

JUNCTION TREE ALGORITHM

Cavenaghi Emanuele, Zanga Alessio



SCELTE IMPLEMENTATIVE

Python

Strutture implementate

Collegamenti alla libreria BNlearn

MAD

SCELTE IMPLEMENTATIVE

Python

È stato scelto di utilizzare il linguaggio Python per svariati motivi, primo tra tutti il solido supporto e il sempre più vasto impiego.

NetworkX[1]

Altra motivazione è la possibilità di avvalersi delle numerose librerie disponibili tra le quali «**NetworkX**» che rende disponibili strutture per la gestione di grafi, grafi orientati, alberi, ecc. Insieme alle strutture è ovviamente disponibile una vasta collezione di algoritmi su grafi con cui poter confrontare i risultati ottenuti dalle nostre implementazioni.

Numba[2]

Infine, è presente la libreria «**Numba**» che consiste in un Just-In-Time compiler per Python in modo da ottenere codice altamente efficiente. Si mantiene così il vantaggio di scrivere codice di alto livello combinandolo con il vantaggio di velocità dato dalla compilazione di codice a basso livello.

Inoltre, dove possibile, rende più facile il compito di parallelizzare le funzioni sia per CPU che per GPU supportando il linguaggio Nvidia CUDA.



SCELTE IMPLEMENTATIVE

Strutture implementate

Sono state implementate diverse strutture generiche nella libreria creata per poter immagazzinare le informazioni richieste durante la costruzione dell'algoritmo Junction Tree.

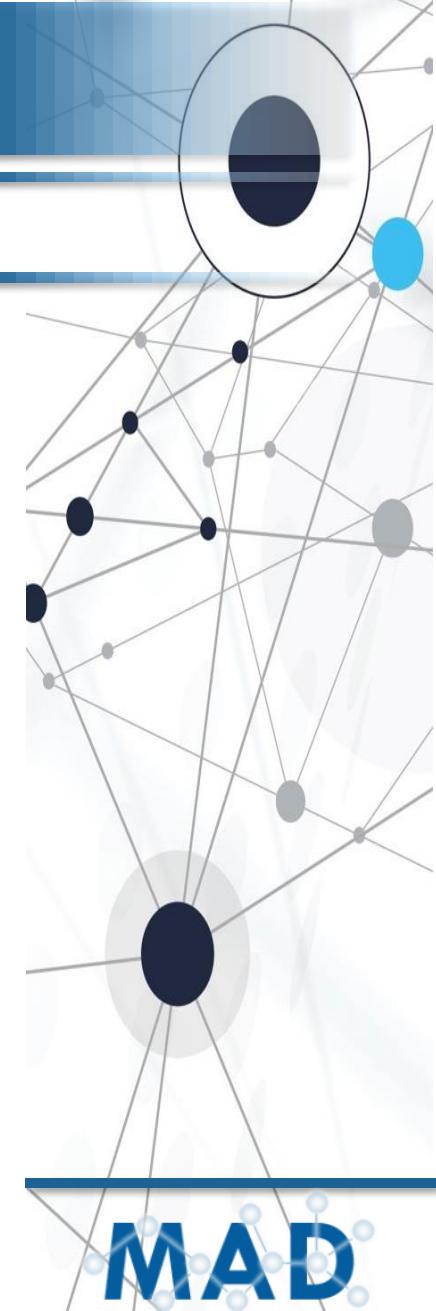
Graph

La classe **Graph** rappresenta un generico grafo, implementato tramite matrice di adiacenza binaria, contiene tutti i metodi per poter operare e manipolare il grafo. Estendendo questa classe è stata creata la classe **DirectedGraph** che rappresenta in modo specifico grafi orientati, differenziando, per esempio, tra nodi *parent* e nodi *child*, non presenti nella classe **Graph**.

Tree

La classe **Tree** rappresenta un generico albero costituito da nodi di cui si conosce però solo il nodo *root*. Parte fondamentale della classe è la classe **Node** che contiene al suo interno i riferimenti ai propri nodi *child* e al proprio nodo *parent* oltre agli attributi del nodo stesso.

La classe **JunctionTree** è un'estensione della classe **Tree** e ne condivide tutti gli attributi e metodi oltre ad avere una funzione di plot personalizzata e una funzione che consente, dato il DAG di una Bayesian Network, di costruire il corrispondente Junction Tree.

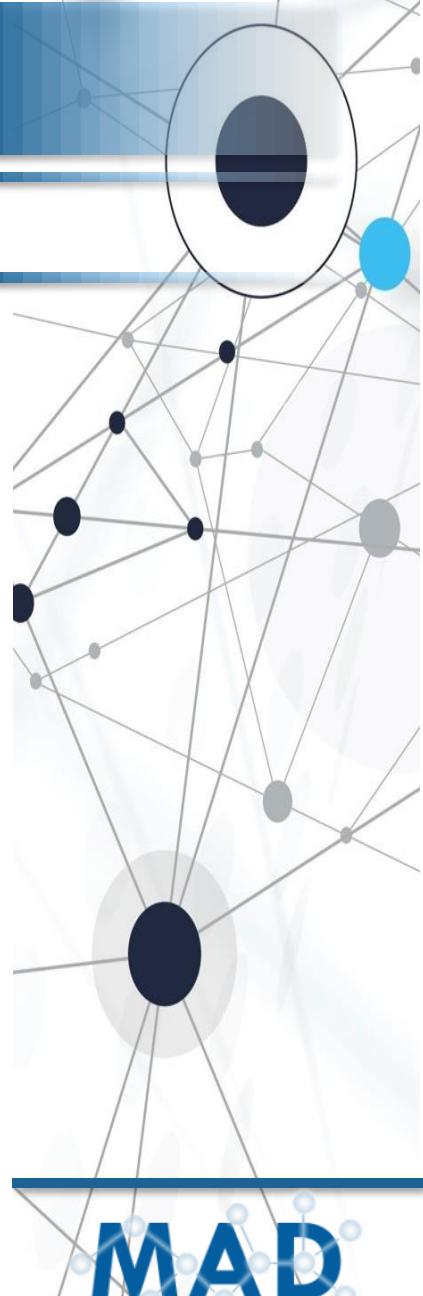


SCELTE IMPLEMENTATIVE

Collegamenti alla libreria BNlearn

La libreria BNlearn[3] resta comunque un punto di partenza per eseguire comparazioni, test e verifiche vista la grande vastità delle risorse disponibili e gli anni di test e miglioramenti apportati.

- Creazione del DAG partendo dalla struttura in stringa (e.g. [A][C][B | A:C])
- Import dei dataset disponibili
- Import dei file delle Bayesian Networks con le *Probability Conditional Table* nei vari formati



MAD

JUNCTION TREE ALGORITHM

Costruzione del Junction Tree

Calcolo dei beliefs

Inferenza sul Junction Tree

Aggiunta di evidenze

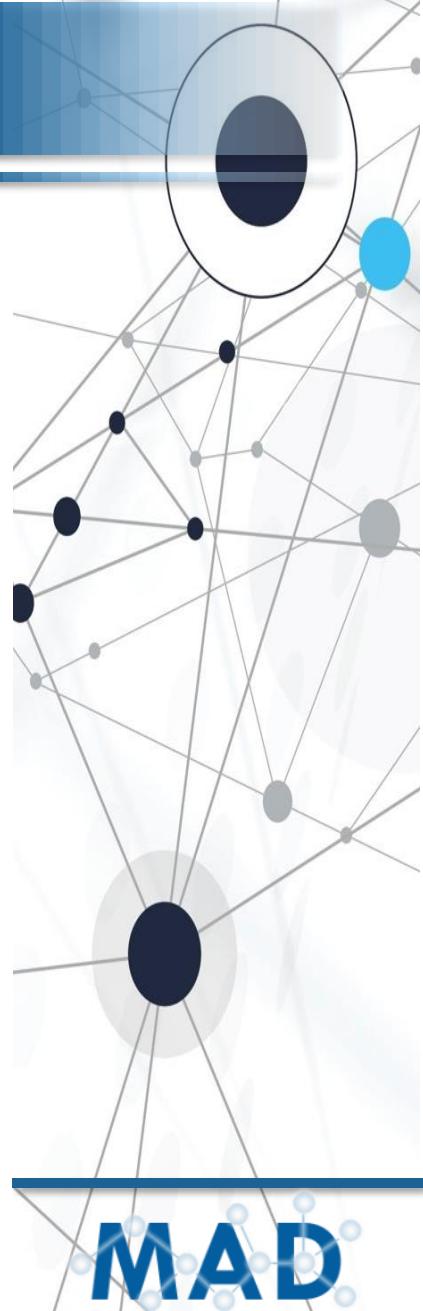
MAD

PROPRIETÀ DEI JUNCTION TREE

Supponendo di avere un grafo non orientato $G = (X, E_G)$, un Junction Tree $T = (C, E_T)$ su G è un albero i cui nodi $c \in C$ sono sottoinsiemi $x_c \subseteq X$ dei vertici del grafo. Il Junction Tree deve soddisfare le proprietà:

- **Family Preservation:** per ogni fattore c'è almeno un nodo nel Junction Tree che lo contiene
- **Running Intersection Property:** per ogni coppia di nodi del Junction Tree che contengono una stessa variabile $x \in X$ deve esistere un cammino che unisce i due nodi in cui la variabile x sia sempre presente

Nel caso di una rete Bayesiana il grafo corrispondente è un DAG quindi prima di poterne costruire il Junction Tree corrispondente è necessario convertirlo in un grafo non orientato moralizzando il grafo.



COSTRUZIONE DEL JUNCTION TREE

Moralizzazione

Triangolazione

Trovare le Cliques

Connessione delle Cliques

MAD

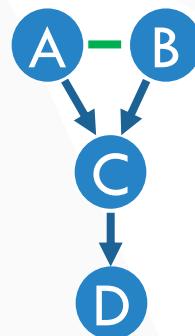
MORALIZZAZIONE

La prima fase di costruzione del Junction Tree da una rete bayesiana (BN) consiste nella moralizzazione del DAG associato alla BN poiché la successiva fase di triangolazione è definita per grafi non orientati.

Nel processo di moralizzazione vengono uniti da un arco i nodi che hanno un figlio in comune.

Per individuare i *parent* di un nodo sfruttiamo il fatto che questi sono individuati sulla colonna corrispondente dove il valore è *True*. Se una colonna ha due o più valori a *True* i nodi sulle righe corrispondenti andranno collegati da un arco.

ESEMPIO



	A	B	C	D
A	0	1	1	0
B	1	0	1	0
C	0	0	0	1
D	0	0	0	0

I *parent* del nodo C sono individuati sulla colonna corrispondente dove il valore è *True*

L'arco da aggiungere è l'arco *A – B*

L'ultimo passaggio della moralizzazione è di rendere il grafo non orientato e questo viene fatto semplicemente tramite un or con la matrice di adiacenza trasposta.



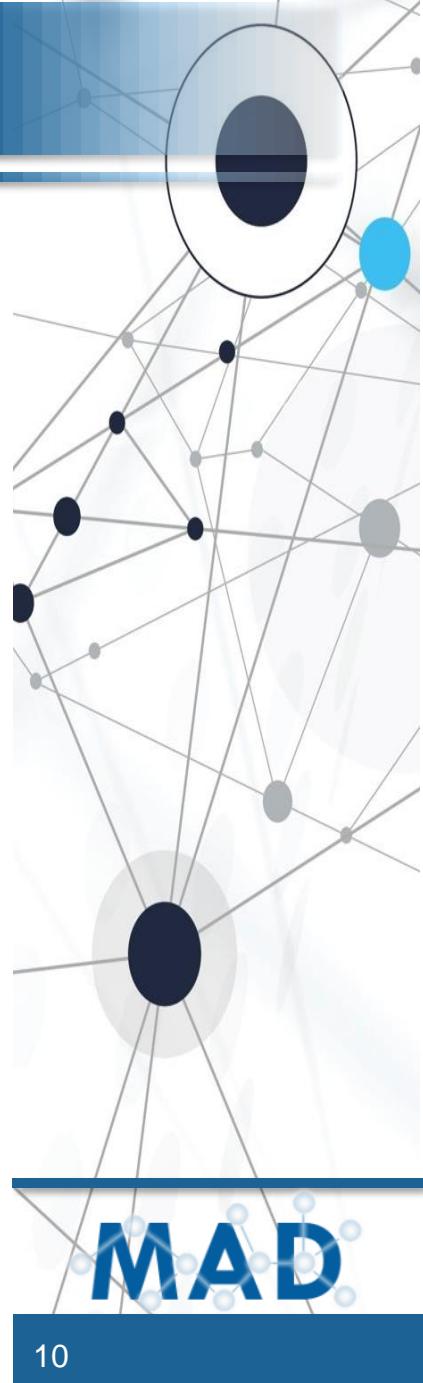
MAD

MORALIZZAZIONE

La funzione di moralizzazione è stata progettata da zero per poter essere altamente efficiente. Per fare ciò abbiamo sfruttato l'implementazione del DAG come grafo salvato su una matrice di adiacenza binaria.

```
def moralize(adjacency_matrix):
    columns = adjacency_matrix.columns.size
    for node in range(columns):
        for i in node.parents:
            for j in node.parents:
                if i < j:
                    out[i,j] = True
    out = out or adjacency_matrix
    out = out or out.T
    return out
```

Inoltre è possibile eseguire parallelamente sia la fase di ricerca che di collegamento dei parent in quanto ogni nodo può essere analizzato in modo indipendente. Infine tutte le operazioni coinvolgono solamente valori binari *inplace* rendendo efficienti i calcoli.

