbf{m}} | \mathbf{m}, D) = \frac{p(\theta_{\mathbf{m}} | \mathbf{m})p(D | \theta_{\mathbf{m}}, \mathbf{m})}{\int p(\theta_{\mathbf{m}} | \mathbf{m})p(D | \theta_{\mathbf{m}}, \mathbf{m})d\theta_{\mathbf{m}}} \quad (2)$$

Dopo aver scelto una distribuzione a priori per \mathbf{X} , condizionata al modello ed ai suoi parametri $p(\mathbf{x} | \theta_{\mathbf{m}}, \mathbf{m})$, la stima della distribuzione a posteriori $p(D | \mathbf{m})$ si trova calcolando il valore atteso del prior rispetto $p(\mathbf{m} | D)$ e $p(\theta_{\mathbf{m}} | D, \mathbf{m})$:

$$p(\mathbf{x} | D) = \sum_{\mathbf{m}} p(\mathbf{m} | D) \int p(\mathbf{x} | \theta_{\mathbf{m}}, \mathbf{m})p(\theta_{\mathbf{m}} | D, \mathbf{m})d\theta_{\mathbf{m}} \quad (3)$$

Purtroppo questo approccio bayesiano puro non può essere applicato quando il numero dei possibili modelli rende il calcolo della sommatoria intrattabile. Anche il caso dell'apprendimento delle reti bayesiane fa parte di questa categoria. Si aggira il problema facendo l'ipotesi che la distribuzione $p(\mathbf{m} | D)$ sia localizzata attorno ad un particolare modello $\hat{\mathbf{m}}$. In questo caso, una volta selezionato $\hat{\mathbf{m}}$, la stima della distribuzione a posteriori di \mathbf{X} si riduce a:

$$p(\mathbf{x} | D, \hat{\mathbf{m}}) = \int p(\mathbf{x} | \theta_{\hat{\mathbf{m}}}, \hat{\mathbf{m}})p(\theta_{\hat{\mathbf{m}}} | D, \hat{\mathbf{m}})d\theta_{\hat{\mathbf{m}}} \quad (4)$$

Per selezionare $\hat{\mathbf{m}}$ si introduce una funzione che applicata ad un modello restituisca un "punteggio" che sia tanto più alto quanto il modello è più prossimo a $\hat{\mathbf{m}}$. È naturale utilizzare una funzione $\text{SCORE}(\mathbf{m})$ derivata da $p(\mathbf{m} | D)$, ad esempio il suo logaritmo, che per il teorema di Bayes può essere messo in relazione con il prior sul modello e sui dati:

$$\begin{aligned} \text{SCORE}(\mathbf{m}) &= \log p(\mathbf{m} | D) \\ &= \log p(\mathbf{m}) + \log p(D | \mathbf{m}) - \log p(D) \\ &\simeq \log p(D | \mathbf{m}) \end{aligned} \quad (5)$$

L'approssimazione compiuta deriva dal fatto che $\log p(D)$ è una costante e che il prior sul modello $\log p(\mathbf{m})$ può anch'esso essere supposto costante se si fa l'ipotesi che ogni modello sia equiprobabile (completa ignoranza a priori sul modello).

Riassumendo, il problema della stima della distribuzione di probabilità congiunta di \mathbf{X} può essere affrontato grazie alla seguente procedura:

Procedura di stima della distribuzione di probabilità congiunta di \mathbf{X} .

passo 0: scegliere opportunamente le distribuzioni a priori per \mathbf{X} e per i parametri,

passo 1: trovare, tramite un opportuno algoritmo di ricerca, il modello $\hat{\mathbf{m}}$ che massimizza $\text{SCORE}(\mathbf{m})$,

passo 2: calcolare la distribuzione a posteriori $p(\mathbf{x} | D, \hat{\mathbf{m}})$ utilizzando l'approccio bayesiano descritto.

2 Introduzione alle reti di Bayes

Una rete bayesiana è un modello grafico che codifica la distribuzione congiunta di probabilità di un insieme di variabili aleatorie $\mathbf{X} = \{X_1, \dots, X_n\}$. Questa consiste in

1. un grafo diretto aciclico S (detto *struttura*) in cui ogni nodo è associato ad un'unica variabile aleatoria X_i e ogni arco rappresenta la dipendenza condizionale fra i nodi che unisce,
2. un insieme P di *distribuzioni locali di probabilità*, ciascuna associata ad una variabile aleatoria X_i e condizionata dalle variabili corrispondenti ai nodi sorgenti degli archi entranti nel nodo a cui è associata X_i .

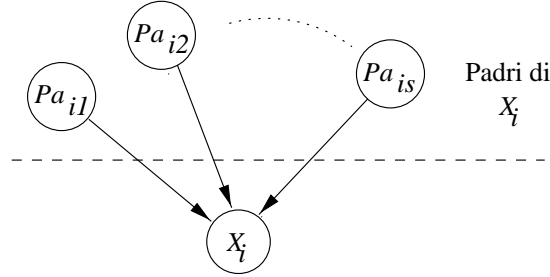


Figura 1: Schema della relazione fra un nodo ed i padri in una generica rete bayesiana.

La mancanza di un arco fra due nodi riflette la loro indipendenza condizionale. Al contrario, la presenza di un arco dal nodo X_i al nodo X_j può essere interpretata come il fatto che X_i sia causa diretta di X_j .

Data una struttura S e le distribuzioni locali di probabilità di ciascun nodo $p(X_i | \mathbf{Pa}_i)$, ove \mathbf{Pa}_i rappresenta l'insieme dei nodi padri di X_i , la probabilità di distribuzione congiunta $p(\mathbf{X})$ si ottiene da

$$p(\mathbf{X}) = \prod_{i=1}^n p(X_i | \mathbf{Pa}_i) \quad (6)$$

ed è evidente come la coppia (S, P) codifichi in modo univoco $p(\mathbf{X})$.

3 Apprendimento di reti bayesiane

Per utilizzare la reti di Bayes come modello nel senso visto nella sezione 1, dobbiamo

1. associare ad ogni realizzazione della variabile \mathbf{M} una struttura S della rete costruita sull'insieme di variabili aleatorie X_i , scrivendo $\mathbf{M} = \mathbf{m}_S$,
2. associare ad ogni distribuzione locale di probabilità un numero finito di parametri che la descrivono, denominati θ_i , che tutti assieme formano i parametri del modello, $\theta_S = \{\theta_1, \dots, \theta_n\}$.

In questo modo l'equazione 6 può essere riscritta condizionandola al modello ed ai parametri:

$$p(\mathbf{X} | \mathbf{T}_S, \mathbf{M}_S) = \prod_{i=1}^n p(X_i | \mathbf{Pa}_i, \mathbf{T}_i, \mathbf{M}_S) \quad (7)$$

Per applicare la metodologia illustrata nella sezione 1, debbono essere calcolate le distribuzioni $p(D | \mathbf{m}_S)$ per poter implementare l'algoritmo di massimizzazione di SCORE(\mathbf{m}_S) e $p(\mathbf{x} | \mathbf{D}, \mathbf{m}_S)$ per poter stimare la distribuzione di \mathbf{X} . Il loro calcolo può esser compiuto in forma chiusa applicando le seguenti ipotesi sulla rete bayesiana.

hp1. Ogni variabile X_i è *discreta* (può assumere gli stati x_i^k , con $k = 1, \dots, r_i$) e la sua distribuzione locale di probabilità è una collezione di *distribuzioni multinomiali*, una per ogni stato \mathbf{pa}_i^j delle variabili padri ($j = 1, \dots, q_i$ con $q_i = \prod_{X_s \in \mathbf{Pa}_i} r_s$):

$$p(x_i^k | \mathbf{pa}_i^j, \theta_i, \mathbf{m}_i) = \theta_{ijk} > 0 \quad (8)$$

tale che $\sum_{k=1}^{r_i} \theta_{ijk} = 1 \quad \forall i, j$. Definiamo anche due vettori di parametri $\theta_{ij} = \{\theta_{ijk}\}_{k=1}^{r_i}$ e $\theta_i = \{\theta_{ij}\}_{j=1}^{q_i}$ per semplificare la notazione.

hp2. I parametri θ_{ij} sono *mutuamente indipendenti*. Ciò comporta, come illustrato in [SL-90] che il problema diviene separabile nel senso espresso dalla seguente equazione:

$$p(\theta_S | \mathbf{m}_S) = \prod_{i=1}^n \prod_{j=1}^{q_i} p(\theta_{ij} | \mathbf{m}_S) \quad (9)$$

hp3. Ogni insieme di parametri θ_{ij} ha come distribuzione la coniugata della distribuzione della variabile X_i corrispondente. In questo caso è la *distribuzione di Dirichlet*:

$$p(\theta_{ij} | \mathbf{m}_S) = \frac{\Gamma(\alpha_{ij})}{\prod_{k=1}^{r_i} \Gamma(\alpha_{ijk})} \prod_{k=1}^{r_i} \theta_{ijk}^{\alpha_{ijk}-1} = \text{Dir}(\theta_{ij}, \alpha_{ij1}, \dots, \alpha_{ijr_i}) \quad (10)$$

ove gli α_{ijk} sono gli iperparametri della distribuzione di Dirichlet, tali che $\alpha_{ijk} > 0 \quad \forall i, j, k$ e che $\alpha_{ij} = \sum_{k=1}^{r_i} \alpha_{ijk}$.

Sono necessarie anche delle ipotesi sul database di esempi D :

hp4. D è *completo*, quindi non ci sono osservazioni mancanti.

hp5. Il campione D deve essere estratto da un fenomeno il cui modello è una struttura S di una rete di Bayes.

Sotto queste ipotesi in [CH-92] sono riportati i seguenti risultati:

$$p(D | \mathbf{m}_S) = \prod_{i=1}^n \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + N_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ijk} + N_{ijk})}{\Gamma(\alpha_{ijk})} \quad (11)$$

$$p(x_i^k | \mathbf{pa}_i^j, \mathbf{D}, \mathbf{m}_S) = \frac{\alpha_{ijk} + N_{ijk}}{\alpha_{ij} + N_{ij}} \quad (12)$$

ove N_{ijk} è il numero delle volte che nel database D si ha che $X_i = x_i^k$ e che $\mathbf{Pa}_i = \mathbf{pa}_i^j$, mentre $N_{ij} = \sum_{k=1}^{r_i} N_{ijk}$. Dall'equazione 11 è immediato ricavare che $\text{SCORE}(\mathbf{m}_S)$ può essere calcolato come

$$\text{SCORE}(\mathbf{m}_S) = \sum_{i=1}^n \sum_{j=1}^{q_i} \left[\log \left(\frac{\Gamma(\alpha_{ij})}{\Gamma(\alpha_{ij} + N_{ij})} \right) + \sum_{k=1}^{r_i} \log \left(\frac{\Gamma(\alpha_{ijk} + N_{ijk})}{\Gamma(\alpha_{ijk})} \right) \right] \quad (13)$$

Per poter avere una formula computabile bisogna assegnare dei valori agli α_{ijk} . Questi iperparametri codificano la conoscenza a priori che l'utente ha sui parametri delle probabilità associate alla rete. Dato che abbiamo supposto una completa ignoranza, è logico porre $\alpha_{ijk} = \frac{\alpha}{q_i r_i}$ che equivale

a porre $\alpha_i = \sum_{j=1}^{q_i} \sum_{k=1}^{r_i} \frac{\alpha}{q_i r_i} = \alpha$, interpretabile come l'equiprobabilità di ogni istanza dello spazio delle probabilità congiunte su \mathbf{X}_i e \mathbf{Pa}_i . Resta così da assegnare un unico "iper"iperparametro α che in letteratura è chiamato *dimensione di un campione equivalente*.

L'equazione 13 permette di realizzare il **passo 1** della procedura di apprendimento illustrata nella sezione 1 che calcola la struttura "migliore" rispetto a $\text{SCORE}(\mathbf{m}_S)$ e l'equazione 12 il **passo 2**, che stima i parametri della struttura selezionata al passo precedente.

4 Ricerca nello spazio delle strutture delle reti

Per poter effettivamente implementare la procedura esposta nella sezione 1 c'è bisogno di definire una metodologia di ricerca nello spazio delle strutture delle reti bayesiane associate all'insieme di variabili aleatorie X . Seguendo i risultati esposti in [HGC-95] è stata scelta una procedura di ricerca "hill-climbing" per cercare di massimizzare la funzione $\text{SCORE}(\mathbf{m}_S)$.

Scelta una struttura S è possibile valutare il guadagno di SCORE che si ha per ogni possibile variazione elementare degli archi, in modo da mantenere l'aciclicità del grafo. Queste variazioni sono l'aggiunta di un arco fra due nodi mutuamente indipendenti, la cancellazione di un arco fra due nodi dipendenti, il cambiamento di verso di un arco fra due nodi. Sfruttando il fatto che la

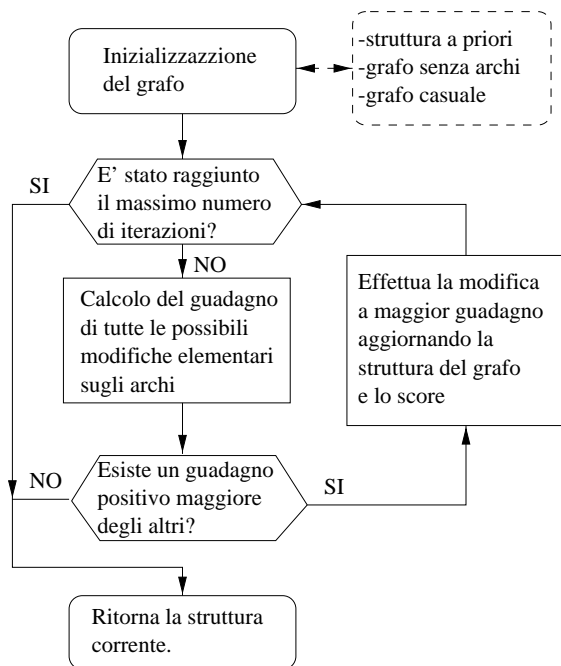


Figura 2: Schema della ricerca Hill-Climbing in uno spazio di grafi associati a reti bayesiane.

funzione di costo descritta dall'equazione 13 può essere scomposta nella somma di n addendi, ciascuno associato ad un nodo X_i ed ai suoi padri \mathbf{Pa}_i , la variazione di un solo arco della struttura S influirà al più su due addendi, relativi ai nodi sorgente e pozzo dell'arco variato. In particolare ciò accade soltanto se un arco della struttura viene invertito. Negli altri casi è sufficiente calcolare la variazione dell'addendo relativo al nodo pozzo del nuovo arco.

Dopo aver calcolato tutte le variazioni elementari possibili si effettua, se esiste, quella che porterebbe un guadagno positivo maggiore e si reitera il procedimento, dopo aver aggiornato il

nuovo SCORE. La ricerca termina nel caso in cui nessuna modifica faccia aumentare lo SCORE oppure se viene raggiunto il limite massimo del numero di iterazioni possibili. Lo schema della procedura di ricerca appena esposta si trova in figura 2.

Questo tipo di approccio necessita di un grafo di partenza. Candidati per questo possono essere il grafo privo di archi, che codifica la completa ignoranza sulle relazioni che intercorrono fra le variabili, un grafo aciclico costruito inserendo archi in modo casuale oppure una rete che rappresenti la conoscenza a priori posseduta sul dominio del problema.

5 Metodologia di analisi su reti simulate

Quanto esposto nelle precedenti sezioni è stato implementato utilizzando il linguaggio C++ in ambiente Linux. Dei problemi riscontrati in questa fase e delle scelte compiute verrà reso conto in appendice. Illustriamo adesso la metodologia di analisi dei risultati ottenuti tramite il nostro prototipo.

Una prima analisi è stata compiuta su reti bayesiane simulate. La procedura seguita, che incorpora quella per l'apprendimento esposta nella sezione 1, viene di seguito illustrata.

Procedura di apprendimento di una rete simulata.

- passo 1:** Generazione di una rete random, detta *gold network*, avente il numero dei nodi e il massimo numero di stati per nodo prefissati e con una specifica probabilità per l'inserimento degli archi fra i nodi.
- passo 2:** Campionamento della *gold network* in modo da ottenere un database di esempi D con un numero di campioni prestabilito.
- passo 3:** Perturbazione della *gold network* per ottenere una *prior network* da cui cominciare la ricerca. Il metodo di perturbazione utilizza una certa probabilità per modificare gli archi esistenti ed un'altra per aggiungere nuovi archi, con il vincolo di mantenere il grafo aciclico.
- passo 4:** Apprendimento strutturale, a partire da D e dalla *prior network*, utilizzando l'algoritmo esposto nella sezione 4.
- passo 5:** Apprendimento delle tabelle di probabilità associate a ciascun nodo tramite una procedura che implementa l'equazione 12.

Per completare l'analisi è stato necessario studiare un modo per misurare la "bontà" sia della struttura selezionata che delle tabelle apprese, rispetto a quelle della *gold network* di partenza. Per fare ciò ci siamo basati sui suggerimenti riportati in [HGC-95], in cui vengono utilizzati due tipi di misure:

1. La *differenza strutturale* fra due reti, che rappresenta il grado con cui la struttura appresa ha catturato le relazioni causali fra variabili. Definita la differenza simmetrica δ_i fra i padri di X_i in due differenti reti P e Q come

$$\delta_i = \#_{\{(\mathbf{Pa}_i^Q \cup \mathbf{Pa}_i^P) \setminus (\mathbf{Pa}_i^Q \cap \mathbf{Pa}_i^P)\}} \quad (14)$$

ove il simbolo $\#_{\mathbf{A}}$ sta per “numero degli elementi dell’insieme \mathbf{A} ”, la differenza strutturale δ si calcola sommando tutte le δ_i

$$\delta = \sum_{i=1}^n \delta_i \quad (15)$$

Si osservi come la differenza strutturale sia la misura del numero degli archi in cui le reti P e Q differiscono, contando due volte gli archi che sono stati invertiti nel passaggio da P a Q .

2. La *cross-entropia* fra due reti, che denota quanto bene la rete bayesiana appresa predurrà il prossimo campione del database D . Dette $p(\mathbf{X} | \mathbf{m}_P)$ e $p(\mathbf{X} | \mathbf{m}_Q)$ le distribuzioni congiunte di probabilità condizionate dalle reti P e Q , la cross-entropia $H(P, Q)$ è data da

$$H(P, Q) = \sum_{\mathbf{X}} p(\mathbf{X} | \mathbf{m}_P) \log \frac{p(\mathbf{X} | \mathbf{m}_P)}{p(\mathbf{X} | \mathbf{m}_Q)} \quad (16)$$

6 Analisi dell’apprendimento su reti simulate

Un esempio di risultato ottenuto si può osservare in figura 3, dove non sono state riportate le tabelle di probabilità associate ai nodi, in quanto troppo voluminose. Ad esempio, la tabella associata al nodo tre ha dimensioni 5×240 in quanto X_3 può assumere valori in 5 stati differenti ed i nodi padri \mathbf{Pa}_3 hanno in totale 240 stati. Si osservi come la rete appresa differisca dalla *gold network* di soli tre archi, tutti e tre cancellati rispetto all’originale. Questi archi fanno passare il numero degli stati dei padri di X_3 da 240 a 6, semplificando notevolmente la tabella di probabilità condizionate associate. Una spiegazione di questo comportamento è data dal fatto che gli algoritmi proposti tendono a selezionare le reti più semplici rispetto ad un determinato database D . Se vi aggiungiamo il fatto che D non ha in questo caso una dimensione elevata (soltanto 4000 campioni), appare evidente come l’algoritmo abbia semplificato la rete per l’assenza di sufficienti informazioni per cogliere la complessità del terzo nodo. Resta comunque il fatto che il resto della rete è stato correttamente riconosciuto.

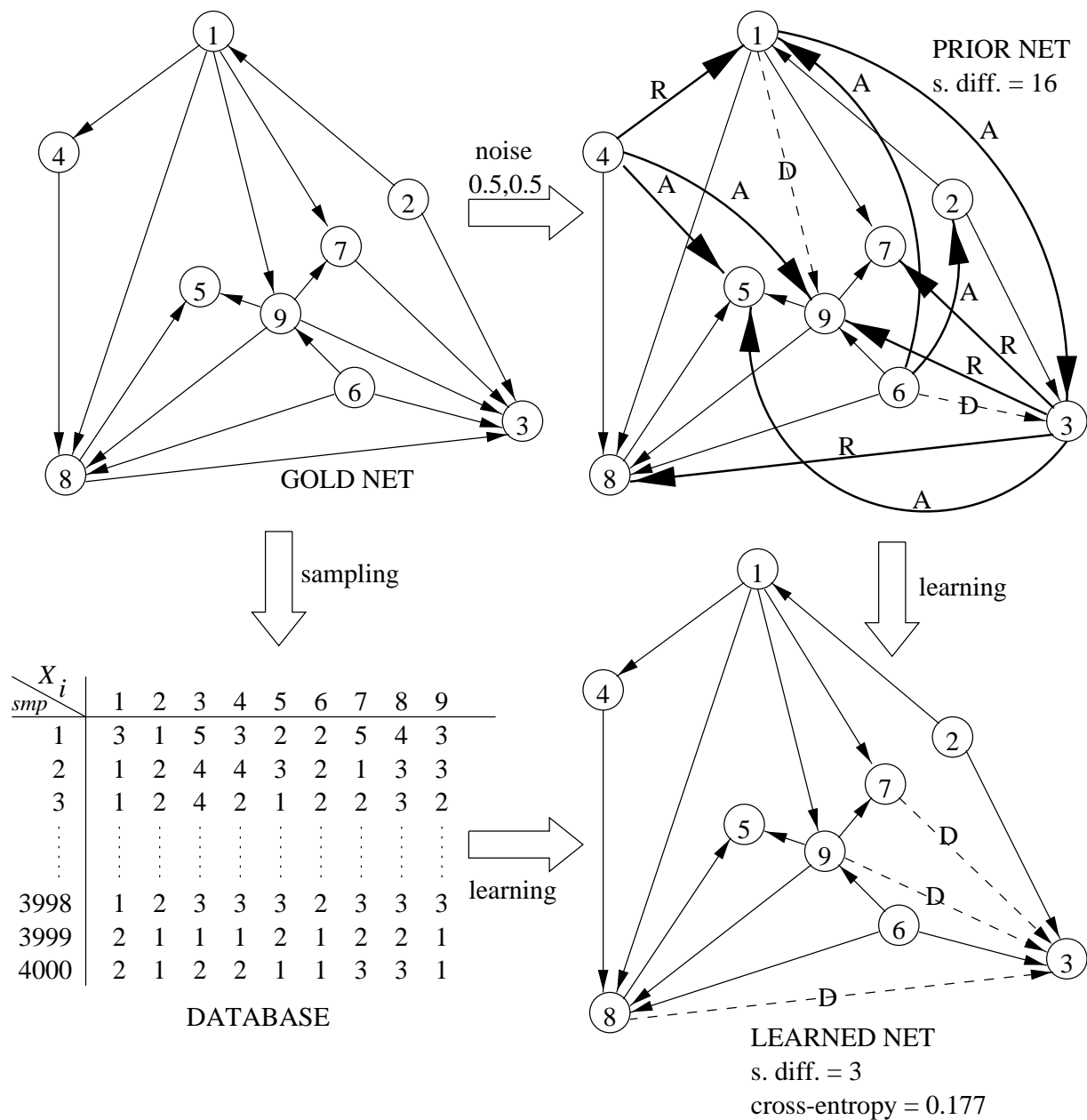


Figura 3: Risultati di un test su una rete “gold” generata a caso con 9 variabili e al più cinque stati per variabile. In un primo momento sono stati estratti 4000 campioni da questa rete, che poi è stata perturbata con probabilità $\frac{1}{2}$ ottenendo una rete “prior”. Le lettere associate agli archi indicano che un arco è stato aggiunto(A), rimosso(D) o invertito(R). A partire dal “prior” utilizzando i dati contenuti nel database è stata eseguita la procedura di apprendimento a due passi, con $\alpha = 100$, che ha dato come risultato la rete “learned”. Si osservi come la differenza strutturale dalla “gold” sia scesa da 16 a 3 con l’apprendimento, mentre le tabelle di probabilità della “learned” hanno una bassa cross-entropia, 0.176679, con quelle della “gold”.

+ net initialize ++++++

GOLD NETWORK:
Nodes : 20
- edeges list -----
1 -> 9
1 -> 12
2 -> 6
2 -> 13
3 -> 17
3 -> 18
5 -> 16
6 -> 5
6 -> 13
7 -> 19
8 -> 2
8 -> 10
8 -> 11
8 -> 12
8 -> 15
9 -> 2
9 -> 5
9 -> 15
10 -> 5
10 -> 20
11 -> 3
11 -> 4
11 -> 18
12 -> 4
12 -> 9
12 -> 13
13 -> 18
14 -> 13
15 -> 4
16 -> 14
16 -> 15
17 -> 2
17 -> 9
17 -> 16
17 -> 18
17 -> 19
18 -> 4
19 -> 15
20 -> 1
20 -> 3
20 -> 15

PRIOR NETWORK:
Nodes : 20
- edeges list -----
1 -> 9
1 -> 12
2 -> 6
2 -> 12 (A)
2 -> 13
3 -> 18
3 -> 20 (R)
4 -> 15 (R)
4 -> 18 (R)
5 -> 4 (A)
5 -> 16
6 -> 5
6 -> 13
7 -> 3 (A)
7 -> 14 (A)
7 -> 19
8 -> 2
8 -> 6 (A)
8 -> 11
8 -> 12
8 -> 15
9 -> 2
9 -> 5
9 -> 6 (A)
9 -> 12 (R)
9 -> 13 (A)
9 -> 15
10 -> 2 (A)
10 -> 3 (A)
10 -> 5
10 -> 14 (A)
10 -> 17 (A)
10 -> 20
11 -> 3
11 -> 4
11 -> 16 (A)
11 -> 18
12 -> 4
12 -> 6 (A)
12 -> 13
13 -> 18
14 -> 1 (A)
14 -> 6 (A)
14 -> 13
14 -> 16 (R)
16 -> 4 (A)
16 -> 15
17 -> 2
17 -> 9
17 -> 16
17 -> 18
17 -> 19
18 -> 15 (A)
19 -> 3 (A)
19 -> 6 (A)
19 -> 14 (A)
19 -> 15
19 -> 20 (A)
20 -> 1
20 -> 4 (A)
20 -> 13 (A)
20 -> 15

PRIOR NETWORK vs GOLD NETWORK
structural-difference measure: 35

+ learning ++++++
- structure -

1^ s.i.: score = -25087.460449 + 119.821777 (add: 8 -> 10)
2^ s.i.: score = -24967.638672 + 68.333008 (del: 18 -X 15)
3^ s.i.: score = -24899.305664 + 48.597778 (add: 17 -> 3) <E> (R)
4^ s.i.: score = -24850.707886 + 44.805664 (del: 9 -X 13)
5^ s.i.: score = -24805.902222 + 44.082397 (del: 2 -X 6) <E> (A)
6^ s.i.: score = -24761.819824 + 34.018677 (del: 4 -X 15) <E> (R)
7^ s.i.: score = -24727.801147 + 22.134277 (del: 5 -X 4)
8^ s.i.: score = -24705.666870 + 19.404419 (del: 19 -X 6)
9^ s.i.: score = -24686.262451 + 19.293823 (del: 20 -X 13)
10^ s.i.: score = -24666.968628 + 18.766235 (del: 10 -X 3)
11^ s.i.: score = -24648.202393 + 17.655396 (rev: 18 -> 4) <E> (R)
12^ s.i.: score = -24630.546997 + 20.442749 (del: 16 -X 4)
13^ s.i.: score = -24610.104248 + 24.396240 (add: 15 -> 4)

14^ s.i.: score = -24585.708008 + 16.250244 (del: 20 -X 4)
15^ s.i.: score = -24569.457764 + 10.084595 (del: 2 -X 12)
16^ s.i.: score = -24559.373169 + 17.817017 (rev: 12 -> 9)
17^ s.i.: score = -24541.556152 + 8.898560 (del: 11 -X 16)
18^ s.i.: score = -24532.657593 + 8.840942 (del: 8 -X 6)
19^ s.i.: score = -24523.816650 + 8.050171 (del: 7 -X 3)
20^ s.i.: score = -24515.766479 + 6.950684 (del: 10 -X 2)
21^ s.i.: score = -24508.815796 + 4.997314 (del: 10 -X 20) <E> (A)
22^ s.i.: score = -24503.818481 + 5.085205 (add: 11 -> 20)
23^ s.i.: score = -24498.733276 + 4.436768 (del: 7 -X 14)
24^ s.i.: score = -24494.296509 + 3.144043 (del: 19 -X 20)
25^ s.i.: score = -24491.152466 + 2.805298 (del: 19 -X 3)
26^ s.i.: score = -24488.347168 + 4.223511 (rev: 3 -> 17) <C>
27^ s.i.: score = -24484.123657 + 2.608276 (del: 14 -X 6)
28^ s.i.: score = -24481.515381 + 2.546143 (del: 9 -X 6)
29^ s.i.: score = -24478.969238 + 2.349854 (del: 14 -X 1)
30^ s.i.: score = -24476.619385 + 5.340576 (rev: 16 -> 14)
31^ s.i.: score = -24471.278809 + 2.826294 (del: 19 -X 14)
32^ s.i.: score = -24468.452515 + 1.565552 (add: 19 -> 16) <E> (D)
33^ s.i.: score = -24466.886963 + 1.408447 (del: 10 -X 17)
34^ s.i.: score = -24465.478516 + 1.064941 (del: 12 -X 6)
35^ s.i.: score = -24464.413574 + 0.884766 (add: 2 -> 6)
36^ s.i.: score = -24463.528809 + 0.639893 (add: 10 -> 6) <E> (D)
37^ s.i.: score = -24462.888916 + 0.584473 (add: 12 -> 7) <E> (D)
38^ s.i.: score = -24462.304443 + 0.333252 (add: 10 -> 7) <E> (D)
39^ s.i.: score = -24461.971191 No better changes.