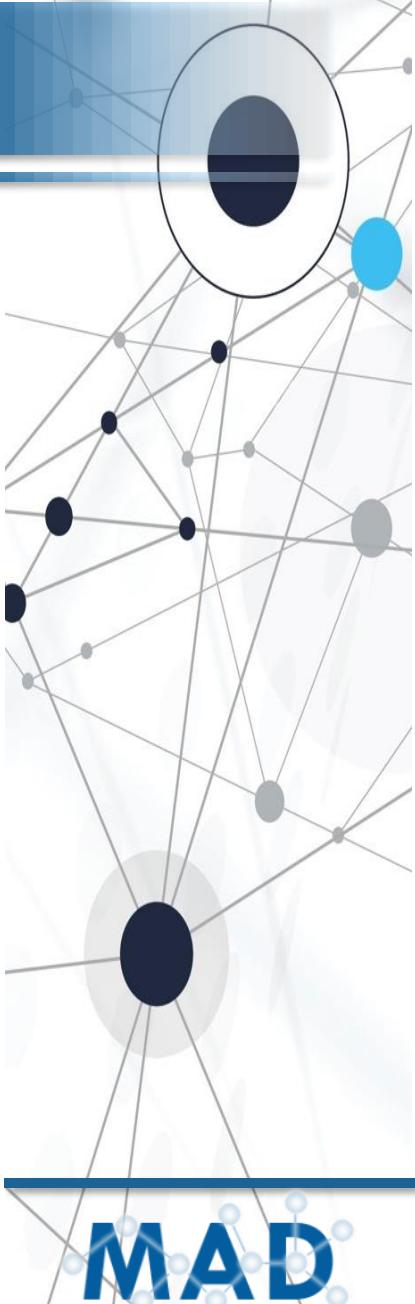


TRIANGOLAZIONE

La fase successiva è la triangolazione del grafo non orientato ottenuto precedentemente. Per fare ciò è stato implementato l'algoritmo «*Maximum Cardinality Search Fill-In*», descritto in [4], di cui viene presentato uno pseudo-codice.

Un grafo non orientato è detto triangolato se ogni ciclo di lunghezza maggiore o uguale a quattro ha almeno una corda, cioè un arco che collega due vertici non adiacenti del ciclo.

```
def MCS(adjacency_matrix):
    nodes = adjacency_matrix.nodes
    numbered[1] = nodes[0]
    while i < nodes.size:
        x = unnumbered node with maximum neighborhood
        numbered[i] = x
        if x.neighbors.intersection(numbered) is not complete:
            Add missing edges to out
            i = 0
        i++
    out = out or adjacency_matrix
    out = out or out.T
    return out
```



RICERCA DELLE CLIQUES

La ricerca delle cliques viene effettuata utilizzando l'algoritmo «Bron-Kerbosh»[5] nella versione originale.

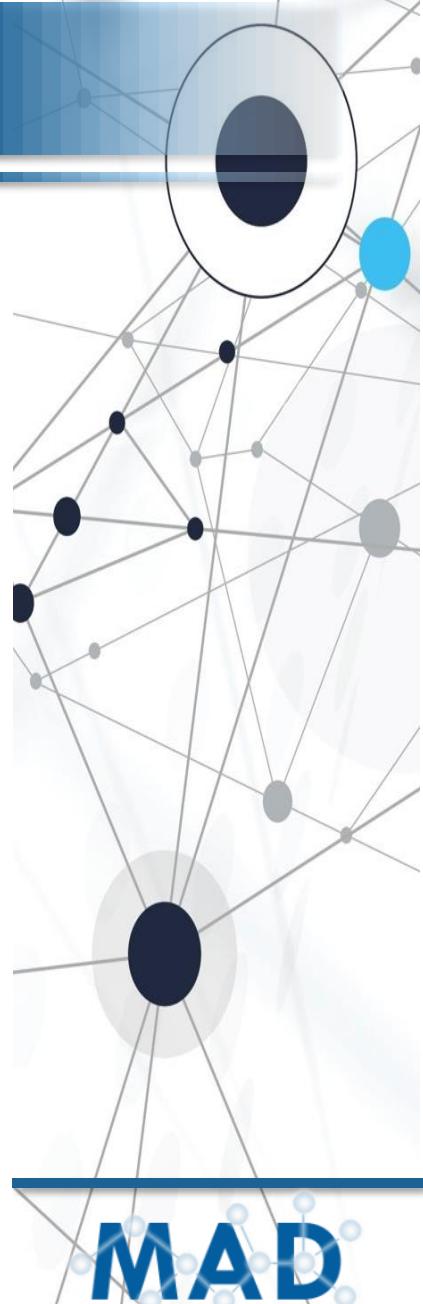
Una clique è un insieme di nodi adiacenti tra loro. In Bron-Kerbosh si seleziona un nodo e, tramite operazioni insiemistiche sul suo vicinato, si prova incrementalmente ad allargare la clique in costruzione. Se non è possibile espandere la clique con altri nodi allora essa è massimale.

```
def bron_kerbosh(A, B, C):
    if B is Empty and C is Empty:
        return A
    out = []
    for node in B:
        out += bron_kerbosh(
            A.union({node}),
            B.intersection(node.neighbors),
            C.intersection(node.neighbors)
        )
        B = B.difference({node})
        C = C.union({node})
    return out
```

Gli insiemi A, B e C significano rispettivamente:

- A – Clique in costruzione
- B – Nodi da aggiungere
- C – Nodi aggiunti

L'algoritmo viene inizializzato ponendo A e C come insiemi vuoti e B come l'insieme che contiene tutti i nodi del grafo.

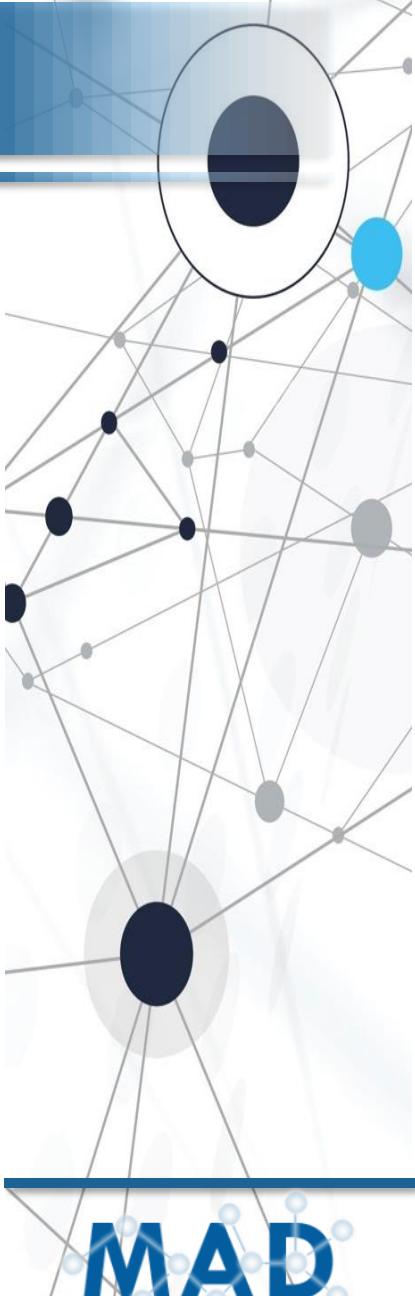


CONNESSIONE DELLE CLIQUES

Trovare la struttura ‘migliore’ del Junction Tree, come connettere tra loro le cliques trovate, è un problema NP-Completo. Si è scelto di implementare l’algoritmo «Generating a Join free»[4] che richiede in ingresso, oltre al grafo ottenuto tramite triangolazione, la lista delle cliques ordinata secondo una funzione di numbering che viene usata anche dall’algoritmo *Maximum Cardinality Search Fill-In*.

L’idea dell’algoritmo è quella di aggiungere una clique alla volta collegandola a quelle già inserite seguendo l’ordine della lista delle cliques per assicurarsi di non ripetere clique e di creare un albero

```
def build_junction_tree(graph, cliques):
    Assign to each clique the largest perfect number of its nodes
    Sort cliques in ascending order according to their assigned numbers
    for i in range(cliques.size()):
        Choose a clique  $C_k$  from clique[1:i-1]
            with maximum number of common nodes
        Add the link  $C_i - C_k$  to the junction_tree
        Add the separator of  $C_i - C_k$  to the junction_tree
    return junction_tree
```



CONNESSIONE DELLE CLIQUES

Esempio (I)

STEP 1

Lista delle Clique = [ABC, BCE, BDE, CF, DG, DH, EI]

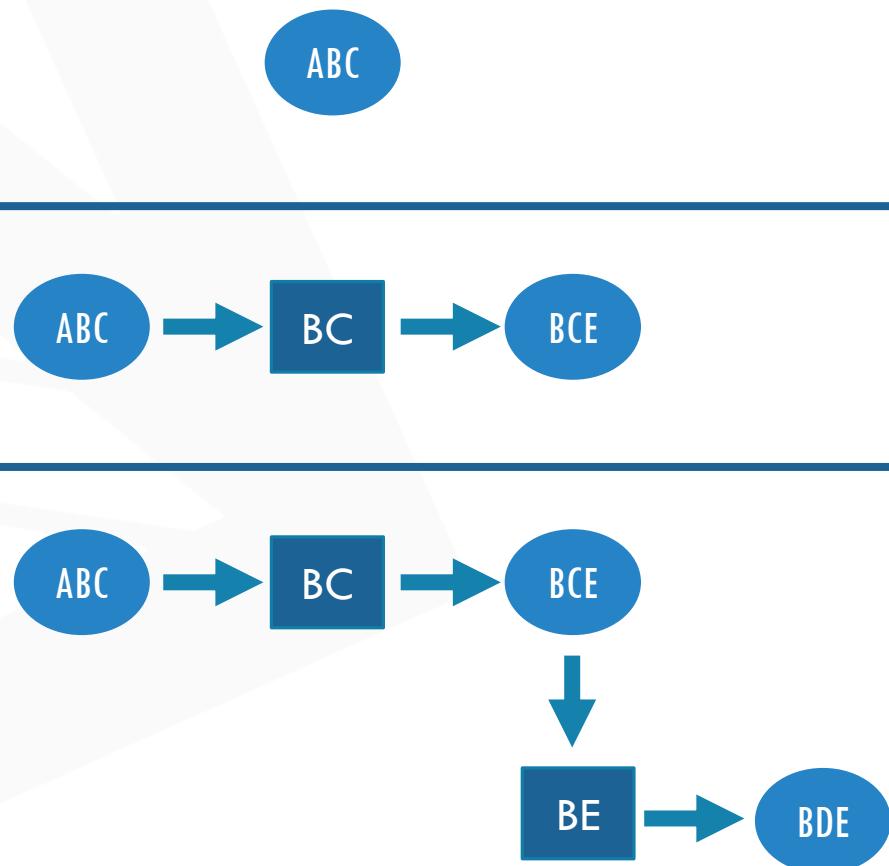
Inizialmente si sceglie la clique ABC e non avendo nessun nodo che lo precede nella lista verrà inserito da solo nel Junction Tree.