- prob. tables -
Learning prob. tables for node 1
(...)
Learning prob. tables for node 20

+ results ++++++

LEARNED NETWORK:
Nodes : 20

- edeges list -----

1 -> 9
1 -> 12
2 -> 6
2 -> 13
3 -> 17
3 -> 18
3 -> 20 (R)
5 -> 16
6 -> 5
6 -> 13
7 -> 19
8 -> 2
8 -> 10 (A)
8 -> 11
8 -> 12
8 -> 15
9 -> 2
9 -> 5
9 -> 15
10 -> 5
10 -> 6 (A)
10 -> 7 (A)
10 -> 14 (A)
10 -> 20 (D)
11 -> 3
11 -> 4
11 -> 18
11 -> 20 (A)
12 -> 4
12 -> 7 (A)
12 -> 9
12 -> 13
13 -> 18
14 -> 13
15 -> 4
16 -> 14
16 -> 15
17 -> 2
17 -> 9
17 -> 16
17 -> 18
17 -> 19
18 -> 4
19 -> 15
19 -> 16 (A)
20 -> 1
20 -> 15

LEARNED NETWORK vs GOLD NETWORK
structural-difference measure: 9
cross-entropy measure: 0.0524207

Un altro esempio, può essere osservato nell'output originale, fornito nella pagina precedente, modificato per mettere in risalto gli archi aggiunti (A), invertiti (R) o cancellati (D). La *gold network* ha 20 nodi e variabili binarie: è stata campionata ottenendo un database D di 2000 campioni. Perturbata con una probabilità di $\frac{1}{5}$, ha generato la *prior network*, annotata nell'output per mettere in luce le differenze rispetto alla rete originale. Il listato riporta la descrizione dei 39 passi dell'algoritmo di ricerca ($\alpha = 64$), che mostrano il valore attuale di $\text{SCORE}(\mathbf{m}_S)$ ed il guadagno ottenuto compiendo la variazione indicata fra parentesi. Di fianco è stata annotata la bontà della mossa (errata \mathbb{E} , correzione di errore precedente \mathbb{C}) e quale sarebbe stata la mossa esatta. L'output si conclude con la lista degli archi della rete appresa.

Si osservi come anche in questo caso la differenza strutturale scenda da 35 a 9 con l'apprendimento e la bssa cross-entropia fra la *gold network* e la *learned network*. Questo dato fa sospettare un possibile "overfitting" dei dati da parte dell'algoritmo di apprendimento delle tabelle di probabilità, in quanto, pur avendo a disposizione una struttura un po' differente dall'originale, questo riesce interpretare molto bene i dati, forse troppo, generando una rete di Bayes molto legata al database D . Interessante anche notare come l'algoritmo di apprendimento della struttura della rete rimedi solo una volta ad una scelta sbagliata compiuta in precedenza. A causa del suo carattere di ricerca locale, le scelte errate fatte ai primi passi fanno quasi certamente cadere in un massimo locale differente da quello globale, con poche probabilità che gli errori possano essere corretti. Comunque, anche in questo esempio la rete appresa è molto più vicina alla rete originale rispetto alla *prior network*, dimostrando come la tecnica implementata affini realmente la conoscenza iniziale grazie al contributo delle osservazioni contenute nel database D .

Per concludere l'analisi dell'algoritmo su reti simulate presentiamo l'apprendimento di una rete più semplice delle precedenti, illustrata in figura 4. In questo caso l'algoritmo ha appreso la rete senza commettere errori: il motivo è il corretto dimensionamento del numero di esempi rispetto alle dimensioni del modello e la corretta scelta del parametro α .

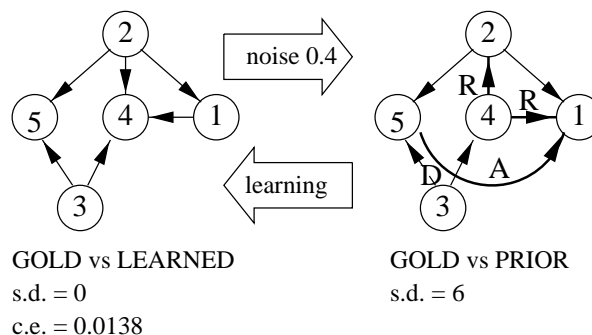


Figura 4: Risultati di un test su una rete "gold" generata a caso con 5 variabili e al più quattro stati per variabile. Sono stati estratti 5000 campioni da questa rete, poi è stata perturbata con probabilità $\frac{2}{5}$ ottenendo una rete "prior". Dal "prior", utilizzando i dati contenuti nel database, è stata eseguita la procedura di apprendimento a due passi, con $\alpha = 8$, che ha prodotto la rete "learned". La rete è stata correttamente riconosciuta sia nella struttura che nelle tabelle di probabilità (cross-entropia pari a 0.0138233). I passi compiuti dall'algoritmo di ricerca strutturale sono stati, partendo da $\text{SCORE}(\mathbf{m}_{\text{prior}}) = -26777.3$, add: $3 \rightarrow 5 (+310.8)$, rev: $2 \rightarrow 4 (+142.8)$, rev: $1 \rightarrow 4 (+179.5)$ e del: $5 - \times 1 (+24.0)$, ottenendo uno $\text{SCORE}(\mathbf{m}_{\text{learned}}) = -26120.2$.

7 Analisi dell'apprendimento da un database reale

Per applicare le tecniche esposte ad un processo proveniente dal mondo reale, abbiamo utilizzato il database riportato a pag.44 in [H-96]. Si tratta di uno studio svolto dai professori Sewell e Shah riguardo le intenzioni di una popolazione scolastica di 10416 studenti dell'ultimo anno di superiori di proseguire gli studi, frequentando l'università. I dati sono stati raccolti alla *Wisconsin High School* nel 1968 e riguardano il sesso dello studente, il suo stato socioeconomico, il suo quoziente di intelligenza, l'incoraggiamento ricevuto dai genitori a proseguire gli studi e l'effettiva decisione di farlo o meno. La seguente tabella illustra in dettaglio le variabili ed i possibili stati che possono assumere, nonché i codici numerici che il prototipo assegna loro.

X_i	sigla	significato	1	2	3	4
X_1	SEX	Sex	male	female		
X_2	SES	SocioEconomic Status	low	lower middle	upper middle	high
X_3	IQ	Intelligence Quotient	low	lower middle	upper middle	high
X_4	PE	Parental Encouragement	low	high		
X_5	CP	College Plans	yes	no		

Abbiamo tentato un primo apprendimento partendo da una rete senza archi fra i nodi, e tenendo il parametro α ad un basso valore, dato che la *prior network* fornita era del tutto non informativa. I risultati ottenuti sono riportati in forma grafica nella figura 5A. Come si può osservare, gli archi appresi codificano molte relazioni causali che ci saremmo potuti aspettare, come ad esempio il fatto che il fattore socioeconomico influenzi i genitori a dare o no ai figli stimoli per andare all'università. A nostro avviso, il solo arco che connette CP a IQ sembra illogico, in quanto supporrebbe il fatto che la scelta di andare all'università influisca sul quoziente di intelligenza dello studente.

Per tentare di evitare questo inconveniente abbiamo ripetuto l'apprendimento partendo da una rete in cui ogni nodo è collegato a CP con un arco. Questo equivale a supporre una conoscenza a priori sul fatto che tutti gli attributi possono influenzare in qualche modo la scelta universitaria. Avendo introdotto un *prior* che ritenevamo molto importante, l'apprendimento è stato eseguito con un parametro α elevato. I risultati ottenuti sono osservabili nella figura 5B. Il grafo appreso mostra che il problema riscontrato è stato eliminato (adesso c'è un arco da IQ a CP, che codifica il fatto che il quoziente di intelligenza può influenzare la scelta di proseguire gli studi). Però, la rete è logicamente debole rispetto ad un altro arco, quello che collega SES a IQ. In questo caso infatti sembrerebbe il quoziente di intelligenza poter essere influenzato dalla condizione socioeconomica dell'individuo.

Abbiamo tentato di percorrere una terza strada. Siamo ritornati ad una *prior network* priva di archi, ma abbiamo imposto dei vincoli sulla struttura. In questo caso si è supposto che il nodo CP non potesse esser padre di altri nodi. I risultati ottenuti dall'apprendimento in queste condizioni sono riportati in figura 5C, partendo da un parametro α basso. La rete appresa è la stessa del caso precedente.

Da questi risultati si può trarre la conclusione che l'algoritmo proposto sia un buon approccio per trattare dati in una prima fase, in modo da ottenere una rete di Bayes che illustri buona parte delle relazioni fra le variabili, commettendo qualche errore. In una seconda fase, partendo dai risultati ottenuti, è possibile utilizzare qualche metodo più sofisticato per tentare di raffinare la "bontà" causale della rete. Un esempio di questa tecnica può essere trovato in [H-96], in

cui l'impiego di una metodologia che utilizza variabili nascoste permette di superare i problemi riscontrati, introducendo una nuova variabile che sembra render conto della "qualità" dei genitori.

Per concludere, ecco l'output del prototipo nel primo caso esaminato. Si osservi come l'algoritmo aggiunga solamente archi, in quanto avviato su una rete vuota.

```
+ reading database ++++++
College Plans
Sewall & Shah, (c) 1968 The University of Chicago.

+ learning ++++++
- structure -
1^ search iteration: current score = -49448.162109 + 1697.895508 (add: 4 -> 5 )
2^ search iteration: current score = -47750.266602 + 1009.042969 (add: 2 -> 4 )
3^ search iteration: current score = -46741.223633 + 768.684570 (add: 5 -> 3 )
4^ search iteration: current score = -45972.539062 + 218.387695 (add: 2 -> 5 )
5^ search iteration: current score = -45754.151367 + 97.516602 (add: 4 -> 3 )
6^ search iteration: current score = -45656.634766 + 77.052734 (add: 1 -> 4 )
7^ search iteration: current score = -45579.582031 No better changes.
- prob. tables -
Learning prob. tables for node 1
Learning prob. tables for node 2
Learning prob. tables for node 3
Learning prob. tables for node 4
Learning prob. tables for node 5

+ results ++++++

LEARNED NETWORK:
# Nodes : 5

- edeges list -----
1 -> 4
2 -> 4
2 -> 5
4 -> 3
4 -> 5
5 -> 3

- nodes tables -----
*Node 1 with 2 states
1      2
-----
0.483730 0.516270

*Node 2 with 4 states
1      2      3      4
-----
0.240364 0.256537 0.256246 0.246853

*Node 3 with 4 states & 4 parents config.
4 5 | 1      2      3      4
-----
1 1 | 0.151274 0.221338 0.300955 0.326433
2 1 | 0.077136 0.176288 0.295662 0.450913
1 2 | 0.375403 0.297852 0.209774 0.116971
2 2 | 0.231995 0.291794 0.267350 0.208861

*Node 4 with 2 states & 8 parents config.
1 2 | 1      2
-----
1 1 | 0.681581 0.318419
2 1 | 0.833584 0.166416
1 2 | 0.512317 0.487683
2 2 | 0.633704 0.366296
1 3 | 0.352194 0.647806
2 3 | 0.492576 0.507424
1 4 | 0.144061 0.855939
2 4 | 0.194551 0.805449

*Node 5 with 2 states & 8 parents config.
2 4 | 1      2
-----
1 1 | 0.035111 0.964889
2 1 | 0.062788 0.937212
3 1 | 0.079839 0.920161
4 1 | 0.144342 0.855658
1 2 | 0.347789 0.652211
2 2 | 0.455231 0.544770
3 2 | 0.532131 0.467869
4 2 | 0.726134 0.273866
```

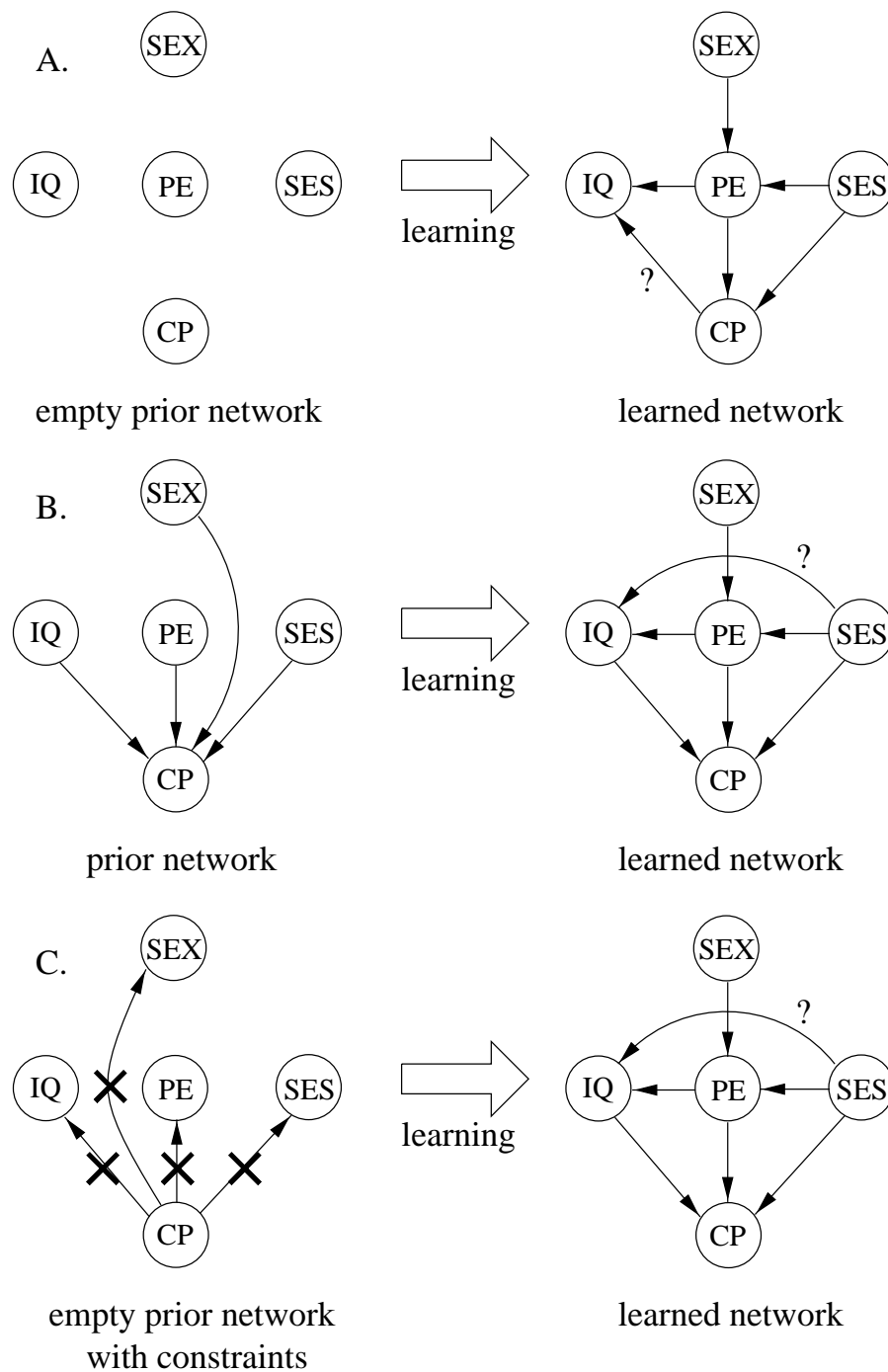


Figura 5: Risultati dell'apprendimento dal database di Sewell & Shah. **A.** $\alpha = 8$ e *prior network* vuota. **B.** $\alpha = 256$ e *prior network* con un arco entrante in CP da ogni nodo. **C.** $\alpha = 8$ e *prior network* vuota, ma con il vincolo che CP non può essere padre di alcun nodo. Gli archi con un "?" sono, a nostro avviso, illogici.

8 Osservazioni conclusive

In questa relazione è stata presentata una tecnica per apprendere reti bayesiane su un insieme di variabili aleatorie a partire da un database di loro realizzazioni. La tecnica esposta può essere utilizzata per comprendere le relazioni di causalità che intercorrono fra le variabili e per attuare una prima analisi dei dati ad esse relativi.

Sono stati messi in luce alcuni difetti, che abbiamo imputato sia alla non applicabilità delle ipotesi a molti casi di interesse reale che alla difficoltà di una corretta assegnazione dei parametri dell'algoritmo, primo fra tutti il coefficiente α . Inoltre è stato mostrato che nel caso in cui le ipotesi siano soddisfatte, i parametri correttamente dimensionati e si stia utilizzando un numero sufficiente di campioni, l'algoritmo apprende senza errori la rete.

In ultima analisi, l'apprendimento basato su selezione del modello e stima bayesiana dei parametri è un buon metodo per affrontare il problema di trovare relazioni fra dati, almeno in una prima approssimazione. Un ulteriore raffinamento può essere raggiunto a partire dai risultati ottenuti con le metodologie mostrate, impiegando tecniche di apprendimento più potenti, quali sono quelle che suppongono la presenza di variabili nascoste.

Questo lavoro è stato realizzato per sostenere l'esame di "intelligenza artificiale" tenuto dal Prof. G.Soda con la collaborazione del Dott. P.Frasconi. Sono state utilizzate le strutture didattiche del laboratorio dell'informazione "ex-forno", all'interno della facoltà di ingegneria dell'università di Firenze.

Firenze, 9 marzo 1999

Riferimenti bibliografici

- [SL-90] Spiegelhalter & Lauritzen, *Sequential updating of conditional probabilities on directed graphical structures*, Networks, 20:579-605, 1990 4
- [CH-92] Cooper & Herskovits, *A Bayesian method for the induction of probabilistic networks from data*, Machine Learning, 9:309-347, 1992 4
- [HGC-95] Heckerman, Geiger & Chickering, *Learning Bayesian networks: The combination of knowledge and statistical data*, Machine Learning, 20:197-243, 1995 5, 6
- [H-96] Heckerman, *A tutorial on learning with Bayesian Networks*, Microsoft Technical Report, TR-95-06, revised Nov. 1996 11, 11

A Implementazione

L'implementazione degli algoritmi per l'apprendimento di reti bayesiane partendo da database di esempi è stata fatta utilizzando il linguaggio C++ in ambiente Linux. Qui di seguito vengono illustrate le scelte compiute e alcuni degli algoritmi di supporto utilizzati, nonché come sono state realizzate le equazioni 13 e 12.

A.1 Datatypes di base

Abbiamo implementato quattro template di classi per realizzare vettori, matrici, code circolari e grafi diretti aciclici. Sui primi tre tipi di dati c'è poco da dire: sono classi abbastanza standard che mettono a disposizione oggetti vettori, matrici e code circolari (realizzate tramite vettori). La loro allocazione è dinamica e la loro dimensione può essere cambiata all'interno del programma, ma deve essere specificata prima di inserirvi nuovi dati.

Più interessante è la classe `d_a_graph` che implementa i grafi aciclici orientati. Premettiamo che i nodi sono rappresentati con degli interi e gli archi con coppie di nodi, quindi coppie di interi. All'interno della classe sono contenuti due vettori, uno di nodi di dimensione N ed uno di archi di dimensione $N(N-1)/2$, e un intero N . Il vettore di nodi è composto da elementi impostabili dall'utente, caratteristica utilizzata nella stesura della classe `bayes_net` per memorizzare le tabelle di probabilità associate a ciascun nodo. Il vettore di archi ha componenti intere e tiene traccia degli archi $i \rightarrow j$ tali che $j > i$. La funzione che mappa gli archi in tale vettore è la seguente.

$$\text{pos}(i \rightarrow j) = \begin{cases} j - i & \text{per } i = 1 \\ j - i + (N - 1)(i - 1) - \frac{(i-1)(i-2)}{2} & \text{per } i > 1 \end{cases} \quad (17)$$

Nel vettore un arco $i \rightarrow j$ è rappresentato da un 1 se è presente, da un -1 se è presente con il verso invertito e da uno 0 se non è presente né $i \rightarrow j$ né $j \rightarrow i$. Per rendere trasparente all'utente il meccanismo di accesso agli archi così rappresentati è stato creato un metodo `edges_manager` che permette ai metodi interfaccia di compiere le operazioni di inserimento, cancellazione e accesso agli archi in modo intuitivo.

Abbiamo realizzato un metodo `nPa(node v)` che conta il numero di padri del nodo v e sono state create delle macro "iteratori" per scandire tutti gli elementi presenti nella struttura. In particolare, `forall_nodes(v, G)` assegna in sequenza i valori dei nodi di G a v e `forall_edges(e, G)` i valori degli archi di G a e . Ci sono anche degli iteratori per gli archi mancanti nella struttura della rete, per i nodi padri e per quelli figli di un certo nodo.

Per controllare l'aciclicità di un grafo si è sfruttato l'algoritmo citato nei sorgenti, che attua un ordinamento topologico del grafo in modo da attraversare per primi i nodi sorgenti, poi i loro figli e così via. Se l'algoritmo riesce ad ordinare in questo modo tutti i nodi, il grafo è aciclico. Ci siamo basati su questa procedura per realizzare un metodo per l'inserimento di archi nella struttura, senza violare la condizione di aciclicità.

```
/*
 * Vector.h
 * vector datatype
 * (c) 1997 E.Papalini & M.Piccini
 */
#ifdef _Vector_DataType
#define _Vector_DataType
#include <iostream.h>
```

```

template <class ItemType>
class vector {
private:
    ItemType *V;
    int N; // vector dimension

public:
    /* constructor */
    vector() { N = 0; }

    /* copy constructor */
    vector(const vector &Obj) {
        if (Obj.N != 0) {
            if (!V = new ItemType [Obj.N]) {
                cout << "Error: Out of Memory.\n";
                exit(1);
            }
            N = Obj.N;
            for (int k = 1; k <= N; k++)
                *(V+(k-1)) = *(Obj.V+(k-1));
        } else {
            cout << "Error: Object not initialized.\n";
            exit(1);
        }
    }

    /* initializer: set the vector dimension */
    int init(int n) {
        if (n > 0) {
            if (N != 0) delete V;
            if (!V = new ItemType [n]) {
                cout << "Error: Out of Memory.\n";
                exit(1);
            }
            N = n;
            return 0;
        }
        cout << "Error: Object size must be positive.\n";
        exit(3);
    }

    /* get vector dimension */
    int getN() { return N; }

    /* set all elements to "item" */
    int set(ItemType item) {
        if (N != 0) {
            for (int k = 1; k <= N; k++)
                *(V+(k-1)) = item;
            return 0;
        }
        cout << "Error: Object not initialized.\n";
        exit(1);
    }

    /* get the i-th element */
    ItemType& operator()(int i) {
        if (N != 0) {
            if ((i > 0) && (i <= N))
                return *(V+(i-1));
            cout << "Error: index out of range.\n";
            exit(1);
        }
        cout << "Error: Object not initialized.\n";
        exit(1);
    }

    /* copy operator */
    vector operator=(const vector &Obj) {
        // precondition: ItemType must have a valid = operator
        if (N != 0) {
            if (Obj.N == N) {
                for (int k = 1; k <= N; k++)
                    *(V+(k-1)) = *(Obj.V+(k-1));
                return *this;
            }
            cout << "Error: Objects with different sizes.\n";
            exit(1);
        }
        cout << "Error: Object not initialized.\n";
        exit(1);
    }
};

#endif // _Vector_DataType

/*
 * Matrix.h
 * matrix datatype
 * (c) 1997 E.Papalini & M.Piccini
 */

#endif _Matrix_Data_Type

```

```

#define _Matrix_Data_Type

#include <iostream.h>

template <class ItemType>
class Matrix {
private:
    ItemType *T;
    int N; // matrix first dimension
    int M; // matrix second dimension
public:

    /* constructor */
    Matrix() { N = M = 0; }

    /* copy constructor */
    Matrix(const Matrix &Obj) {
        if (Obj.N != 0 && Obj.M != 0) {
            if (!(T = new ItemType [Obj.N*Obj.M])) {
                cout << "Error: Out of Memory.\n";
                exit(1);
            }
            N = Obj.N; M = Obj.M;
            for (int j = 1; j <= M; j++)
                for (int i = 1; i <= N; i++)
                    *(T+((i-1)*M+j-1)) = *(Obj.T+((i-1)*Obj.M+j-1));
        } else {
            cout << "Error: It is impossible to copy a NULL object.\n";
            exit(1);
        }
    }

    /* get first matrix dimension */
    int getN() { return N; }

    /* get second matrix dimension */
    int getM() { return M; }

    /* set all elements to "item" */
    int set(ItemType item) {
        if (N != 0 && M != 0) {
            for (int j = 1; j <= M; j++)
                for (int i = 1; i <= N; i++)
                    *(T+((i-1)*M+j-1)) = item;
            return 0;
        }
        cout << "Error: I can't access an uninitialized Matrix.\n";
        exit(1);
    }

    /* initialize matrix dimensions */
    int init(int n,int m) {
        if (n>0 && m>0) {
            if (N != 0 && M != 0) delete T;
            if (!(T = new ItemType [n*m])) {
                cout << "Error: Out of Memory.\n";
                return 1;
            }
            N = n; M = m;
            return 0;
        }
        cout << "Error: Matrix size must be positive.\n";
        exit(1);
    }

    /* get (i,j) element */
    ItemType& operator()(int i, int j) {
        if (N != 0 && M != 0) {
            if ((i>0) && (j>0) && (i <= N) && (j <= M))
                return *(T+((i-1)*M+j-1));
            cout << "Error: ( " << i << ", " << j << " ) Out of Range.\n";
            exit(1);
        }
        cout << "Error: I can't access an uninitialized Matrix.\n";
        exit(1);
    }
};

#endif // _Matrix_Data_Types

/*
 * C_Queue.h
 * circular queue datatype
 * (c) 1997 E.Papalini & M.Piccini
 */

#ifndef _Circular_Queue_DataType
#define _Circular_Queue_DataType

#include <iostream.h>
#include "../Vector.h"

template <class ItemType>

```

```

class c_queue {
private:
    vector<ItemType> CQ;
    int first;
    int size;

public:
    /* constructor */
    c_queue() { first = 1; size = 0; }

    /* destructor */
    ~c_queue() { /**/ }

    /* initialize an empty queue */
    int init() { first = 1; size = 0; return 0; }

    /* initialize a queue of n elements */
    int init(int n) { first = 1; size = 0; CQ.init(n); return 0; }

    /* is queue empty ? */
    int empty() { return (size == 0); }

    /* is queue full ? */
    int full() { return (size == CQ.getN()); }

    /* pop current element from the queue */
    ItemType pop() {
        if (!empty()) {
            int tmp = first;
            first = (first % (CQ.getN())) + 1;
            size--;
            return CQ(tmp);
        }
        cout << "Error: Empty object.\n";
        exit(6);
    }

    /* push element "item" in the queue */
    void push(ItemType item) {
        if (!full()) {
            CQ(((first + size - 1) % (CQ.getN())) + 1) = item;
            size++;
            return;
        }
        cout << "Error: Full object.\n";
        exit(7);
    }
};

```

```

#endif // _Circular_Queue_DataType

```

```

/*
 * D_A_Graph.h
 * direct acyclic graph datatype
 * (c) 1997 E.Papalini & M.Piccini
 *
 */

```

```

#ifndef _Direct_Acyclic_Graph_DataType
#define _Direct_Acyclic_Graph_DataType

```

```

#include <iostream.h>
#include "../Vector.h"
#include "../C_Queue.h"

```

```

typedef int node; // node type

```

```

struct edge { node s; node t; }; // edge type

```

```

// iterators

```

```

#define forall_nodes(v,G)
    for((v) = 1; (v) <= ((G).getN()); (v)++)

```

```

#define forall_edges(e,G)
    for((e.s) = 1; (e.s) <= ((G).getN()); (e.s)++) \
        for ((e.t) = 1; (e.t) <= ((G).getN()); (e.t)++) \
            if (((G).get_edge((e))) != 1); else