STEP 2

Si sceglie la clique BCE e avendo solo ABC che lo precede nella lista si procede a connettere i due nodi

STEP 3

Si sceglie la clique BDE, nella lista è preceduto da ABC e BCE. La clique con il quale condivide più nodi è BCE quindi verrà connessa con questa clique



CONNESSIONE DELLE CLIQUES

Esempio (II)

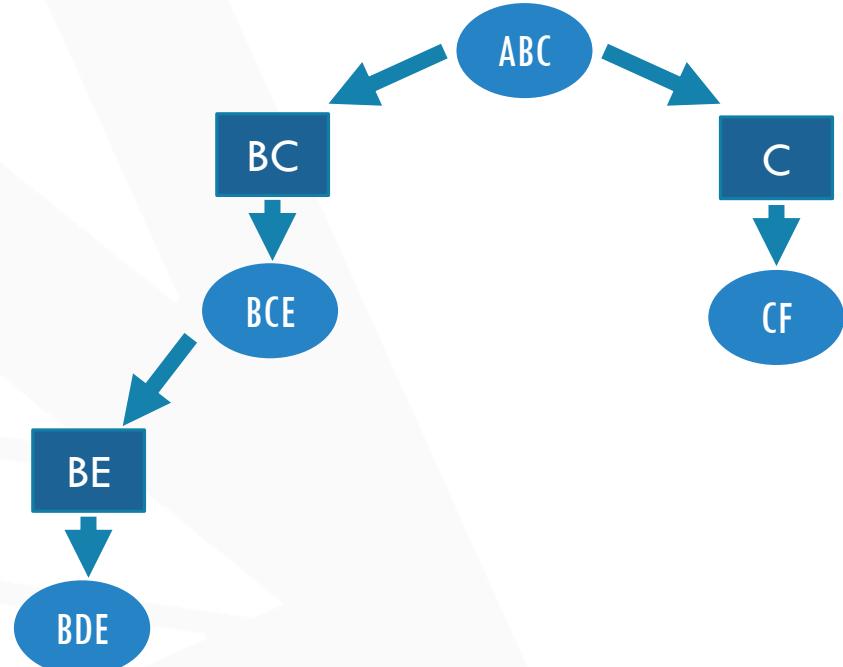
STEP 4

Lista delle Clique = [ABC, BCE, BDE, CF, DG, DH, EI]

Si sceglie la clique CF. Nella lista è preceduto da ABC, BCE, BDE. Che hanno in comune:

- ABC – 1 nodo
- BCE – 1 nodo
- BDE – 0 nodi

Viene scelta arbitrariamente la clique ABC a discapito della clique BCE



CONNESSIONE DELLE CLIQUES

Esempio (III)

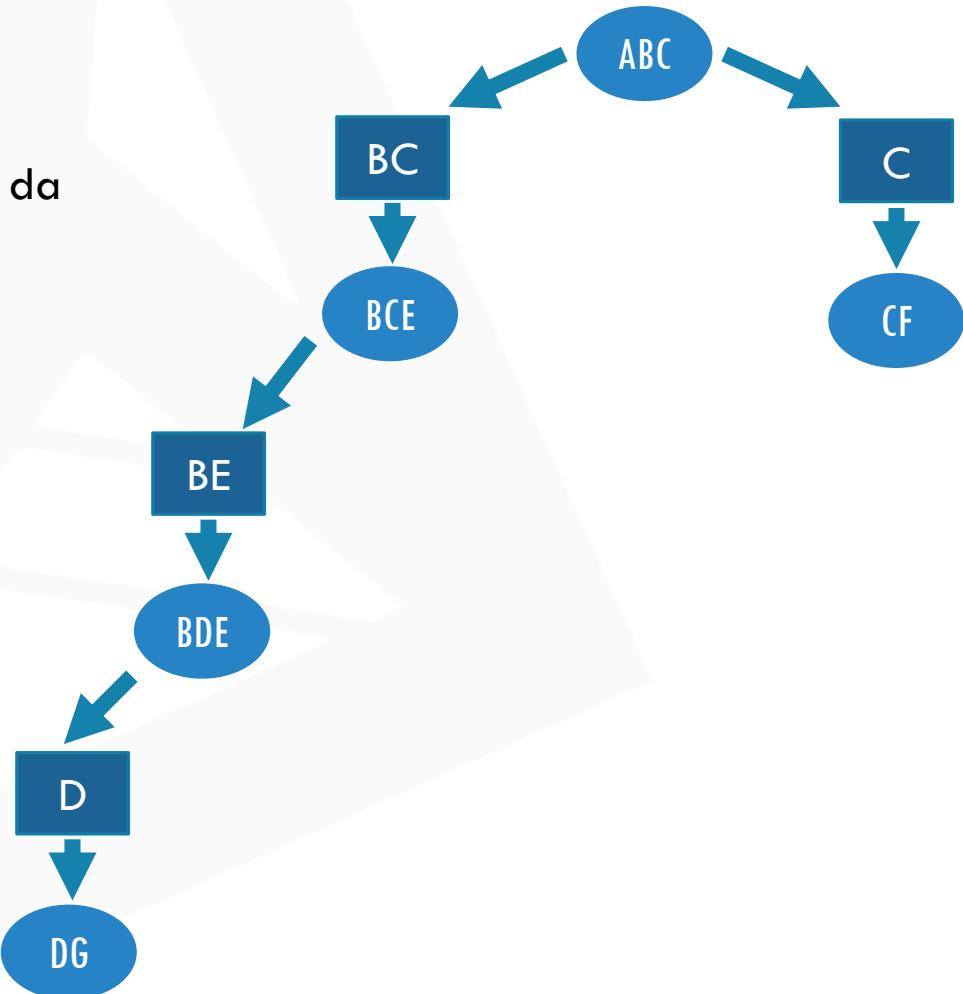
STEP 5

Lista delle Clique = [ABC, BCE, BDE, CF, DG, DH, EI]

Si sceglie la clique DG. Nella lista è preceduto da ABC, BCE, BDE, CF. Che hanno in comune:

- ABC – 0 nodi
- BCE – 0 nodi
- BDE – 1 nodo
- CF – 0 nodi

Viene scelta la clique BDE



CONNESSIONE DELLE CLIQUES

Esempio (IV)

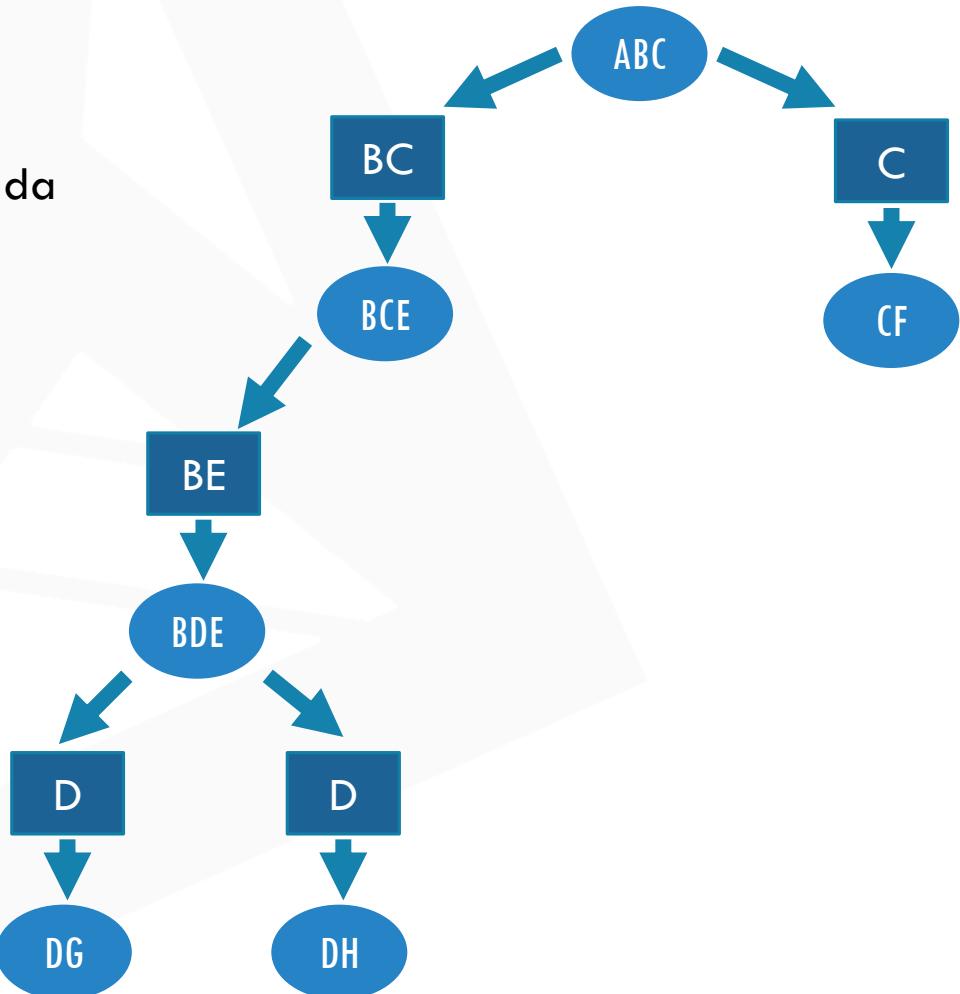
STEP 6

Lista delle Clique = [ABC, BCE, BDE, CF, DG, DH, EI]

Si sceglie la clique DH. Nella lista è preceduto da ABC, BCE, BDE, CF, DG. Che hanno in comune:

- ABC – 0 nodi
- BCE – 0 nodi
- BDE – 1 nodo
- CF – 0 nodi
- DG – 1 nodo

Viene scelta arbitrariamente la clique BDE



CONNESSIONE DELLE CLIQUES

Esempio (V)

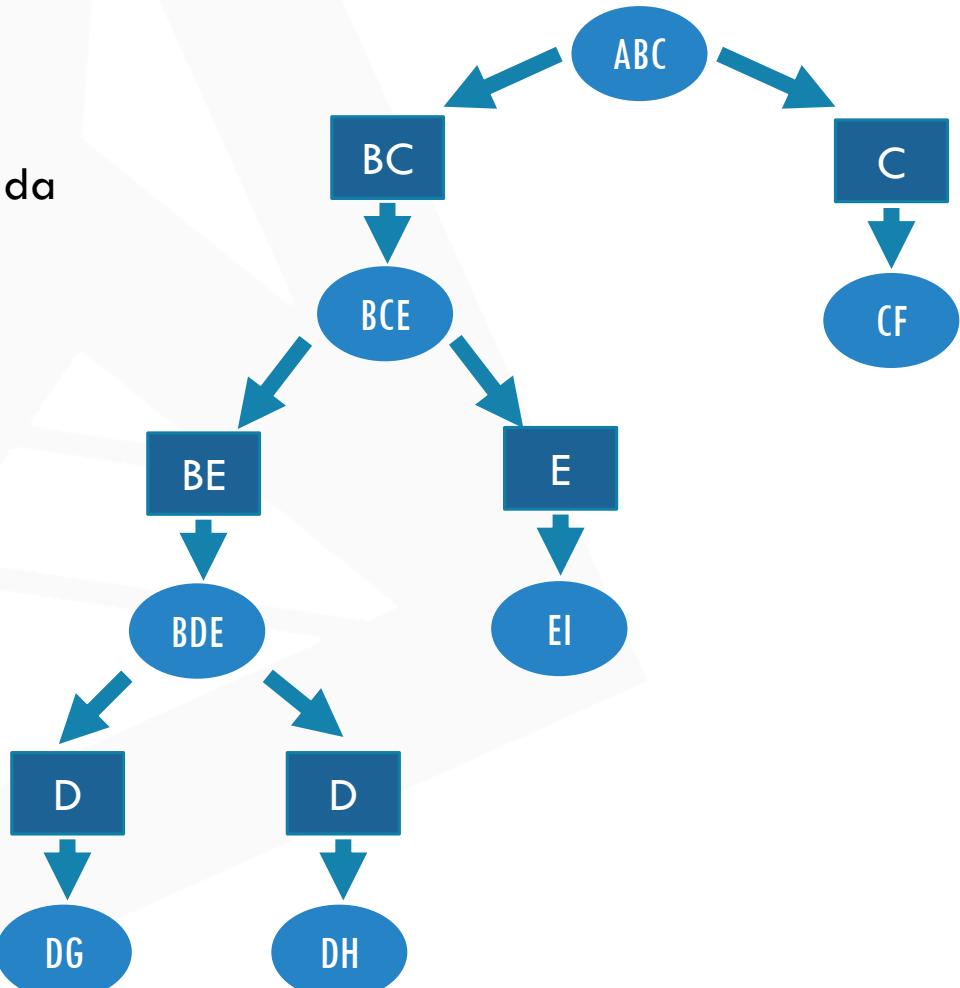
STEP 7

Lista delle Clique = [ABC, BCE, BDE, CF, DG, DH, EI]