```

```

#define forall_missing_edges(e,G) \
    for((e.s) = 1; (e.s) <= (((G).getN())-1); (e.s)++) \
        for ((e.t) = ((e.s)+1); (e.t) <= ((G).getN()); (e.t)++) \
            if (((G).get_edge((e))) != 0); else

```

```

#define forall_parents(Pa,v,G) \
    for((Pa) = 1; (Pa) <= ((G).getN()); (Pa)++) \
        if (((G).get_edge((Pa),(v))) != 1); else

```

```

#define forall_children(Ch,v,G) \
    for((Ch) = 1; (Ch) <= ((G).getN()); (Ch)++) \
        if (((G).get_edge((v),(Ch))) != 1); else

```

```

template <class NodeType>
class d_a_graph {
private:
    vector<NodeType> Nodes; // vector of nodes
    vector<int> Edges; // vector of edges
    int N; // number of nodes

    /* get the position of edge i -> j in edge vector */
    int pos(node i, node j) {
        // precondition: i < j && N > 1
        int res = j-i;
        if (i > 1) res += (N-1)*(i-1) - ((i-1)*(i-2))/2;
        return res;
    }

    /* manager of edge insertion, deletion & getting */
    int edges_manager(node i, node j, int op) {
        int c = 0;
        if (N > 1) {
            if (i > 0) && (j > 0) && (i <= N) && (j <= N) {
                if (i == j) return 0;
                if (i < j) c = 1;
                if (i > j) { c = i; i = j; j = c; c = -1; }
                switch (op) {
                    case 0: // delete
                        Edges[pos(i,j)] = 0;
                        return 0; break;
                    case 1: // insert
                        Edges[pos(i,j)] = c;
                        return c; break;
                    case 2: // get
                        return Edges[pos(i,j)]*c; break;
                    default:
                        cout << "Error: Option unknown.\n";
                        exit(7);
                }
            }
            cout << "Error: index out of range.\n";
            exit(1);
        }
        cout << "Error: Object uninitialized.\n";
        exit(1);
    }

public:
    /* constructor */
    d_a_graph() { N = 0; }

    /* initialize nodes & edges vectors */
    int init(int n) {
        if (n > 1) {
            Nodes.init(n);
            Edges.init((n*(n-1))/2);
            Edges.set(0);
            N = n;
            return 0;
        }
        cout << "Error: Nodes number must be > 1.\n";
        exit(1);
    }

    /* get parents number of node i */
    int nPa(node i) {
        int nPai = 0;
        // precondition: N > 1
        for (int aux = 1; aux <= N; aux++)
            if (get_edge(aux,i) == 1) nPai++;
        return nPai;
    }

    /* get nodes number */
    int getN() { return N; }

    /* get i-th node */
    NodeType& operator() (node i) {
        // precondition: N > 1
        return Nodes(i);
    }

    /* delete edge */
    int del_edge(node i, node j) { return edges_manager(i,j,0); }
    int del_edge(edge e) { return edges_manager(e.s,e.t,0); }

    /* insert edge */
    int set_edge(node i, node j) { return edges_manager(i,j,1); }
    int set_edge(edge e) { return edges_manager(e.s,e.t,1); }

    /* get edge */
    int get_edge(node i, node j) { return edges_manager(i,j,2); }
    int get_edge(edge e) { return edges_manager(e.s,e.t,2); }

    /* is graph acyclic ? */
    int acyclic() {
        // based upon
    }

```

```

// A.B.Kahn, Topological Sorting of Large Networks,
// Communications of the ACM, vol.5, 558-562, 1962
vector<int> nParents;
c.queue<int> orphans;

int n = 0;
node v, w;
// precondition: N > 1
nParents.init(N);
orphans.init(N);

forall_nodes(v,* this)
  if ((nParents(v) = nPa(v)) == 0) orphans.push(v);
while (!orphans.empty()) {
  v = orphans.pop();
  n++;
  forall_children(w,v,* this)
    if (--nParents(w) == 0) orphans.push(w);
}

return (n == N);
}

/* insert edge only if graph remains acyclic */
int acyclic_set_edge (node i, node j) {
  // precondition: N > 1
  int c = get_edge(i,j);
  set_edge(i,j);
  if (acyclic())
    return 1;
  else {
    if (c == 0) del_edge(i,j);
    if (c == -1) set_edge(j,i);
    return 0;
  }
}

int acyclic_set_edge (edge e) {
  return acyclic_set_edge (e.s,e.t);
}
};

#endif // _Direct_Acyclic_Graph_DataType

```

A.2 Classi per rappresentare i nodi

Abbiamo creato una classe base `node_base` contenente l'identificatore del nodo e il numero degli stati della variabile aleatoria associata. Da questa classe è stata derivata una classe `node_type` contenente la tabella di probabilità condizionata associata al nodo. Questa tabella è implementata come una matrice di float di dimensioni pari a $r_i \times q_i$, ove X_i è la variabile a cui è associata. È anche stata derivata dalla classe base la classe `node_info` che invece contiene lo stato corrente della variabile aleatoria.

Questa ultima classe è necessaria per implementare la classe `node_parents` che contiene l'elenco dei nodi padri di un certo nodo: `node_parents` viene utilizzata nel calcolo di formule come la 13, in cui c'è bisogno di un iteratore che assegni in successione ad una variabile tutti gli stati che un insieme di nodi (\mathbf{Pa}_i in questo caso) può assumere.

All'unico scopo di calcolare la cross-entropia fra due reti di Bayes è stata derivata una classe `node_more_info` dalla classe `node_info`, che contiene due vettori di nodi rappresentanti le liste di padri del nodo corrente in due differenti reti. Questa classe è inclusa in `all_nodes`: qui c'è la lista di tutte le variabili con associate quelle dei padri nelle due reti, utilizzata per implementare la formula 16, che necessita di un iteratore su tutti i nodi che aggiorni le configurazioni dei padri ogni volta.

In figura 6 è rappresentato il diagramma delle classi. La relazione “is referred by” si riferisce all'inclusione di una classe da parte di un'altra tramite un puntatore, ma nel nostro progetto questo è mascherato dall'utilizzo dei templates.

È necessario osservare come le classi `node_parents` e `all_nodes` abbiano delle macro “iteratori” che, tramite la ridefinizione dell'operatore ++ sulle rispettive classi, permettono di assegnare

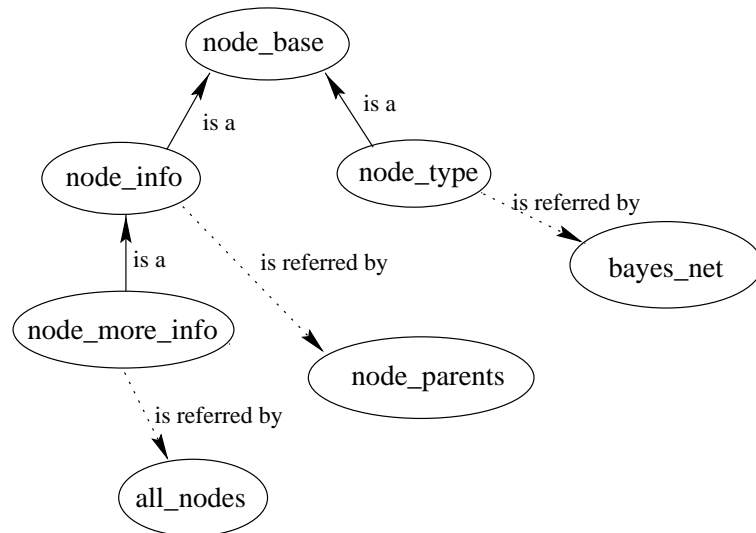


Figura 6: Albero parziale delle classi del progetto.

in successione ad una variabile tutte le possibili configurazioni assunte dai nodi padri oppure da tutti i nodi del grafo.

```

/*
 * Nodes.h
 * classes of nodes
 * (c) 1997 E.Papalini & M.Piccini
 *
 */

#ifndef _Nodes_Types
#define _Nodes_Types

#include <iostream.h>
#include "../Vector.h"
#include "../Matrix.h"
#include "../DAGraph.h"

/* node_base */

#define forall_node_states(k,xi) \
    for((k) = 1; (k) <= ((xi).ri); (k)++)

class node_base {
public:
    node i; // # of node
    int ri; // # max of states

    node_base();
    node_base(int j, int rj = 1);
};

/* node_type */

class node_type : public node_base {
public:
    Matrix<float> P; // P(j,k) j = 1 .. Qi; k = 1 .. ri;
};

/* node_info */

class node_info : public node_base {
public:
    int s; // current state

    node_info();
    node_info(int j, int rj = 1, int inits = 1);

    void init_state();
    void last_state();

    node_info operator++(); //prefix
};

/* node_more_info */

```

```

class node_more_info : public node_info {
public:
    vector<node> P;
    vector<node> Q;

    node_more_info();
    node_more_info(int j, int rj = 1, int inits = 1, int nP = 1, int nQ = 1);

    int init(node v, d_a_graph<node_type> &DAG_P,
            d_a_graph<node_type> &DAG_Q);
};

/* node_parents */
#define forall_parents_states(j,Pai) \
    (Pai).init_state(); \
    for((j) = 1; (j) <= ((Pai).getQ()); ((j)++), (++(Pai)))

#define view_parents_state(state,Pai) \
    for((state) = 1; (state) <= ((Pai).getN()); (state)++)

class node_parents {
private:
    vector<node_info> P;
    int Q;

public:
    node_parents(node i, d_a_graph<node_type> &G);

    int getQ();
    int getN();
    void init_state();
    void last_state();

    void operator++(); // prefix
    node_info& operator() (int i);

    node_parents setj(int j);
    int getj();
};

/* all_nodes */
#define forall_nodes_states(ki,Nodes) \
    (Nodes).init_state(); \
    for((ki) = 1; (ki) <= ((Nodes).getQ()); ((ki)++), (++(Nodes)))

class all_nodes {
private:
    vector<node_more_info> N;
    int Q;

public:
    all_nodes(d_a_graph<node_type> &DAG_P,
            d_a_graph<node_type> &DAG_Q);

    void init_state();
    void operator++(); // prefix
    node_more_info& operator() (int i);

    int getQ();
    int getj_P(node v);
    int getj_Q(node v);
};

#endif // _Nodes.Types

/*
 * Nodes.cc
 * methods for classes of nodes
 * (c) 1997 E.Papalini & M.Piccini
 */

#include "../Nodes.h"

//
// class node_base
//
// constructors */
node_base::node_base() { i = 0; ri = 1; }
node_base::node_base(int j, int rj = 1) { i = j; ri = rj; }

//
// class node_info : public node_base
//
// constructors */
node_info::node_info() : node_base() { s = 0; }
node_info::node_info(int j, int rj = 1, int inits = 1) : node_base(j,rj) { s = inits; }

/* set initial state */
void node_info::init_state() { s = 1; }

```

```

/* set final state */
void node_info::last_state() { s = ri;}

/* next state */
node_info node_info::operator++() {
    if (++s > ri) s = 1;
    return *this;
}

//
// class node_more_info : public node_info
// only used for calculate cross-entropy between two bayesian networks
//
/* constructors */
node_more_info::node_more_info() : node_info() { P.init(1); Q.init(1); }
node_more_info::node_more_info(int j, int rj = 1, int inits = 1, int nP = 1, int nQ = 1)
    : node_info(j,rj,inits) { P.init(nP); Q.init(nQ); }

/* initialize */
int node_more_info::init(node v, d_a_graph<node_type> &DAG_P, d_a_graph<node_type> & DAG_Q) {
    node w;
    int k;

    i = DAG_P(v).i;
    ri = DAG_P(v).ri;
    s = 1;

    if (DAG_P.nPa(v) > 0) {
        P.init(DAG_P.nPa(v)); k = 0;
        forall_parents(w,v,DAG_P)
            P(++k)=w;
    } else {
        P.init(1);
        P(1) = 0;
    }

    if (DAG_Q.nPa(v) > 0) {
        Q.init(DAG_Q.nPa(v)); k = 0;
        forall_parents(w,v,DAG_Q)
            Q(++k)=w;
    } else {
        Q.init(1);
        Q(1) = 0;
    }
    return 0;
}

//
// class node_parents
//
/* constructors */
node_parents::node_parents(node i, d_a_graph<node_type> & G) {
    node v;
    int counter = 0;

    Q = 1;
    P.init(G.nPa(i));

    forall_parents(v,i,G) {
        counter++;
        P(counter).i = G(v).i;
        Q * = P(counter).ri = G(v).ri;
    }
    init_state();
}

/* get number of parents states */
int node_parents::getQ() { return Q; }

/* get number of parents */
int node_parents::getN() { return P.getN(); }

/* set initial parents state */
void node_parents::init_state() {
    for (int i = 1; i <= P.getN(); i++)
        P(i).s = 1;
}

/* set last parents state */
void node_parents::last_state() {
    for (int i = 1; i <= P.getN(); i++)
        P(i).s = P(i).ri;
}

/* next parents state */
void node_parents::operator++() {
    int aux = 1;
    do {
        if (++P(aux).s > P(aux).ri)
            P(aux++).s = 1;
        else
            return; // *this;
    } while(aux <= P.getN());
}

```

```

    return; // *this;
}

/* return i-th state */
node_info& node_parents::operator() (int i) { return P(i); }

/* set j-th parents state */
node_parents node_parents::setj(int j) {
    last_state();
    for (int counter = 1; counter <= j; counter++) {
        int aux = 1;
        do {
            if (++P(aux).s > P(aux).ri)
                P(aux++).s = 1;
            else
                break;
        } while(aux <= P.getN());
    }
    return *this;
}

/* get j-th parents state */
int node_parents::getj() {
    int tot = P(1).s;
    if (P.getN() > 0) {
        for (int i = 2; i <= P.getN(); i++) {
            int fact = 1;
            for (int j = 1; j <= (i-1); j++) {
                fact *= P(j).ri;
            }
            tot += fact*(P(i).s - 1);
        }
    }
    return tot;
}

//
// class all_nodes
// only used for calculate cross-entropy between two bayesian networks
//
// constructors */
all_nodes::all_nodes(d_a_graph<node_type> &DAG_P, d_a_graph<node_type> &DAG_Q) {
    node v;
    // precondition: DAG_P & DAG_Q must be based on the same node variables
    N.init(DAG_P.getN());
    Q = 1;
    forall_nodes(v,DAG_P) {
        N(v).init(v,DAG_P,DAG_Q);
        Q *= N(v).ri;
    }
}

/* set initial state of all nodes */
void all_nodes::init_state() {
    for(int i = 1; i <= N.getN(); i++)
        N(i).s = 1;
}

/* next state */
void all_nodes::operator++() {
    int aux = 1;
    do {
        if (++N(aux).s > N(aux).ri)
            N(aux++).s = 1;
        else
            return; // *this;
    } while(aux <= N.getN());
    return; // *this;
}

/* get i-th node */
node_more_info& all_nodes::operator() (int i) { return N(i); }

/* get number of states of all nodes */
int all_nodes::getQ() { return Q; }

/* get j-th state of parents of node v in graph Q */
int all_nodes::getj_Q(node v) {
    if(N(v).Q(1) > 0) {
        int tot = N(N(v).Q(1)).s;
        if (N(v).Q.getN() > 0) {
            for (int i = 2; i <= N(v).Q.getN(); i++) {
                int fact = 1;
                for (int j = 1; j <= (i-1); j++) {
                    fact *= N(N(v).Q(j)).ri;
                }
                tot += fact*(N(N(v).Q(i)).s - 1);
            }
        }
    }
    return tot;
}
return 1;
}
}

```

```

/* get j_th state of parents of node v in graph P */
int all_nodes::getj_P(node v) {
    if (N(v).P(1) > 0) {
        int tot = N(N(v).P(1)).s;
        if (N(v).P.getNO() > 0) {
            for (int i = 2; i <= N(v).P.getNO(); i++) {
                int fact = 1;
                for (int j = 1; j <= (i-1); j++) {
                    fact *= N(N(v).P(j)).ri;
                }
                tot += fact*(N(N(v).P(i)).s - 1);
            }
        }
        return tot;
    }
    return 1;
}

```

A.3 Classe per la gestione del database D

Il database di esempi è implementato tramite una classe che contiene una matrice di interi. Questa matrice ha come prima dimensione il numero dei campioni e come seconda il numero delle variabili associate alla rete. La classe mette a disposizione metodi per inserire un campione e fare delle semplici queries riguardo al numero dei campioni in cui $X_i = x_i^k$ oppure in cui $X_i = x_i^k$ e $\mathbf{Pa}_i = \mathbf{pa}_i^j$. Queste sono necessarie per il calcolo di N_{ijk} nelle equazioni 12 e 13.

Anche in questa classe è stato realizzato un iteratore per scandire l'intero database. Abbiamo scelto di dotare tutte le classi di opportuni iteratori per rendere più leggibile il codice. Osservando i sorgenti ci si renderà conto come alcuni algoritmi risultano di comprensione più immediata grazie all'uso degli iteratori, che sostituiscono oscuri cicli `for` altrimenti necessari per compiere le medesime operazioni.

A.4 Funzioni ausiliarie

Sono state implementate delle funzioni ausiliarie, una per generare numeri casuali ed un'altra per calcolare il logaritmo della funzione $\Gamma(x)$, utilizzato per il calcolo di $\text{SCORE}(\mathbf{m}_S)$ nell'equazione 13. Gli algoritmi utilizzati nei moduli `Math.cc` e `Random.cc` sono stati ispirati da quelli presenti nel testo citato nei sorgenti.

Abbiamo deciso di implementare una nostra versione del generatore di numeri pseudocasuali per poter avere delle condizioni di prova ripetibili. Ad esempio, l'uso del nostro generatore per perturbare i grafi permette, registrando opportunamente il valore dell'`idum`, di ripetere la *prior network* così generata in varie realizzazioni degli esperimenti.

```

/*
 * Database.h
 * database class
 * (c) 1997 E.Papalini & M.Piccini
 */

#ifndef _Database
#define _Database

#include <iostream.h>
#include "../Matrix.h"
#include "../Nodes.h"

#define forall_data(d,DB) \
    for((d) = 1; (d) <= ((DB).getNO()); (d)++)

class database {
private:
    Matrix<int> DB;

public:
    int init(int N, int n);
}

```

```

    int getN();
    int getn();

    int& operator() (int j, int k);

    int query(node_info& xi, node_parents& Pai);
    int query(node_info& xi);

    void insert(int j, vector<int>& datum);
    // void save();
    // void load();

    #ifdef _DEBUG_
        void debug_view();
    #endif // _DEBUG_
};

#endif // DataBase

/*
 * Database.h
 * methods for database class
 * (c) 1997 E.Papalini & M.Piccini
 *
 */

#include "../Database.h"

/* initialize database of N samples and n variables */
int database::init(int N, int n) { DB.init(N,n); return 0; }

/* get number of samples N */
int database::getN() { return DB.getN(); }

/* get number of variables n */
int database::getn() { return DB.getM(); }

/* get (i,j) element */
int& database::operator() (int j, int k) { return DB(j,k); }

/* query for number of samples where Xi = xi.s & Pai = Pai.s */
int database::query(node_info& xi, node_parents& Pai) {
    int l,s,rec = 0;
    forall_data(l,DB) {
        if (DB(l,xi.i) != xi.s) continue;
        view_parents_state(s,Pai)
        if (DB(l,Pai(s,i)) != Pai(s,s) { rec--; break; }
        rec++;
    }
    return rec;
}

/* query for number of samples where Xi = xi.s */
int database::query(node_info& xi) {
    int l,rec = 0;
    forall_data(l,DB)
        if (DB(l,xi.i) == xi.s) rec++;
    return rec;
}

/* insert j-th sample */
void database::insert(int j, vector<int>& datum) {
    if(datum.getN() == getn()) {
        for(int k = 1; k <= getn(); k++)
            DB(j,k) = datum(k);
        return ;
    }
    cout << "Error: sizes are not equal.\n";
    exit(9);
}

#ifdef _DEBUG_
/* view database */
void database::debug_view() {
    if (getN() > 0 && getn() > 0) {
        int j,k;
        cout << "- database ----- \n";
        forall_data(j,DB) {
            cout << "\ndatum " << j << " \t | ";
            for(k = 1; k <= getn(); k++)
                cout << DB(j,k) << " ";
        }
        cout << "\n";
    } else {
        cout << "Database is empty.\n";
    }
}
#endif // _DEBUG_

/*
 * Random.h
 * random number generator
 * (c) 1997 E.Papalini & M.Piccini
 *
 */

```

```

*/
#ifdef _Random_generator
#define _Random_generator

unsigned int rnd ();
unsigned int rnd (unsigned int N);
void randomize (unsigned int seed);
unsigned int get_idum();

#endif // _Random_generator

/*
 * Random.cc
 * random number generator
 * (c) 1997 E.Papalini & M.Piccini
 */

#include "../Random.h"

static unsigned int idum = 0;

/* random number generator */
unsigned int rnd () {
    return (idum = 1664525 * idum + 1013904223);
}
unsigned int rnd (unsigned int N) {
    return (rnd() % N);
}

/* randomizer */
void randomize (unsigned int seed) {
    idum = seed;
}

/* get current idum */
unsigned int get_idum() {
    return idum;
}

/*
 * Math.h
 * math functions
 * (c) 1997 E.Papalini & M.Piccini
 */

#ifdef _Math_Functions
#define _Math_Functions

#include <math.h>

float lngamma(float x);

#endif // _Math_Functions

/*
 * Math.cc
 * math functions
 * (c) 1997 E.Papalini & M.Piccini
 */

#include "../Math.h"

/* log(gamma(x)) */
float lngamma (float x) {
    // based upon
    // Numerical Recipes in C
    // alg. gammaln (6.1 Gamma,Beta and related functions 214)
    // 1992, Cambridge Press
    double a,b,c,d;
    static double coef[6] = { 76.18009172947146,
        -86.50532032941677,
        24.01409824083091,
        -1.231739572450155,
        0.1208650973866179e-2,
        -0.5395239384953e-5 };

    b = a + x;
    c = a + 5.5;
    c -= (a + 0.5) * log(c);
    d = 1.000000000190015;
    for (int i = 0; i < 6; i++) d += coef[i]/++b;
    return -c + log(2.5066282746310005 * d / a);
}

```

A.5 Classe per la gestione delle reti di Bayes

La classe per la gestione delle reti di Bayes contiene un puntatore ad un oggetto database ed un grafo aciclico orientato nei cui nodi sono registrate informazioni di tipo `node_type`, che, come abbiamo visto, corrispondono alle tabelle di probabilità associate ai nodi. Una descrizione grafica di questa classe può essere osservata in figura 7, in cui l'ellisse tratteggiata attorno a `d_a_graph` indica che questa è un template di classe.

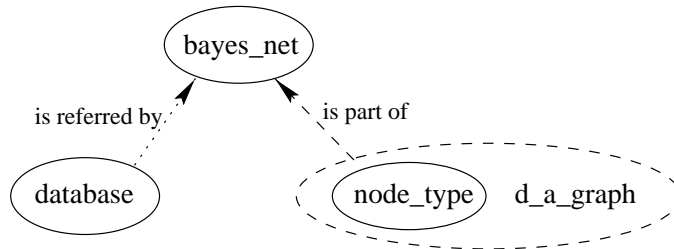


Figura 7: Albero parziale delle classi del progetto, relativo alla classe `bayes_net`.

Oltre a metodi di servizio per collegarsi ad un database (`link_database`), per cancellare le tabelle di probabilità (`del_prob_tbls`) oppure l'intera struttura della rete (`clear_net`), per copiare una rete in un altro oggetto dello stesso tipo (overload dell'operatore `=`), sono state implementate le seguenti procedure:

1. `random_net(int n, int max_ri, float conn)` che genera una rete casuale avente `n` nodi, con al più `max_ri` stati associati a ciascuna variabile (la procedura sceglie a caso un numero di stati da 2 a `max_ri`) e con una probabilità che esista un arco fra un nodo ed un arco pari a `conn`. Il metodo genera per prima cosa una struttura casuale, poi inizializza le varie tabelle di probabilità, che poi riempie in modo casuale. Alla fine del processo le tabelle vengono normalizzate.
2. `sample(int n, database &db)` che genera `n` campioni della rete corrente e li registra nel database `db`. Il meccanismo di campionamento si basa su un algoritmo analogo a quello utilizzato per il controllo dell'aciclicità del grafo: viene fatto un ordinamento topologico della rete a partire da nodi sorgenti, passando per i loro figli e successivamente dai figli dei figli e così via. Ogni volta che un nodo viene ordinato, si genera una realizzazione della variabile aleatoria corrispondente tramite il confronto di un valore casuale con le tabelle di probabilità del nodo. L'ordinamento topologico serve per campionare i nodi con un ordine che garantisca di avere tutte le informazioni dei padri disponibili prima del campionamento effettivo del nodo corrente.
3. `strct_noise(float prob_mod, float prob_add)` che perturba la rete con una probabilità `prob_mod` per modificare gli archi esistenti e `prob_add` per aggiungere nuovi archi. La perturbazione è soltanto strutturale. Dopo che è avvenuta, le tabelle di probabilità vengono modificate per essere dimensionalmente coerenti con la rete e ogni loro elemento è annullato.
4. `node_score(node xi)` che calcola il valore della funzione SCORE relativamente ad un unico nodo, utilizzando la seguente formula:

$$\begin{aligned} \text{SCORE}_i(\mathbf{m}_S) = & \sum_{j=1}^{q_i} \left[\log \Gamma(\alpha_{ij}) - \log \Gamma(\alpha_{ij} + N_{ij}) \right. \\ & \left. + \sum_{k=1}^{r_i} \log \Gamma(\alpha_{ijk} + N_{ijk}) - \log \Gamma(\alpha_{ijk}) \right] \end{aligned} \quad (18)$$

La procedura crea un oggetto temporaneo `parents_node` contenente i dati del nodo passato come argomento. In questo modo, tramite il meccanismo degli iteratori, la formula 18 può essere implementata nel modo presentato nei sorgenti, a nostro avviso molto leggibile.

5. `learn_net_struct(int max_iter)` che apprende la struttura della rete tramite la procedura illustrata nella figura 2. Inizialmente vengono calcolati per ogni nodo i valori di $\text{SCORE}_i(\mathbf{m}_S)$ e memorizzati in un apposito vettore di n elementi. L'algoritmo entra così nei cicli di ricerca in cui vengono scanditi per primi gli archi non ancora presenti nella struttura: si valuta il guadagno di SCORE ottenibile da un loro inserimento (se questo è possibile, mantenendo l'aciclicità del grafo). Se il guadagno è maggiore del massimo guadagno trovato nella corrente iterazione, si salva la mossa. La stessa procedura viene utilizzata per ogni arco presente, valutando il guadagno della cancellazione e dell'eventuale inversione di verso. Alla fine del ciclo, se esiste una mossa migliore delle altre con un incremento positivo del guadagno, si compie, aggiornando al più gli SCORE_i relativi ai nodi pozzo e sorgente del nuovo arco, sommando ad essi la parte del guadagno relativa al nodo in esame. Questa ricerca iterativa è proseguita fino a che non si raggiunge il massimo numero di iterazioni `max_iter` oppure fino a che non esiste più un'ulteriore mossa a guadagno positivo. Una volta appresa la struttura, le tabelle di probabilità relative a ciascun nodo vengono rese compatibili con la nuova rete.
6. `learn_prob_tbls()` che apprende le tabelle di probabilità associate ad i nodi di una particolare struttura di rete. Tale apprendimento avviene utilizzando la formula 12, calcolata per ogni riga di ciascuna tabella (che corrisponde ad una data configurazione dei padri del nodo in esame), in modo da computare prima il numeratore e poi normalizzare tutta la riga con la somma di tutti i numeratori.

La classe si conclude con due funzioni `friend` definite per valutare la bontà delle reti riconosciute nel caso di apprendimento da database D simulati. Queste accettano entrambe in ingresso due oggetti `bayes_net` e calcolano la differenza strutturale (`structural_difference`) e la cross-entropia (`cross_entropy`). Per calcolare la differenza strutturale, viene computata per ciascun nodo la differenza simmetrica, contando il numero dei nodi padri del nodo corrente nella rete Q che non lo erano nella rete P e sommandolo al numero dei padri del nodo corrente nella rete P che non lo sono più nella rete Q . La somma di tutte le differenze simmetriche restituisce quella strutturale fra i grafi.