Si sceglie la clique EI. Nella lista è preceduto da ABC, BCE, BDE, CF, DG, DH. Che hanno in comune:

- ABC – 0 nodi
- BCE – 1 nodo
- BDE – 1 nodo
- CF – 0 nodi
- DG – 0 nodi
- DH – 0 nodi

Viene scelta arbitrariamente la clique CBE



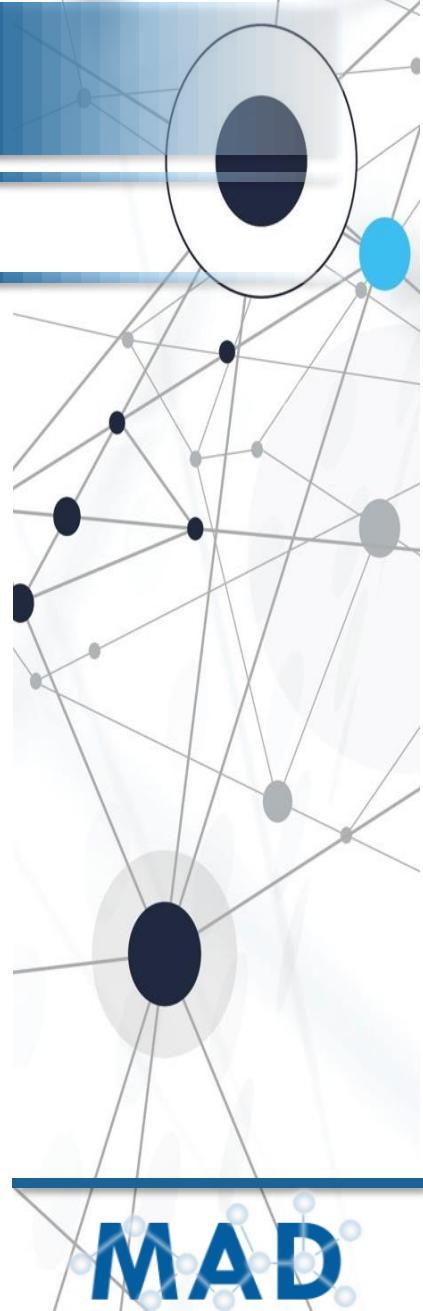
CONNESSIONE DELLE CLIQUES

Struttura dei nodi del Junction Tree

Un nodo generico della struttura Tree implementata nella libreria creata consente di settare il suo *parent* e aggiungere un nodo figlio per volta o una lista di nodi figli. Gli attributi del nodo vengono salvati in un dizionario lasciando quindi completa libertà sui valori da salvare al suo interno.

Nello specifico, i nodi del Junction Tree, sia cliques che separatori, vengono salvati con i seguenti attributi:

- ‘**label**’ = stringa che contiene i nodi separati da una virgola e tra parentesi graffe, es. {a,b,c}
- ‘**type**’ = tipo di nodo, per distinguere tra ‘clique’ e ‘separator’
- ‘**nodes**’ = la lista dei nodi che forma la clique
- ‘**clique**’ = il sottografo del DAG della BN contenente i nodi della clique con gli archi che li collegano
- ‘**parent**’ = il reference al nodo parent nel Junction Tree
- ‘**children**’ = la lista dei nodi figli nel Junction Tree



CALCOLO DEI BELIEFS

Variable Elimination

Junction Tree Calibration

Message passing

MAD

VARIABLE ELIMINATION

Teoria

Supponiamo di voler conoscere $P(X_1)$, per ottenere la probabilità di questa variabile dovremo calcolare:

$$P(X_1) = \sum_{X_2} \sum_{X_3} \dots \sum_{X_n} P(X) = \sum_{X_2} \sum_{X_3} \dots \sum_{X_n} \prod_i P(X_i | \Pi_{X_i})$$

Dove Π_{X_i} sono i parent di X_i nel DAG.

È semplice vedere che la maggioranza dei prodotti verranno calcolati moltissime volte ottenendo gli stessi risultati con uno spreco di risorse computazionali non indifferente.

Idea dell'algoritmo VE è quindi quella di spostare verso le somme più esterne i prodotti che è possibile **non** eseguire nelle somme più interne. Il prodotto delle somme interne verrà quindi sostituito con fattori che verranno usati poi nelle somme più esterne



VARIABLE ELIMINATION

Esempio: Asia network (I)

Siamo interessati a conoscere $P(D)$ e dovremo quindi «eliminare» tutte le altre variabili (A, S, X, T, L, E, B)

Le *Conditional Probability Distribution* (CPD) iniziali, o fattori iniziali, sono:

$$P(A) \cdot P(T|A) \cdot P(S) \cdot P(L|S) \cdot P(B|S) \cdot P(E|T,L) \cdot P(X|E) \cdot P(D|B,E)$$

L'approccio a forza bruta consiste quindi nel calcolare:

$$P(D) = \sum_a \sum_s \sum_x \sum_t \sum_l \sum_e \sum_b P(A) \cdot P(T|A) \cdot P(S) \cdot P(L|S) \cdot P(B|S) \cdot P(E|T,L) \cdot P(X|E) \cdot P(D|B,E)$$

Tuttavia con l'approccio di Variable Elimination è possibile evitare moltissimi calcoli. Usiamo come ordine di eliminazione delle variabili quello presente sopra. Quindi:

$$A \rightarrow S \rightarrow X \rightarrow T \rightarrow L \rightarrow E \rightarrow B$$

MAD

VARIABLE ELIMINATION

Esempio: Asia network (II)

$$A \rightarrow S \rightarrow X \rightarrow T \rightarrow L \rightarrow E \rightarrow B$$

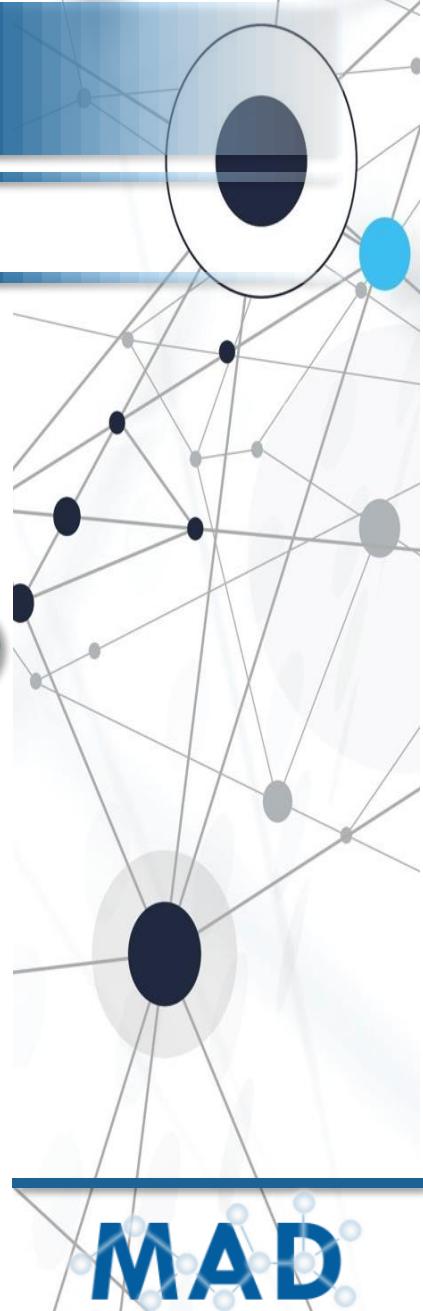
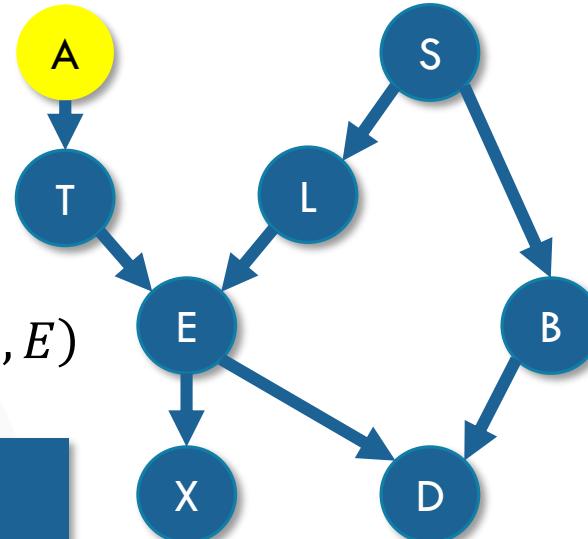
Fattori iniziali

$$P(A) \cdot P(T|A) \cdot P(S) \cdot P(L|S) \cdot P(B|S) \cdot P(E|T,L) \cdot P(X|E) \cdot P(D|B,E)$$

$$f_A(T) = \sum_a P(a) \cdot P(T|a)$$

Combinando i fattori rispetto ad A otteniamo i nuovi fattori dai quale spariscono quelli utilizzati e compare il nuovo fattore calcolato:

$$f_A(T) \cdot P(S) \cdot P(L|S) \cdot P(B|S) \cdot P(E|T,L) \cdot P(X|E) \cdot P(D|B,E)$$



MAD

VARIABLE ELIMINATION

Esempio: Asia network (III)

$$A \rightarrow S \rightarrow X \rightarrow T \rightarrow L \rightarrow E \rightarrow B$$

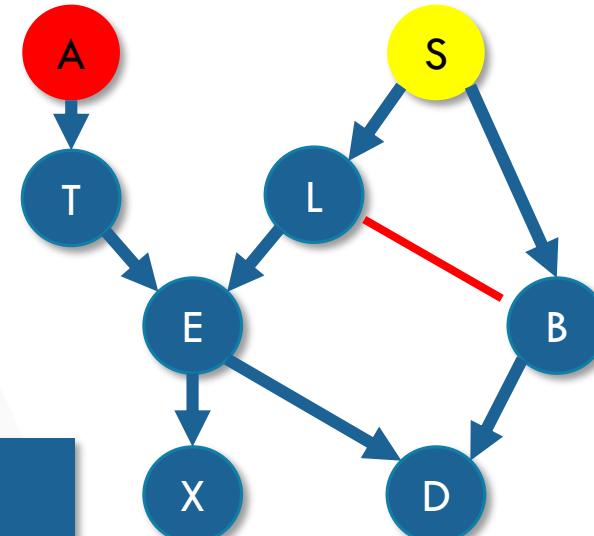
Fattori

$$f_A(T) \cdot P(S) \cdot P(L|S) \cdot P(B|S) \cdot P(E|T,L) \cdot P(X|E) \cdot P(D|B,E)$$

$$f_S(B, L) = \sum_s P(s) \cdot P(L|s) \cdot P(B|s)$$

Combinando i fattori rispetto ad S otteniamo:

$$f_A(T) \cdot f_S(B, L) \cdot P(E|T,L) \cdot P(X|E) \cdot P(D|B,E)$$



Nota

Come risultato
dell'eliminazione i nodi L
e B vengono connessi

MAD

VARIABLE ELIMINATION

Esempio: Asia network (IV)

$$A \rightarrow S \rightarrow X \rightarrow T \rightarrow L \rightarrow E \rightarrow B$$

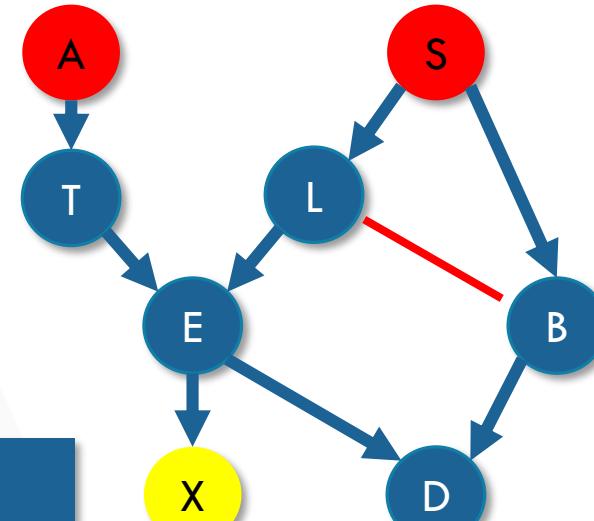
Fattori

$$f_A(T) \cdot f_S(B, L) \cdot P(E|T, L) \cdot P(X|E) \cdot P(D|B, E)$$

$$f_X(E) = \sum_x P(x|E)$$

Combinando i fattori rispetto ad X otteniamo:

$$f_A(T) \cdot f_S(B, L) \cdot f_X(E) \cdot P(E|T, L) \cdot P(D|B, E)$$



VARIABLE ELIMINATION

Esempio: Asia network (V)

$$A \rightarrow S \rightarrow X \rightarrow T \rightarrow L \rightarrow E \rightarrow B$$