Il calcolo della cross-entropia si basa invece sulla seguente formula, che rappresenta una riscrittura della 16.

$$\begin{aligned}
H(P, Q) &= \sum_{k_1=1}^{r_1} \dots \sum_{k_n=1}^{r_n} p(X_1 = x_1^{k_1}, \dots, X_n = x_n^{k_n} | \mathbf{m}_P) \log \frac{p(X_1 = x_1^{k_1}, \dots, X_n = x_n^{k_n} | \mathbf{m}_P)}{p(X_1 = x_1^{k_1}, \dots, X_n = x_n^{k_n} | \mathbf{m}_Q)} \\
&= \sum_{k_1=1}^{r_1} \dots \sum_{k_n=1}^{r_n} \prod_{i=1}^n p(X_i = x_i^{k_i} | \mathbf{Pa}_i = \mathbf{pa}_i^{j^P}, \mathbf{m}_P) \log \frac{\prod_{i=1}^n p(X_i = x_i^{k_i} | \mathbf{Pa}_i = \mathbf{pa}_i^{j^P}, \mathbf{m}_P)}{\prod_{i=1}^n p(X_i = x_i^{k_i} | \mathbf{Pa}_i = \mathbf{pa}_i^{j^Q}, \mathbf{m}_Q)}
\end{aligned} \tag{19}$$

In sostanza vengono vagliate tutte le possibili configurazioni che i nodi della rete possono assumere e per ciascuna si calcola l'addendo corrispondente della formula, utilizzando un oggetto di tipo `all_nodes` per implementare in modo trasparente l'operazione di ricerca della configurazione che i nodi padri assumono nelle due reti (rispettivamente $\mathbf{pa}_i^{j^P}$ e $\mathbf{pa}_i^{j^Q}$) per ogni variabile.

Si osservi che è stato implementato un metodo `debug_view`, del tutto provvisorio, che restituisce il formato della rete (struttura e tabelle di probabilità) sul video. Esempi del suo funzionamento possono essere osservati nell'output presente nella sezione 7. Questo metodo, come tutti gli altri denominati `debug`, può essere escluso in fase di compilazione non definendo la variabile di preprocessore `_DEBUG_`.

```

/*
 * Bayes_Net.h
 * bayesian network class
 * (c) 1997 E.Papalini & M.Piccini
 *
 */

#ifndef _Bayesian_Network
#define _Bayesian_Network

#include <iostream.h>
#include "../Vector.h"
#include "../Matrix.h"
#include "../C_Queue.h"
#include "../D_A_Graph.h"
#include "../Nodes.H"
#include "../Database.h"
#include "../Math.h"
#include "../Random.h"

class bayes_net {
private:
    d_a_graph<node_type> DAG;
    database *Data;

    void del_prob_tbls();
    float node_score(node xi); // Dirichlet Bayesian score
                                // hp: alphaijk = alpha / qi / ri

public:
    float alpha; // equivalent sample size

#ifdef _DEBUG_
    void debug_viewer(int op); // DEBUG
#endif // _DEBUG_

#ifdef _PROOF_
    void init();
#endif // _PROOF_

    void clear_net();
    // void load(*file*); not implemented yet
    // void save(*file*); not implemented yet
    void random_net(int n, int max_ri = 5, float conn = 0.5);
    void sample(int n, database &db);
    void strct_noise(float prob_mod, float prob_add);
    void learn_net_strct(int max_iter);
    void learn_prob_tbls();

    void link_database(database &db);
    bayes_net operator=(bayes_net &Obj);

```

```

    friend int structural_difference(bayes_net &Prior, bayes_net &Post);
    friend float cross_entropy(bayes_net &Prior, bayes_net &Post);
};

#endif // _Bayesian_Network

/*
 * Bayes_Net.cc
 * general methods for bayesian networks
 * (c) 1997 E.Papalini & M.Piccini
 */

#include "../Bayes_Net.h"

/* generate a random network */
void bayes_net::random_net(int n, int max_ri = 5, float conn = 0.5) {
    edge e;
    node v;
    // random network structure
    DAG.init(n);
    forall_missing_edges(e,DAG) {
        float prob = rnd(1000)/1000.0;
        if (prob <= conn)
            if (rnd(2) == 1)
                DAG.acyclic_set_edge(e.s,e.t);
            else
                DAG.acyclic_set_edge(e.t,e.s);
    }
    // random number of states for variables
    forall_nodes(v,DAG) {
        DAG(v).i = v;
        DAG(v).ri = rnd(max_ri - 1) + 2;
    }
    // random probability tables
    forall_nodes(v,DAG) {
        node w;
        int qi = 1;

        if (DAG.nPa(v) > 0)
            forall_parents(w,v,DAG)
                qi * = DAG(w).ri;
        DAG(v).P.init(qi,DAG(v).ri);

        for (int j = 1; j <= qi; j++) {
            float tot = 0;
            for (int k = 1; k <= DAG(v).ri; k++)
                tot += DAG(v).P(j,k) = (rnd(10)+1);
            for (int k = 1; k <= DAG(v).ri; k++)
                DAG(v).P(j,k) /= tot;
        }
    }
}

/* sample the network: put n samples in database "db" */
void bayes_net::sample(int n, database &db) {
    vector<int> nParents;
    c_queue<int> orphans;
    vector<int> sample;

    node v, w;

    link_database(db);
    nParents.init(DAG.getN());
    orphans.init(DAG.getN());
    sample.init(DAG.getN());
    Data->init(n,DAG.getN()); // initialize DB
    for (int counter = 1; counter <= n; counter++) {
        // based upon
        // A.B.Kahn, Topological Sorting of Large Networks,
        // Communications of the ACM, vol.5, 558-562, 1962

        // precondition: N > 1
        nParents.set(0);
        sample.set(0);
        orphans.init();

        forall_nodes(v,DAG)
            if ((nParents(v) = DAG.nPa(v)) == 0) orphans.push(v);

        while (!orphans.empty()) {
            int j = 1;
            float cum_prob = 0;
            float prob = rnd(1000) / 1000.0;

            v = orphans.pop();
            // sampling */
            if (DAG.nPa(v) > 0) { // parents configuration already sampled
                node_parents Pav(v,DAG);
                for (int s = 1; s <= Pav.getN(); s++)
                    Pav(s).s = sample(Pav(s).i);
                j = Pav.getj();
            }
        }
    }
}

```

```

        for(int k = 1; k <= DAG(v).ri; k++) // get current variable sample
            if (prob <= (cum_prob += DAG(v).P(j,k))) { sample(v) = k; break; }

        forall_children(w,v,DAG)
            if (--nParents(w) == 0) orphans.push(w);
    }
    Data->insert(counter.sample); // insert a new datum
}
}

/* link an existing database */
void bayes_net::link_database(database &db) { Data = &db; }

/* copy operator */
bayes_net bayes_net::operator=(bayes_net &Obj) {
    if(Obj.DAG.getN() > 1) {
        node v;
        edge e;

        Data = Obj.Data; // copy pnt to DataBase
        DAG.init(Obj.DAG.getN()); // init graph
        forall_edges(e,Obj.DAG) // copy graph edges
            DAG.set_edge(e);
        forall_nodes(v,Obj.DAG) { // copy graph nodes
            DAG(v).i = Obj.DAG(v).i;
            DAG(v).ri = Obj.DAG(v).ri;
            DAG(v).P.init(Obj.DAG(v).P.getN(),Obj.DAG(v).P.getM()); // copy prob.
            for(int i = 1; i <= DAG(v).P.getN(); i++)
                for(int j = 1; j <= DAG(v).P.getM(); j++)
                    DAG(v).P(i,j) = Obj.DAG(v).P(i,j);
        }
        return *this;
    }
    cout << "Error: Object not initialized.\n";
    exit(1);
}

/* noise perturbation of current net structure */
void bayes_net::struct_noise(float prob_mod, float prob_add) {
    // prob_mod : probability of changing an edge
    // prob_add : probability of adding an edge
    float prob;
    edge e;
    bayes_net aux;

    aux = *this;
    forall_edges(e,aux.DAG) { // noise on edges already present
        prob = rnd(1000) / 1000.0;
        if (prob <= prob_mod)
            if (rnd(2) == 1)
                DAG.del_edge(e);
            else
                DAG.acyclic_set_edge(e,t,e,s);
    }
    forall_missing_edges(e,DAG) { // insert new edges
        prob = rnd(1000) / 1000.0;
        if (prob <= prob_add)
            if (rnd(2) == 1)
                DAG.acyclic_set_edge(e,s,e,t);
            else
                DAG.acyclic_set_edge(e,t,e,s);
    }
    del_prob_tbls();
}

/* delete probability tabels */
void bayes_net::del_prob_tbls() {
    int qi;
    node v,w;

    forall_nodes(v,DAG) {
        DAG(v).i = v;
        qi = 1;
        if(DAG.nPa(v) > 0)
            forall_parents(w,v,DAG)
                qi *= DAG(w).ri;
        DAG(v).P.init(qi,DAG(v).ri);
        DAG(v).P.set(0);
    }
}

/* clear all edges */
void bayes_net::clear_net() {
    node v;
    edge e;

    forall_nodes(v,DAG) {
        DAG(v).i = v;
        forall_edges(e,DAG) // clear all edges
            DAG.del_edge(e);
        DAG(v).P.init(1,DAG(v).ri);
        DAG(v).P.set(0);
    }
}

```

```

#ifdef _DEBUG_
/* view network structure & probability tables */
void bayes_net::debug_viewer(int op) {
    if (DAG.getN() > 0) {
        node v;
        edge e;

        cout << "# Nodes : " << DAG.getN() << "\n";
        cout << "\n- edges list ----- \n";
        forall_edges(e,DAG)
            cout << e.s << " -> " << e.t << "\n";
        if(op == 0) return;
        cout << "\n- nodes tables ----- \n";
        forall_nodes(v,DAG) {
            if (DAG.nPa(v) != 0) {
                node_parents Pav(v,DAG);
                int j,s;
                cout << "*Node " << DAG(v).i ;
                cout << " with " << DAG(v).ri << " states & " << DAG(v).P.getN() << " parents config.\n";
                view_parents_state(s,Pav)
                    cout << Pav(s).i << " ";
                cout << " | ";
                for(int k = 1; k <= DAG(v).ri; k++)
                    cout << k << " ";
                cout << "\n";
                for(int k = 1; k <= (9*DAG(v).ri+2*DAG.nPa(v)+1); k++)
                    cout << "-";
                cout << "\n";
                forall_parents_states(j,Pav) {
                    view_parents_state(s,Pav)
                        cout << Pav(s).s << " ";
                    cout << " | ";
                    cout.setf(ios::showpoint | ios::fixed);
                    for(int k = 1; k <= DAG(v).ri; k++)
                        cout << DAG(v).P(j,k) << " ";
                    cout << "\n";
                    cout.unsetf(ios::showpoint | ios::fixed);
                }
            } else {
                cout << "*Node " << DAG(v).i ;
                cout << " with " << DAG(v).ri << " states\n";
                for(int k = 1; k <= DAG(v).ri; k++)
                    cout << k << " ";
                cout << "\n";
                for(int k = 1; k <= (9*DAG(v).ri+1); k++)
                    cout << "-";
                cout << "\n";
                cout.setf(ios::showpoint | ios::fixed);
                for(int k = 1; k <= DAG(v).ri; k++)
                    cout << DAG(v).P(1,k) << " ";
                cout << "\n";
                cout.unsetf(ios::showpoint | ios::fixed);
            }
        }
    }
}
#endif // _DEBUG_

#ifdef _PROOF_
/* College Plans BN initialization */
void bayes_net::init() {
    DAG.init(5);
    DAG(1).i = 1;
    DAG(1).ri = 2;
    DAG(2).i = 2;
    DAG(2).ri = 4;
    DAG(3).i = 3;
    DAG(3).ri = 4;
    DAG(4).i = 4;
    DAG(4).ri = 2;
    DAG(5).i = 5;
    DAG(5).ri = 2;
}
#endif // _PROOF_

/*
 * DB_score.cc
 * general methods for calculating score on bayesian networks
 * (c) 1997 E.Papalini & M.Piccini
 */

#include "../Bayes_Net.h"

/* calculate bayesian-dirichlet score */
float bayes_net::node_score(node v) {
    float tot = 0;
    node_info xi;

    xi.i = v;

```

```

xi.ri = DAG(v).ri;
xi.s = 0;

if(DAG.nPa(xi.i) > 0) { // xi has some parents
    int j,Nij,Nijk;
    node_parents Pai(xi.i,DAG);

    forall_parents_states(j,Pai) {
        Nij = 0;
        forall_node_states(xi.s,xi) {
            Nij += Nijk = Data→query(xi,Pai);
            tot += lngamma(alpha / Pai.getQ() / xi.ri + Nijk) -
                lngamma(alpha / Pai.getQ() / xi.ri);
        }
        tot += lngamma(alpha / Pai.getQ()) -
            lngamma(alpha / Pai.getQ() + Nij);
    }
} else { // xi hasn't any parent
    int Nik, Ni;

    Ni = 0;
    forall_node_states(xi.s,xi) {
        Ni += Nik = Data→query(xi);
        tot += lngamma(alpha / xi.ri + Nik) - lngamma(alpha / xi.ri);
    }
    tot += lngamma(alpha) - lngamma(alpha + Ni);
}
return tot;
}

/* learn probability tables using dirichlet approach */
void bayes_net::learn_prob_tbls() {
    node v;
    forall_nodes(v,DAG) {
        node_info xi;

        xi.i = v;
        xi.ri = DAG(v).ri;
        xi.s = 0;

        #ifdef _DEBUG_
            cout << " Learning prob. tables for node " << v << "\n";
        #endif // _DEBUG_

        if(DAG.nPa(xi.i) > 0) { // xi has some parents
            int j,Nij,Nijk;
            node_parents Pai(xi.i,DAG);

            forall_parents_states(j,Pai) {
                Nij = 0;
                forall_node_states(xi.s,xi) {
                    Nij += Nijk = Data→query(xi,Pai);
                    DAG(v).P(j,xi.s) = Nijk + alpha / Pai.getQ() / DAG(v).ri;
                }
                forall_node_states(xi.s,xi) { // normalization
                    DAG(v).P(j,xi.s) /= (Nij + alpha / Pai.getQ());
                }
            }
        } else { // xi hasn't any parent
            int Nik, Ni;

            Ni = 0;
            forall_node_states(xi.s,xi) {
                Ni += Nik = Data→query(xi);
                DAG(v).P(1,xi.s) = Nik + alpha / DAG(v).ri;
            }
            forall_node_states(xi.s,xi) { // normalization
                DAG(v).P(1,xi.s) /= (Ni + alpha);
            }
        }
    }
}

/*
 * Hill_Climbing.cc
 * method for structural search (bayesian networks)
 * (c) 1997 E.Papalini & M.Piccini
 *
 */

#include "../Bayes_Net.h"

void bayes_net::learn_net_struct(int max_iter) {
    int counter;
    node v;
    edge e, new_edge;
    enum {null = 0, add, inv, del} best_move;
    float tot, delta, max_delta;
    float delta_s, delta_t, max_delta_s, max_delta_t;
    vector<float> current_score;

    current_score.init(DAG.getN());

    // initialize structures