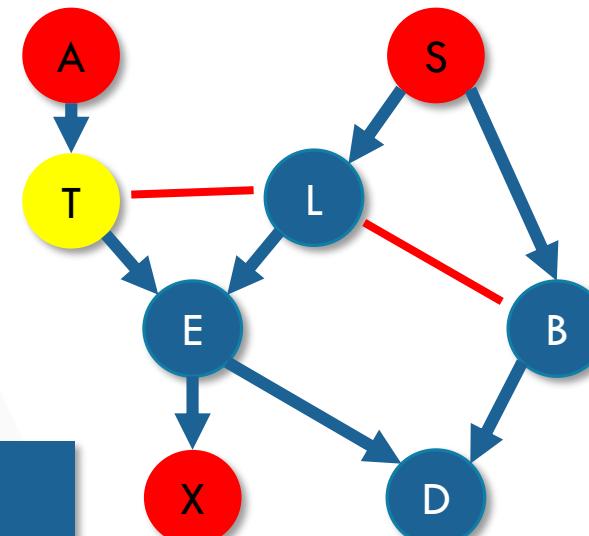
Fattori

$$f_A(T) \cdot f_S(B, L) \cdot f_X(E) \cdot P(E|T, L) \cdot P(D|B, E)$$

$$f_T(E, L) = \sum_t f_A(t) \cdot P(E|t, L)$$

Combinando i fattori rispetto ad T otteniamo:

$$f_S(B, L) \cdot f_X(E) \cdot f_T(E, L) \cdot P(D|B, E)$$



Nota

Come risultato
dell'eliminazione i nodi L
e T vengono connessi

MAD

VARIABLE ELIMINATION

Esempio: Asia network (VI)

$$A \rightarrow S \rightarrow X \rightarrow T \rightarrow L \rightarrow E \rightarrow B$$

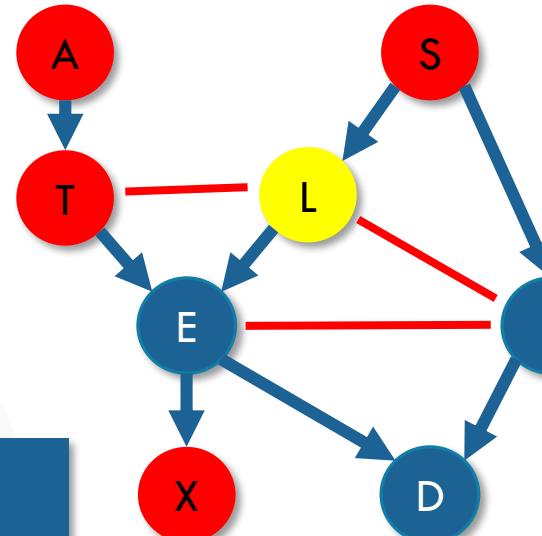
Fattori

$$f_S(B, L) \cdot f_X(E) \cdot f_T(E, L) \cdot P(D|B, E)$$

$$f_L(B, E) = \sum_l f_S(B, l) \cdot f_T(E, l)$$

Combinando i fattori rispetto ad L otteniamo:

$$f_X(E) \cdot f_L(B, E) \cdot P(D|B, E)$$



Nota

Come risultato
dell'eliminazione i nodi E
e B vengono connessi

VARIABLE ELIMINATION

Esempio: Asia network (VII)

$$A \rightarrow S \rightarrow X \rightarrow T \rightarrow L \rightarrow E \rightarrow B$$

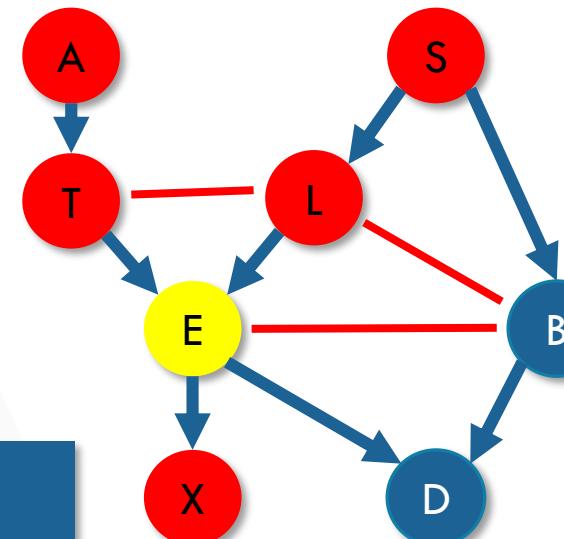
Fattori

$$f_X(E) \cdot f_L(B, E) \cdot P(D|B, E)$$

$$f_E(B, D) = \sum_e f_X(e) \cdot f_L(B, e) \cdot P(D|B, e)$$

Combinando i fattori rispetto ad E otteniamo:

$$f_E(B, D)$$



VARIABLE ELIMINATION

Esempio: Asia network (VIII)

$$A \rightarrow S \rightarrow X \rightarrow T \rightarrow L \rightarrow E \rightarrow B$$

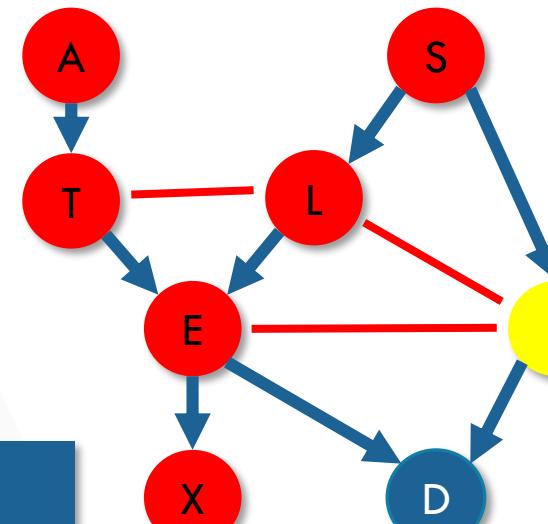
Fattori

$$f_E(B, D)$$

$$f_B(D) = \sum_b f_E(b, D)$$

Combinando i fattori rispetto ad B otteniamo:

$$f_B(D)$$



VARIABLE ELIMINATION

Sommario

Per eseguire l'algoritmo Variable Elimination è stato creato un ordinamento secondo cui vengono eliminate le variabili. Tuttavia questo è unico e vi sono più ordinamenti corretti, l'unica cosa che cambia è la computazione richiesta.

L'esecuzione di Variable Elimination su un DAG, seguendo un qualsiasi ordine di eliminazione delle variabili, crea una struttura ad albero, il **Junction Tree**, i cui nodi sono formati da gruppi di variabili così come vengono raggruppate nei fattori creati dall'algoritmo. Questo può essere sfruttato per introdurre un ulteriore miglioramento rispetto all'algoritmo base.

Infatti, nonostante Variable Elimination sia un miglioramento dell'algoritmo a forza bruta per ogni query questo deve essere ripetuto interamente. Considerando che una query abbia costo C per n query avremo un costo totale di $n \cdot c$. Questo tempo può essere notevolmente ridotto facendo ricorso a una struttura che fungendo da cache tenga memorizzati i calcoli eseguiti che saranno i medesimi per tutte le query.



VARIABLE ELIMINATION ON JUNCTION TREE

Assegnazione dei potenziali iniziali alle clique

Il metodo più semplice per utilizzare un Junction Tree(JT) consiste nell'eseguire l'algoritmo VE seguendo l'ordine dato dall'albero. Per fare ciò come prima cosa, supponendo di avere già la struttura del JT sarà quella di calcolare i potenziali di ogni clique.

Il calcolo del potenziale $\psi_i(C_i)$ della generica clique C_i viene fatto moltiplicando tra loro i **fattori (CPDs)** assegnati alla clique. Un CPD è assegnato ad una clique X quando le sue variabili sono un sottoinsieme delle variabili di X . Il potenziale della clique C_i sarà quindi calcolato come:

$$\psi_i(C_i) = \prod_{\phi: \alpha(\phi)=i} \phi$$

Dove $\phi \in \Phi$ è un CPD assegnato a una clique $\alpha(\phi)$.

Un CPD deve essere associato ad **una e una sola** clique anche se potrebbe essere associato a più di una. È tuttavia ininfluente a quale venga assegnata. Il fatto che tutte le CPDs siano associabili ad almeno una clique è garantito dalla proprietà di **family preservation** di cui un JT valido deve godere.



VARIABLE ELIMINATION ON JUNCTION TREE

Message passing (I)

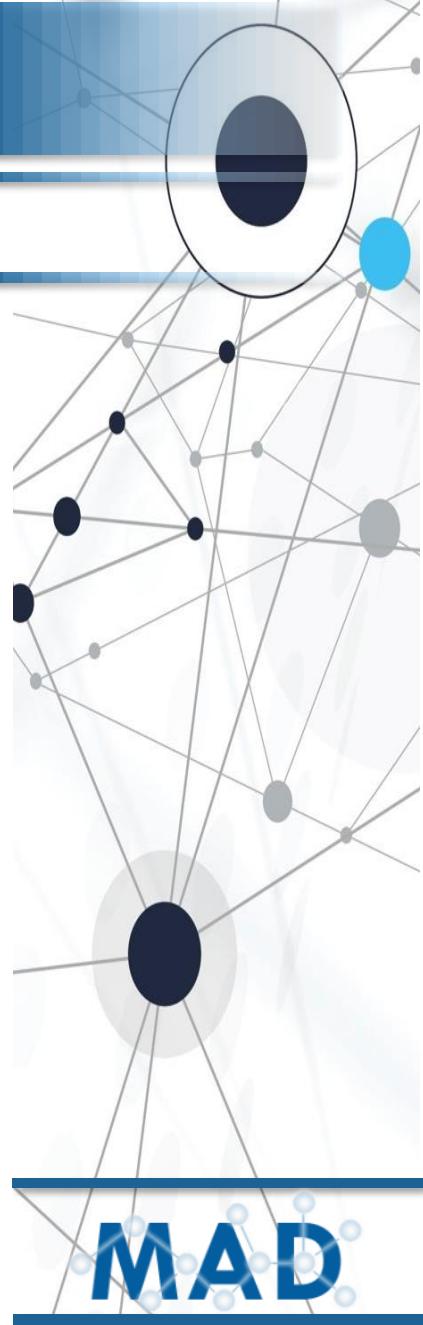
Date due clique generiche C_i e C_j di un JT. Se C_i è sul cammino tra C_j e la radice del JT diremo che:

- C_i è **upstream** rispetto a C_j
- C_j è **downstream** rispetto a C_i

Per conoscere la probabilità $P(A)$ di una variabile A è necessario trovare una clique del JT che contenga tale variabile, questa fungerà da nodo radice durante il calcolo di **questa query**. A questo punto, partendo dalle clique che si trovano come foglie rispetto alla radice scelta, viene eseguito l'algoritmo di **message passing** verso l'alto fino alla radice.

Message passing

Secondo l'idea di base dell'algoritmo di **message passing** ogni clique, una volta ricevuti **tutti** i messaggi dalle cliques *downstream* adiacenti, moltiplica tutti i messaggi con il potenziale iniziale della clique. A questo punto deve passare a sua volta il suo messaggio alla clique *upstream*. Questo è ottenuto marginalizzando il prodotto calcolato rispetto alle variabili che **non** sono presenti nel separatore in comune tra le due clique.



VARIABLE ELIMINATION ON JUNCTION TREE

Message passing (II)

Più formalmente il messaggio ($\delta_{i \rightarrow j}$) della clique C_i verso la clique *upstream* C_j è calcolato come:

$$\delta_{i \rightarrow j} = \sum_{C_i - S_{i,j}} \left(\psi_i \cdot \prod_{k \in (Nb_i - \{j\})} \delta_{k \rightarrow i} \right)$$

Dove $S_{i,j}$ è il set di variabili comuni alle clique C_i e C_j . La sommatoria è l'operazione di marginalizzazione, che «elimina» le variabili che **non** sono presenti nella clique C_j e che non saranno più presenti nel JT.

Ciò che garantisce che si possa eliminare definitivamente una variabile è il fatto che un Junction Tree soddisfi la proprietà di **Running Intersection Property**.

Arrivati alla radice questa può calcolare il suo fattore che viene chiamato **belief** $\beta_r(C_r)$ dato da:

$$\beta_r = \psi_r \cdot \prod_{k \in Nb_r} \delta_{k \rightarrow r}$$

Dal **belief** del nodo radice è quindi possibile estrarre $P(A)$ marginalizzando rispetto a tutte le altre variabili presenti in $\beta_r(C_r)$, ottenendo al termine solo la probabilità chiesta.



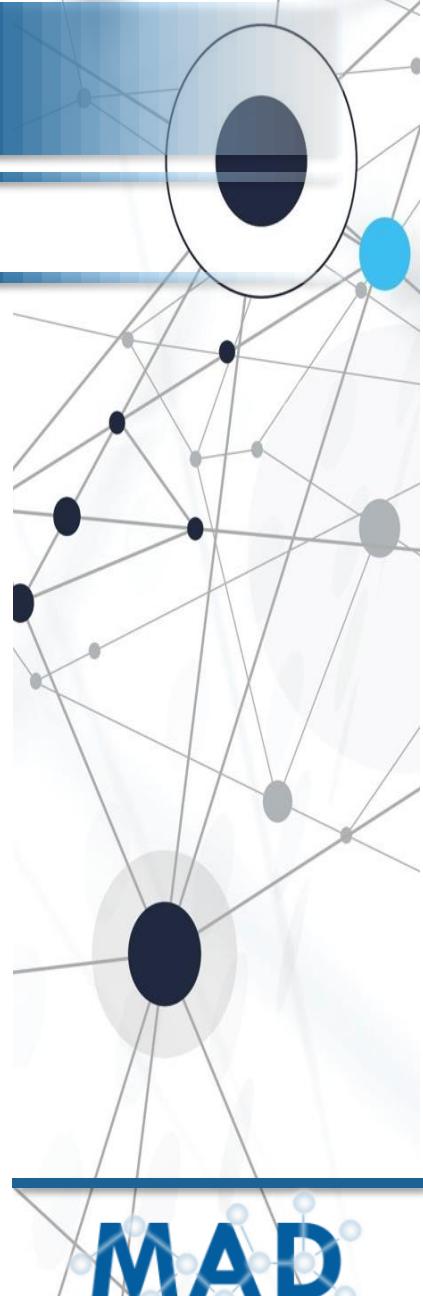
VARIABLE ELIMINATION ON JUNCTION TREE

Sommario

In questa versione dell'algoritmo sul Junction Tree i potenziali iniziali non vanno modificati. Questo per non perdere le informazioni iniziali poiché per ogni singola query andrà eseguito l'algoritmo su tutto il JT.

Anche in questo caso, considerando ancora il tempo di esecuzione di una query pari a c , il tempo per eseguire n query sarà pari a $n \cdot c$. L'unica modifica introdotta è che non è stato seguito un ordine di eliminazione delle variabili ma si è seguito l'ordine dato dalla struttura del Junction Tree.

Tuttavia per migliorare il tempo di esecuzione in modo considerevole ed avere un preprocessing di tutti i valori necessari e salvarli nel Junction Tree saranno necessarie poche aggiunte mantenendo intatta l'idea generica appena introdotta.



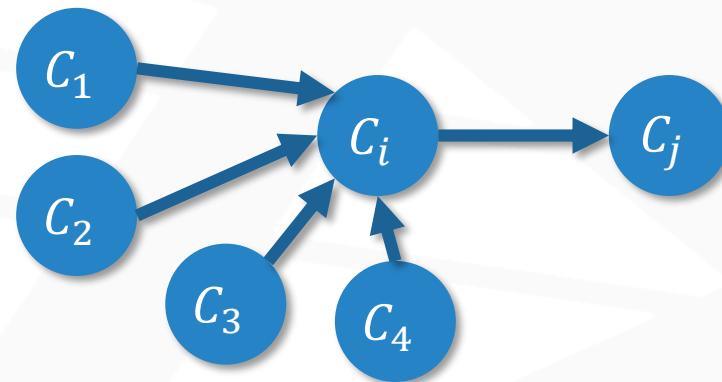
MAD

JUNCTION TREE CALIBRATION

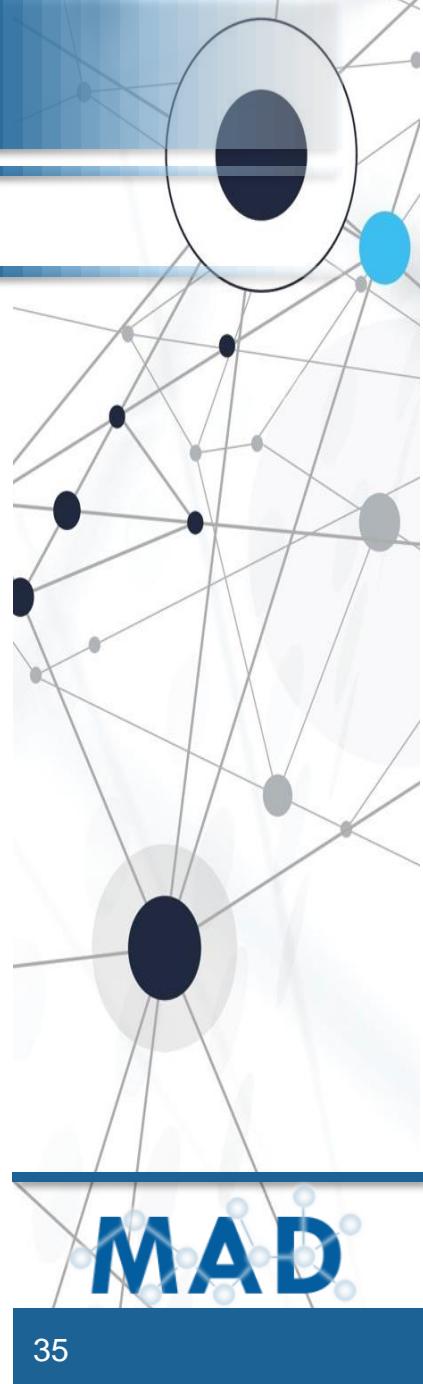
Messaggi da calcolare

Finora per ogni nuova query si sono ricalcolati tutti i messaggi passati tra le varie cliques. Tuttavia, considerando due generiche clique C_i e C_j adiacenti possiamo notare che il valore dei messaggi che queste si scambiano, $\delta_{i \rightarrow j}$ e $\delta_{j \rightarrow i}$, rimane lo stesso indipendentemente dal nodo radice scelta.

Quindi sarà sufficiente calcolare il valore di questi una sola volta e poi riutilizzarli indipendentemente dalla query. Per un JT di n cliques, e quindi di $n - 1$ archi, si possono calcolare solamente $2(n - 1)$ messaggi.



Per adattarsi a ciò ridefiniamo il concetto di *ready clique*, cioè quando una clique è pronta a passare il suo messaggio, per astrarla dalla presenza di un nodo scelto come radice. Una clique C_i è detta **ready to transmit** verso la clique C_j quando C_i ha tutti i messaggi dei suoi vicini tranne quello della clique C_j .

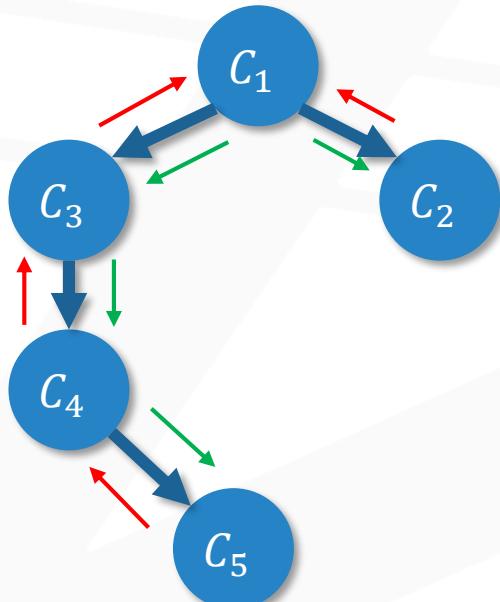


JUNCTION TREE CALIBRATION

Calcolo dei belief

Per far sì che ogni clique “raccolga” **tutti** i messaggi dei suoi vicini sarà necessario eseguire un doppio passaggio attraverso tutto l’albero, uno *upward* e uno *downward* rispetto a una radice arbitraria. Terminata la fase in cui vengono raccolti i messaggi, per ogni clique C_i viene calcolato il **belief** definito come:

$$\beta_i(C_i) = \psi_i \cdot \prod_{k \in Nb_i} \delta_{k \rightarrow i}$$



→ **Fase 1:** fase upward

→ **Fase 2:** fase downward

L’unica clique che può calcolare il suo belief dopo un solo passaggio dei messaggi è la radice del JT come è stato fatto nella versione precedente dell’algoritmo.

Prendiamo come esempio la clique C_3 . Eseguendo solo una fase *upward* non potrebbe calcolare il suo belief in quanto gli mancherebbe il messaggio $\delta_{1,3}$.



JUNCTION TREE CALIBRATION

Definizione di Junction Tree calibrato

Due clique adiacenti C_i e C_j sono dette **calibrate** se le variabili in comune hanno le stesse probabilità in entrambe. In altre parole, marginalizzando sulle variabili che **non** sono presenti nel separatore abbiamo:

$$\sum_{C_i - S_{i,j}} \beta_i(C_i) = \sum_{C_j - S_{i,j}} \beta_j(C_j)$$

Diremo quindi che un **Junction Tree** è detto calibrato se ogni coppia di clique adiacenti è calibrata. In questo caso $\beta_i(C_i)$ è detto **clique belief** e $\mu_{i,j}(S_{i,j})$ è detto **sepset belief**. Quest'ultimo è definito come:

$$\mu_{i,j}(S_{i,j}) = \delta_{j \rightarrow i} \cdot \delta_{i \rightarrow j} = \sum_{C_i - S_{i,j}} \beta_i(C_i) = \sum_{C_j - S_{i,j}} \beta_j(C_j)$$

Possiamo vedere un Junction Tree come una rappresentazione alternativa della probabilità congiunta che ci rivela in modo diretto le probabilità marginali delle cliques. Altro vantaggio dell'algoritmo è quello di calcolare queste probabilità con un costo pari a $2 \cdot c$ anziché un costo di $n \cdot c$ come in precedenza.

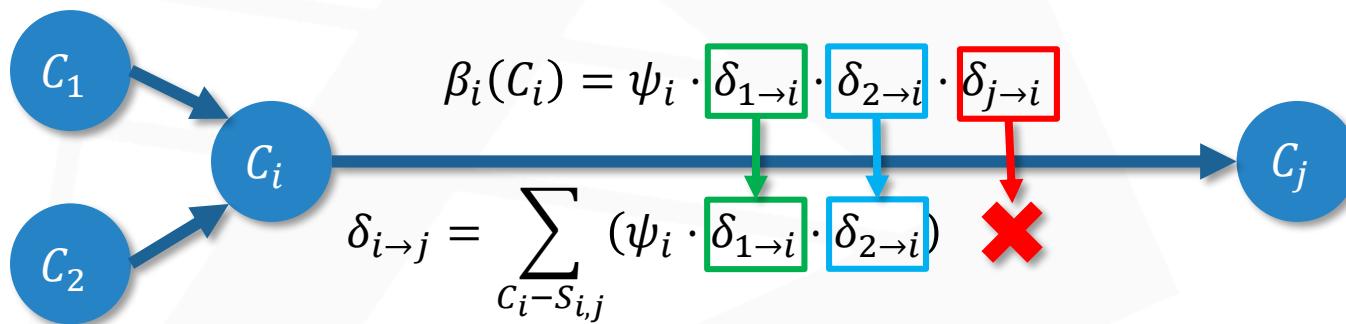


MESSAGE PASSING: BELIEF UPDATE WITH DIVISION

Problema: belief e calcolo dei messaggi

Nonostante sia stato calcolato il **belief** di una clique, non possiamo eliminare il potenziale iniziale e tutti i messaggi ricevuti. Questo finché non vengono calcolati i **belief** di **tutte** le cliques e quindi una volta terminata la fase di calibrazione del Junction Tree.