```

```

forall_nodes(v,DAG)
    current_score(v) = node_score(v);

// iteration
for (counter = 1; counter <= max_iter; counter++) {

#ifdef _DEBUG_
    cout << " " << counter << "\n search iteration:"; // DEBUG
    cout.setf(ios::showpoint | ios::fixed | ios::unitbuf); // DEBUG
    tot = 0; // DEBUG
    forall_nodes(v,DAG) // DEBUG
        tot += current_score(v); // DEBUG
    cout << " current score = " << tot; // DEBUG
#endif // _DEBUG_

    max_delta = max_delta_s = max_delta_t = 0.0;
    new_edge.s = 0; new_edge.t = 0;
    best_move = null;
    // score all possible single change
    forall_missing_edges(e,DAG) {
        // insert edge: e.s e.t => e.s -> e.t
        if(DAG.acyclic_set_edge(e.s,e.t)) {
            delta = node_score(e.t) - current_score(e.t);
            if (delta > max_delta) {
                max_delta = delta;
                new_edge.s = e.s; new_edge.t = e.t;
                best_move = add;
            }
            DAG.del_edge(e.s,e.t);
        }
        // insert edge: e.s e.t => e.s <- e.t
        if(DAG.acyclic_set_edge(e.t,e.s)) {
            delta = node_score(e.s) - current_score(e.s);
            if (delta > max_delta) {
                max_delta = delta;
                new_edge.s = e.t; new_edge.t = e.s;
                best_move = add;
            }
            DAG.del_edge(e.t,e.s);
        }
    }
    forall_edges(e,DAG) {
        // delete edge: e.s -> e.t => e.s e.t
        DAG.del_edge(e.s,e.t);
        delta = node_score(e.t) - current_score(e.t);
        if (delta > max_delta) {
            max_delta = delta;
            new_edge.s = e.s; new_edge.t = e.t;
            best_move = del;
        }
        DAG.set_edge(e.s,e.t);
        // revert edge: e.s -> e.t => e.s <- e.t
        if(DAG.acyclic_set_edge(e.t,e.s)) {
            delta_s = node_score(e.t) - current_score(e.t);
            delta_t = node_score(e.s) - current_score(e.s);
            delta = delta_s + delta_t;
            if (delta > max_delta) {
                max_delta = delta;
                max_delta_s = delta_s; max_delta_t = delta_t;
                new_edge.s = e.t; new_edge.t = e.s;
                best_move = inv;
            }
            DAG.set_edge(e.s,e.t);
        }
    }
}

if (max_delta > 0) { //perform best change & update current score

#ifdef _DEBUG_
    cout << " + " << max_delta; // DEBUG
    cout.unsetf(ios::showpoint | ios::fixed | ios::unitbuf); // DEBUG
#endif // _DEBUG_

    switch(best_move) {
        case add:
            DAG.set_edge(new_edge);
            current_score(new_edge.t) += max_delta;

#ifdef _DEBUG_
            cout << " (add: " << new_edge.s << " -> " << new_edge.t << " )\n"; // DEBUG
#endif // _DEBUG_

            break;
        case inv:
            DAG.set_edge(new_edge);
            current_score(new_edge.s) += max_delta_s;
            current_score(new_edge.t) += max_delta_t;

#ifdef _DEBUG_
            cout << " (rev: " << new_edge.s << " -> " << new_edge.t << " )\n"; // DEBUG
#endif // _DEBUG_

            break;
        case del:
    }
}

```

```

        DAG.del_edge(new_edge);
        current_score(new_edge.t) += max_delta;

        #ifdef _DEBUG_
            cout << " (del: " << new_edge.s << " -X " << new_edge.t << " )\n"; // DEBUG
        #endif // _DEBUG_

        break;
    default:
        cout << "Error: Undef. move.\n";
        exit(10);
    }
} else {

    #ifdef _DEBUG_
        cout.unsetf(ios::showpoint | ios::fixed | ios::unitbuf); // DEBUG
        cout << " No better changes.\n"; // DEBUG
    #endif // _DEBUG_

    break;
}
}
#ifdef _DEBUG_
if (counter > max_iter)
    cout << " Max iteration reached.\n";
#endif // _DEBUG_
del_prob.tbls();
}

/*
 * Measures.cc
 * methods for results evaluation
 * (c) 1997 E.Papalini & M.Piccini
 */

#include "../Bayes.Net.h"

/* compute structural difference between two bayesian networks on X */
int structural_difference(bayes_net &Prior, bayes_net &Post) {
    node v,s,t;
    int c, tot = 0;

    forall_nodes(v, Post.DAG) {
        if (Post.DAG.nPa(v) != 0 && Prior.DAG.nPa(v) != 0) {
            node_parents Pai(v, Post.DAG), Prior_Pai(v, Prior.DAG);

            view_parents_state(s, Pai) {
                // count parents of v which weren't its parents in Prior (1)
                c = 1;
                view_parents_state(t, Prior_Pai)
                    if (Pai(s).i == Prior_Pai(t).i) { c = 0; break; }
                tot += c;
            }
            view_parents_state(s, Prior_Pai) {
                // count parents of v in Prior which aren't its parents now (2)
                c = 1;
                view_parents_state(t, Pai)
                    if (Prior_Pai(s).i == Pai(t).i) { c = 0; break; }
                tot += c;
            }
        } else {
            if (Post.DAG.nPa(v) != 0) tot += Post.DAG.nPa(v);
            if (Prior.DAG.nPa(v) != 0) tot += Prior.DAG.nPa(v);
        }
    }
    return tot; // (1) + (2) forall nodes
}

/* compute cross entropy between two bayesian networks on X */
float cross_entropy(bayes_net &Prior, bayes_net &Post) {
    int dummy;
    node v;
    all_nodes aux_nodes(Prior.DAG, Post.DAG);
    float totP, totQ, tot = 0.0;

    forall_nodes_states(dummy, aux_nodes) {
        totP = totQ = 1;
        forall_nodes(v, Post.DAG) {
            totP *= (Prior.DAG(v).P(aux_nodes.getj_P(v), aux_nodes(v).s));
            totQ *= (Post.DAG(v).P(aux_nodes.getj_Q(v), aux_nodes(v).s));
        }
        tot += (totP * (log(totP) - log(totQ)));
    }
    return tot;
}

```

A.6 Esempi di funzioni main()

Per concludere la nostra presentazione dell'implementazione, ecco due esempi di funzioni main(). Il primo implementa un semplice comando in linea a cui possono essere forniti diversi parametri e che da come risultato l'output a video della procedura di apprendimento di una rete simulata. Un esempio, leggermente modificato, si trova nella sezione 6.

Il secondo carica il database di Sewell e Shah ed attua l'apprendimento a partire da una *prior network* vuota. Nella sezione 7 vi è riportato il suo output.

```
/*
 * Test.cc
 * main for evaluation on simulated bayesian networks
 * (c) 1997 E.Papalini & M.Piccini
 *
 */

#include <stdlib.h>
#include <iostream.h>
#include "../Bayes_Net.h"

int main(int argc, char * argv[] ) {

    unsigned int last_idum;
    bayes_net gold_net,net;
    database samples;

    if (argc == 2) {
        cout << "USAGE: <# nodes> <# max states> <conn. prob.> \\|n";
        cout << " <# samples> \\|n";
        cout << " <noise mod. prob.> <noise add prob.> (<0> <0> for empty prior network) \\|n";
        cout << " <eq. sample size> \\|n";
        cout << " <# max of iterations> \\n";

        return 0;
    }

    cout << "\\n\\n+ net initialize +++++\\n";

    if (argc > 3)
        gold_net.random_net(atoi(argv[1]), atoi(argv[2]), atof(argv[3]));
    else
        gold_net.random_net(5, 4, 0.5);

    last_idum = get_idum();
    if (argc > 4)
        gold_net.sample(atoi(argv[4]), samples);
    else
        gold_net.sample(5000, samples);

    net = gold_net;
    randomize(last_idum);

    if (argc > 6)
        if (atof(argv[5]) == 0.0 && atof(argv[6]) == 0.0)
            net.clear_net();
        else
            net.strct_noise(atof(argv[5]), atof(argv[6]));
    else
        net.strct_noise(0.2,0.3);

    if (argc > 7)
        net.alpha = atof(argv[7]);
    else
        net.alpha = 8.0;

    cout << "\\nGOLD NETWORK:\\n";
    gold_net.debug_viewer(1);
    cout << "\\nPRIOR NETWORK:\\n";
    net.debug_viewer(0);
    cout << "\\n PRIOR NETWORK vs GOLD NETWORK\\n";
    cout << " structural-difference measure: " << structural_difference(gold_net,net) << "\\n";

    cout << "\\n\\n+ learning +++++\\n";

    cout << "- structure -\\n";
    if (argc > 8)
        net.learn_net_strct(atoi(argv[8]));
    else
        net.learn_net_strct(10);

    cout << "- prob. tables -\\n";
    net.learn_prob_tbls();

    cout << "\\n\\n+ results +++++\\n";
    cout << "\\n LEARNED NETWORK:\\n";
    net.debug_viewer(1);
}
```

```

cout << "\n LEARNED NETWORK vs GOLD NETWORK\n";
cout << " structural-difference measure: " << structural_difference(gold_net.net) << "\n";
cout << " cross-entropy measure: " << cross_entropy(gold_net.net) << "\n";
}

/*
 * Test.cc
 * main for evaluation on "College Plans" database
 * (c) 1997 E.Papalini & M.Piccini
 *
 */

#include <iostream.h>
#include <fstream.h>
#include "../Bayes_Net.h"

int main() {
    bayes_net net;
    int number = 0;
    int firstj = 1;
    vector<int> datum;
    database CollegePlans;
    ifstream in("MyTable.txt");

    datum.init(5);
    CollegePlans.init(10416,5);

    cout << "\n\n+ reading database ++++++\n";
    cout << "\n College Plans \n";
    cout << " Sewall & Shah, (c) 1968 The University of Chicago.\n";
    for(int sex = 1; sex <= 2; sex++)
        for(int ses = 1; ses <= 4; ses++)
            for(int iq = 1; iq <= 4; iq++)
                for(int pe = 1; pe <= 2; pe++)
                    for(int cp = 1; cp <= 2; cp++) {
                        in >> number;
                        datum(1) = sex;
                        datum(2) = ses;
                        datum(3) = iq;
                        datum(4) = pe;
                        datum(5) = cp;
                        for(int j = firstj; j < (firstj + number); j++)
                            CollegePlans.insert(j,datum);
                        firstj += number;
                    }

    in.close();

    net.init();
    net.clear_net();
    net.link_database(CollegePlans);
    net.alpha = 8.0;
    cout << "\n\n+ learning ++++++\n";

    cout << "- structure -\n";

    net.learn_net_struct(40);

    cout << "- prob. tables -\n";
    net.learn_prob_tbs();

    cout << "\n\n+ results ++++++\n";
    cout << "\n LEARNED NETWORK:\n";
    net.debug_viewer(1);
    return 0;
}

```

B Notazione

X_1, \dots, X_n	Variabili aleatorie oppure i loro nodi corrispondenti in una rete di Bayes.
\mathbf{X}	Insieme di variabili oppure insieme di nodi.
$X_i = x_i^k$	Variabile aleatoria X_i che assume lo stato x_i^j .
\mathbf{M}	La variabile aleatoria che rappresenta l'incertezza sul modello.
\mathbf{m}	Un particolare modello realizzazione di \mathbf{M} .
\mathbf{T}_m	La variabile aleatoria che rappresenta l'incertezza sui parametri del modello \mathbf{m} .
θ_m	Un particolare insieme di parametri realizzazione di \mathbf{T}_m .
D	Database di esempi.
\mathbf{x}_l	l -esimo caso all'interno del database D .
N	Numero di esempi in un database D .
$p(x y)$	Probabilità che $X = x$ quando è noto che $Y = y$ ed il resto non è rilevante. Attenzione: lo stesso simbolo è anche usato per descrivere densità di probabilità per variabili continue e distribuzioni di probabilità.
S	Una struttura di una rete di Bayes corrispondente ad un grafo diretto aciclico.
P	Un insieme di distribuzioni locali di probabilità associate ai nodi di una rete di Bayes.
\mathbf{m}_S	Modello descritto dalla struttura S della rete di Bayes.
$p(X_i \mathbf{Pa}_i, \mathbf{M}_S, \mathbf{T}_{m_S})$	Distribuzione locale associata al nodo X_i della rete di Bayes.
\mathbf{Pa}_i	Le variabili o i nodi che corrispondono ai padri di X_i nella rete di Bayes.
\mathbf{pa}_i	Una configurazione delle variabili \mathbf{Pa}_i .
r_i	Numero di stati della variabile aleatoria discreta X_i .
q_i	Numero di configurazioni delle variabili aleatorie discrete \mathbf{Pa}_i .
n	Numero di nodi in una rete di Bayes.
θ_{ijk}	Il parametro della distribuzione multinomiale corrispondente alla probabilità $p(X_i = x_i^k \mathbf{Pa}_i = \mathbf{pa}_i^j)$.
θ_{ij}	$= \{\theta_{ij1}, \dots, \theta_{ijr_i}\}$.
θ_i	$= \{\theta_{i1}, \dots, \theta_{iq_i}\}$.
θ_s	$= \{\theta_1, \dots, \theta_n\}$.
$\Gamma(n)$	$= \int_0^\infty x^{n-1} e^{-x} dx$ funzione gamma di Eulero.
α	Dimensione di un campione equivalente.
$\text{Dir}(\theta_{ij}, \alpha_{ij1}, \dots, \alpha_{ijr_i})$	Distribuzione di Dirichlet.
α_{ijk}	Iperparametri della distribuzione di dirichlet corrispondenti a θ_{ijk} .
α_{ij}	$= \sum_{k=1}^{r_i} \alpha_{ijk}$.
N_{ijk}	Numero di campioni in D tali che $X_i = x_i^k$ e $\mathbf{Pa}_i = \mathbf{pa}_i^j$.
N_{ij}	$= \sum_{k=1}^{r_i} N_{ijk}$.
δ_i	Differenza simmetrica fra i padri di X_i in due differenti reti di Bayes.
δ	Differenza strutturale fra due reti di Bayes.
$\#\mathbf{A}$	Numero di elementi dell'insieme \mathbf{A} .
$H(P, Q)$	Cross-entropia fra le tabelle di probabilità associate alle reti di Bayes P e Q .