Come visto il messaggio $\delta_{i \rightarrow j}$ viene calcolato moltiplicando il potenziale iniziale della clique C_i e i messaggi ricevuti dalle clique adiacenti tranne quella dalla clique C_j . Il **belief** di una clique invece incorpora già l'informazione passata da C_j che verrebbe quindi contata due volte.



È possibile mantenere solamente i **belief**, una volta che una clique ha potuto calcolare il suo, e quindi dimenticare il resto delle informazioni introducendo una semplice modifica. Il messaggio verrà calcolato semplicemente **dividendo** il **belief** per il messaggio $\delta_{j \rightarrow i}$ prima di passarlo a C_j .



MESSAGE PASSING: BELIEF UPDATE WITH DIVISION

Divisione tra fattori

È quindi necessario dare una definizione di **divisione tra due fattori**. Siano X e Y due set disgiunti di variabili e siano $\phi_1(X, Y)$ e $\phi_2(Y)$ due fattori definiamo la divisione $\frac{\phi_1}{\phi_2}$ come un fattore ψ con **scope** X, Y .

Più precisamente:

$$\psi(X, Y) = \frac{\phi_1(X, Y)}{\phi_2(Y)} \quad \text{dove definiamo } 0/0 = 0$$

La divisione tra fattori è fatta componente per componente come tutte le altre operazioni tra fattori.



MESSAGE PASSING: BELIEF UPDATE WITH DIVISION

Calcolo del belief

Possiamo quindi calcolare $\delta_{i \rightarrow j}$ come:

$$\delta_{i \rightarrow j} = \frac{\sum_{C_i - S_{i,j}} \beta_i}{\delta_{j \rightarrow i}}$$

Notiamo che $\delta_{j \rightarrow i}$ è presente **da solo** nel **sepset belief** di $S_{i,j}$. Ricordiamo che questo ha valore:

$$\mu_{i,j}(S_{i,j}) = \delta_{j \rightarrow i} \cdot \delta_{i \rightarrow j}$$

Tuttavia, poiché stiamo calcolando in questo momento il valore di $\delta_{i \rightarrow j}$ il **sepset belief** avrà solamente il valore che ci interessa, cioè $\delta_{j \rightarrow i}$. Per avere questo vantaggio basterà aggiornare il **sepset belief** nel momento in cui un messaggio viene passato da C_i a C_j o viceversa senza attendere di averli entrambi.

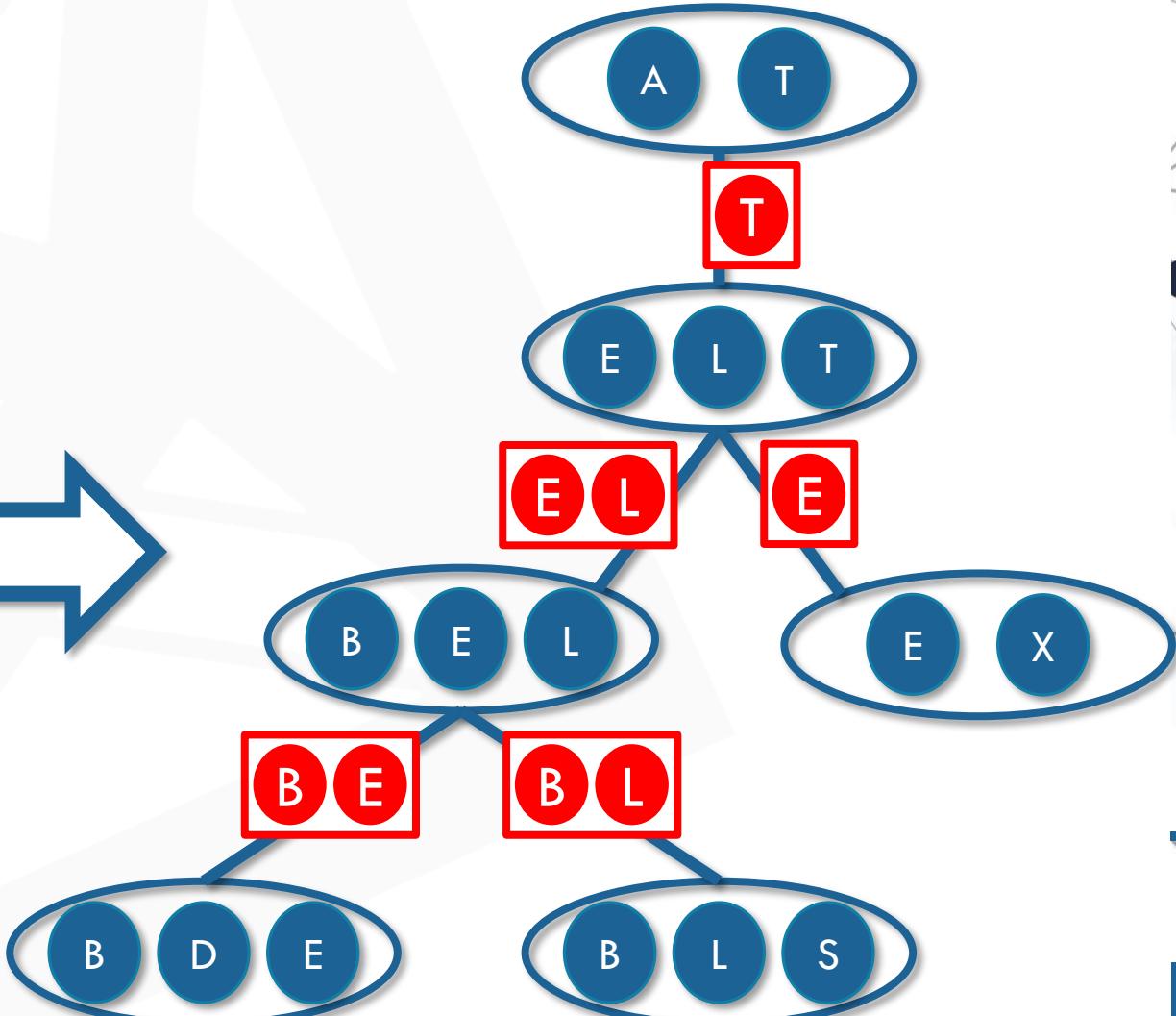
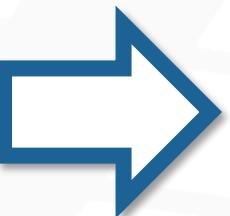
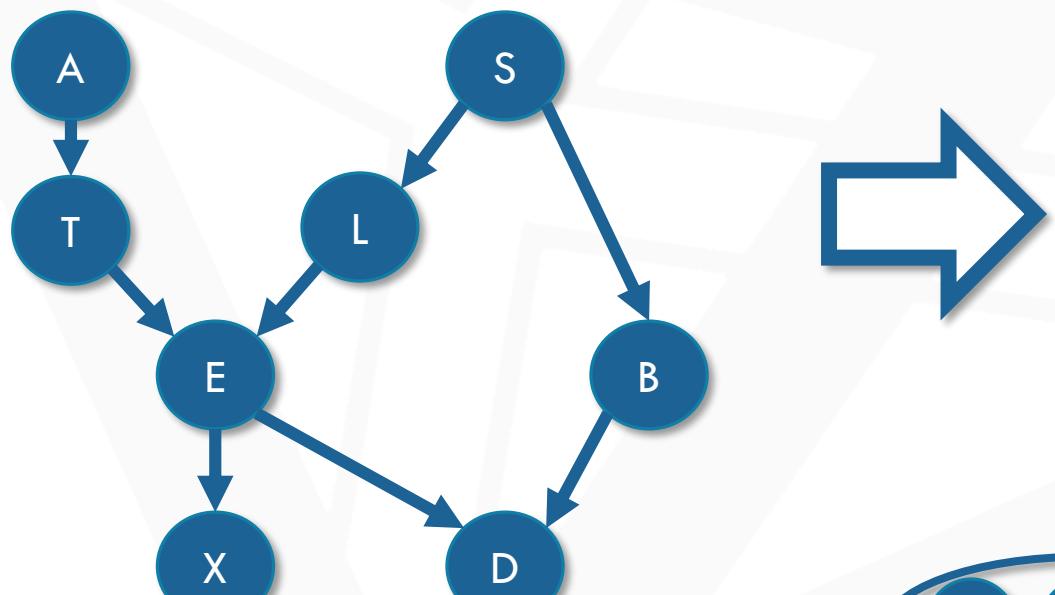
Ora ogni clique può evitare di mantenere al suo interno il valore del potenziale iniziale e tutti i messaggi ricevuti una volta calcolato il **clique belief** β_i potendo quindi risparmiare una grande quantità di spazio.



ESEMPIO FINALE: ASIA NETWORK

Dal DAG al Junction Tree

- Tramite:**
- Moralizzazione
 - Triangolazione
 - Ricerca delle cliques
 - Connessione delle cliques

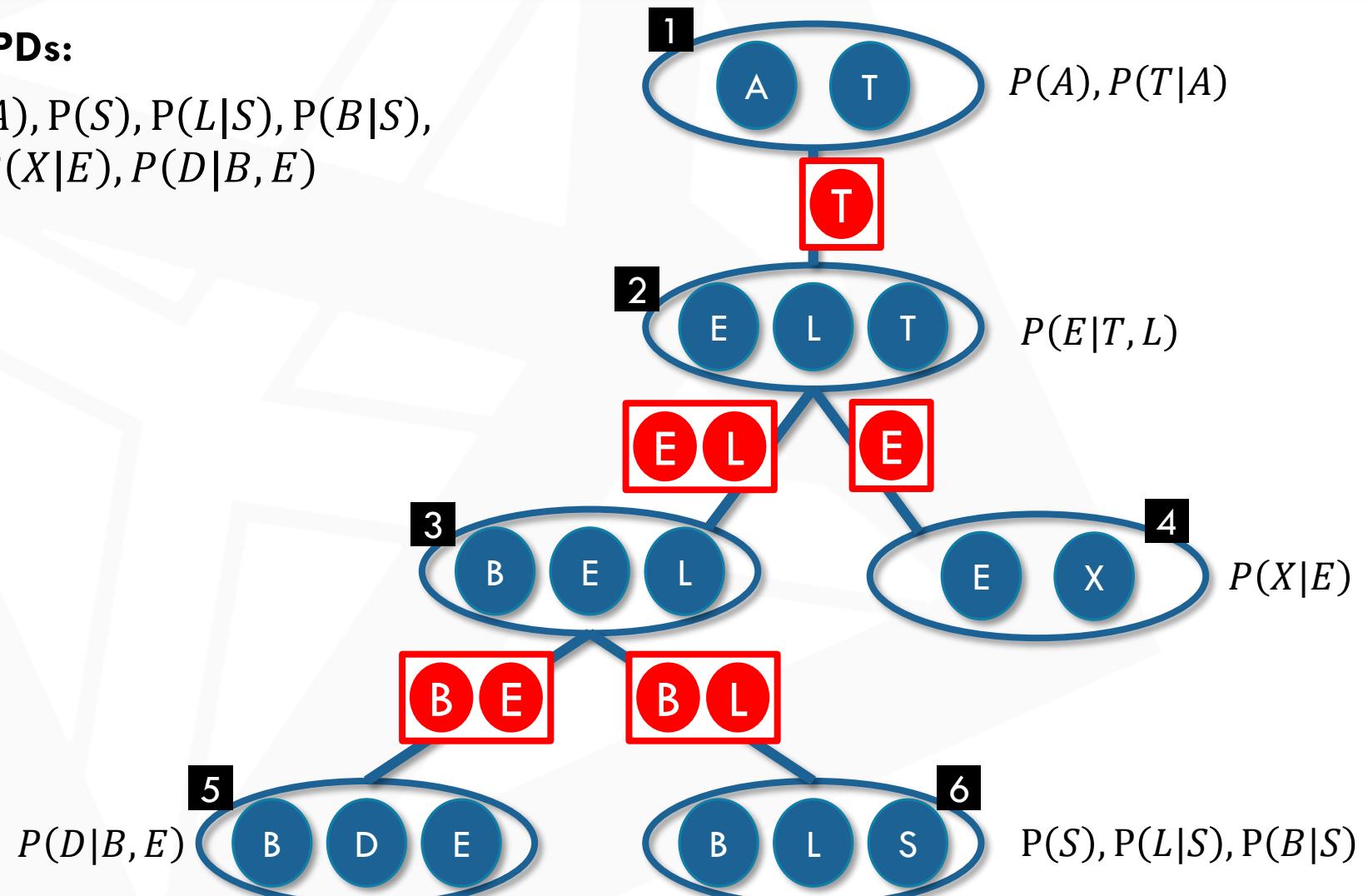


ESEMPIO FINALE: ASIA NETWORK

Assegnazione dei CPDs alle cliques

Lista dei CPDs:

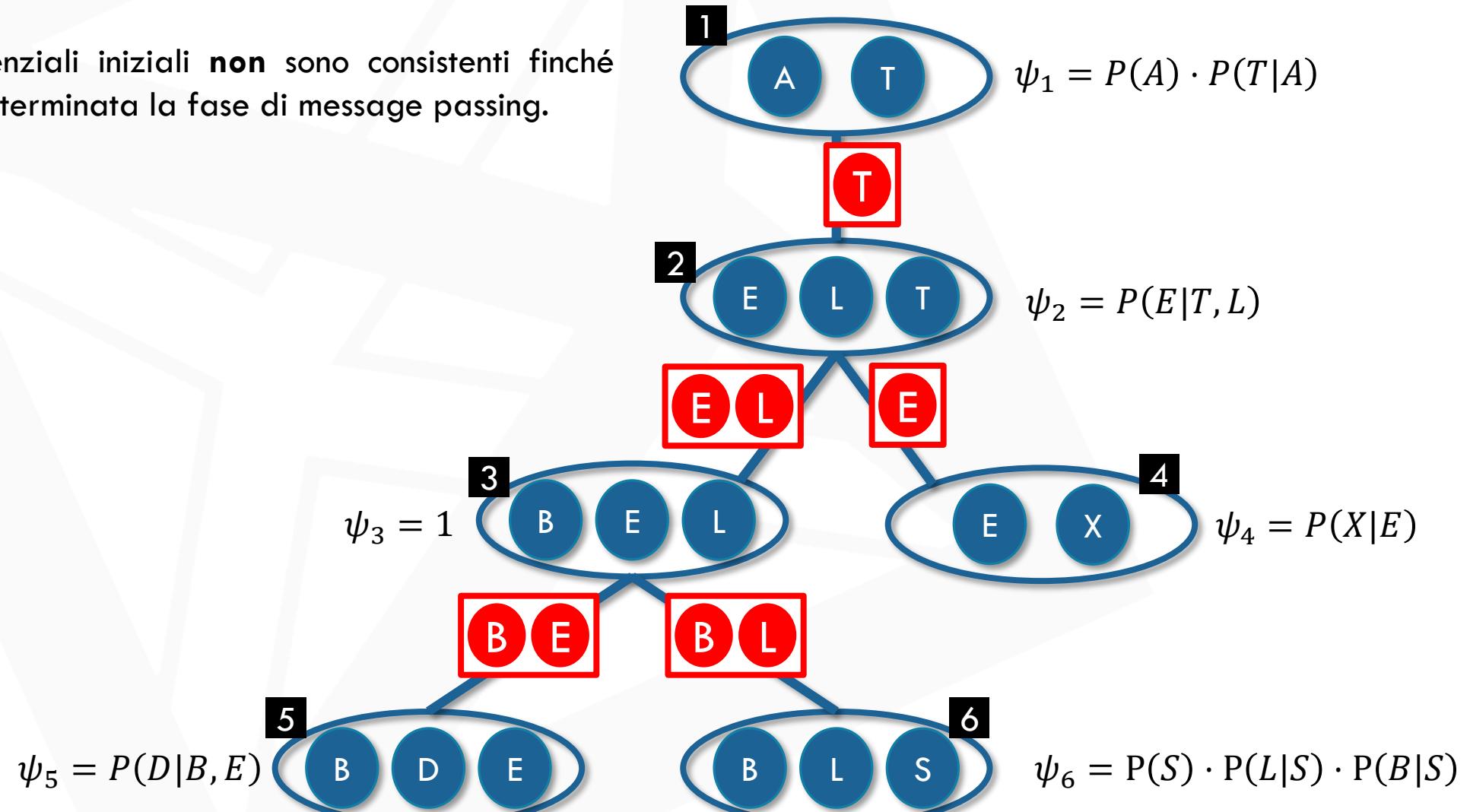
$P(A), P(T|A), P(S), P(L|S), P(B|S),$
 $P(E|T,L), P(X|E), P(D|B,E)$



ESEMPIO FINALE: ASIA NETWORK

Calcolo dei potenziali iniziali

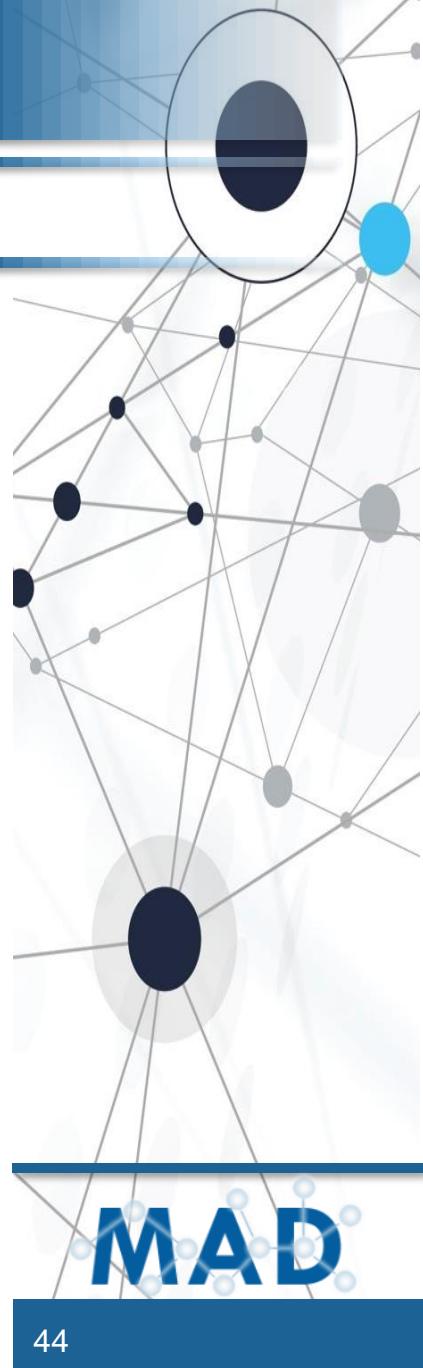
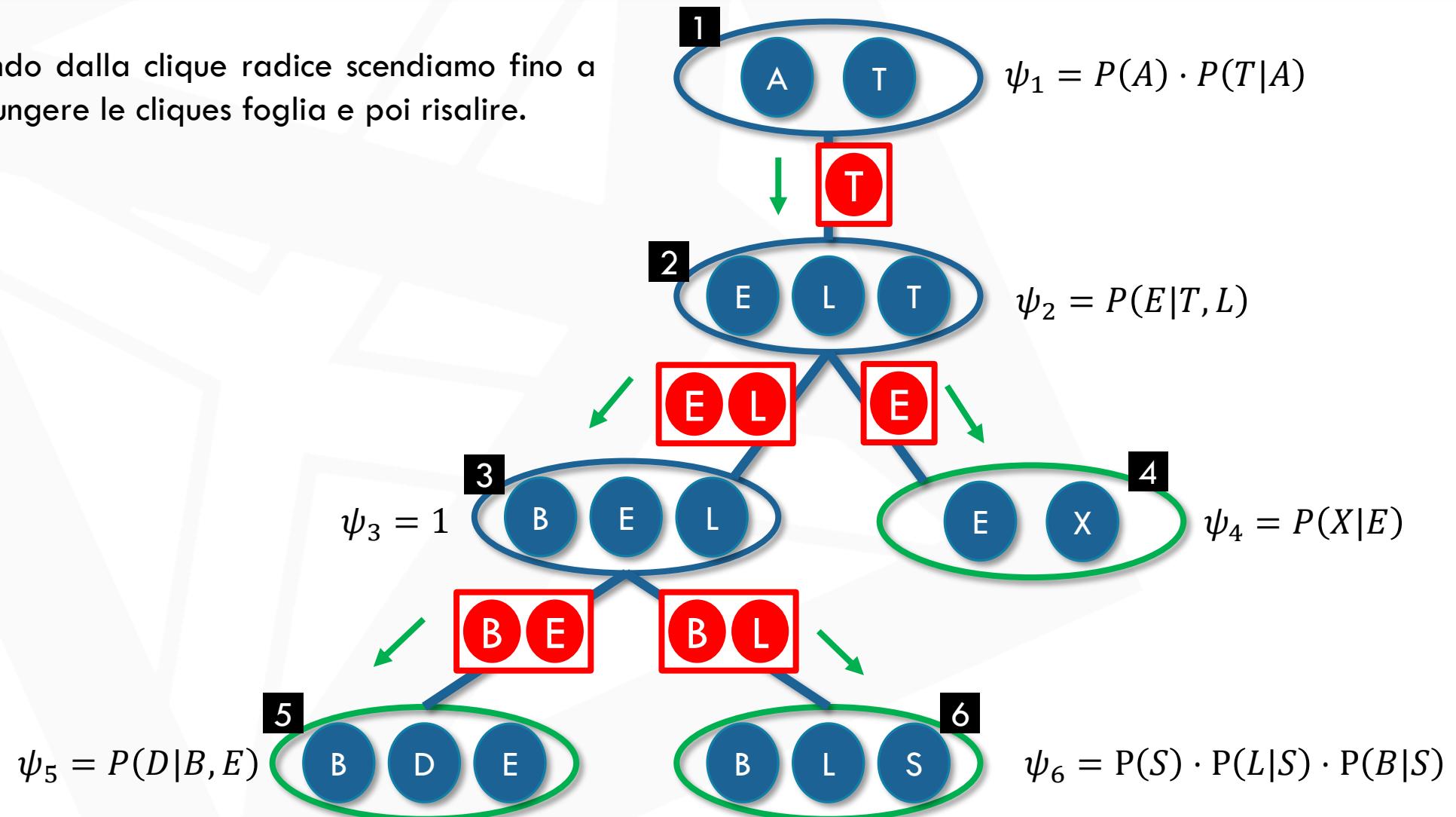
I potenziali iniziali **non** sono consistenti finché non è terminata la fase di message passing.



ESEMPIO FINALE: ASIA NETWORK

Fase upward (I)

Partendo dalla clique radice scendiamo fino a raggiungere le cliques foglia e poi risalire.



ESEMPIO FINALE: ASIA NETWORK

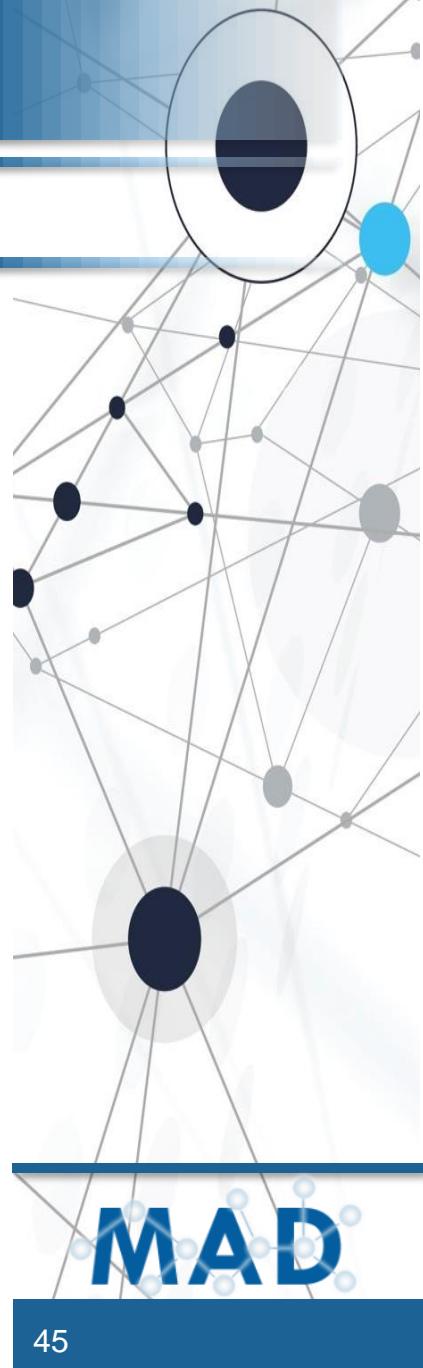
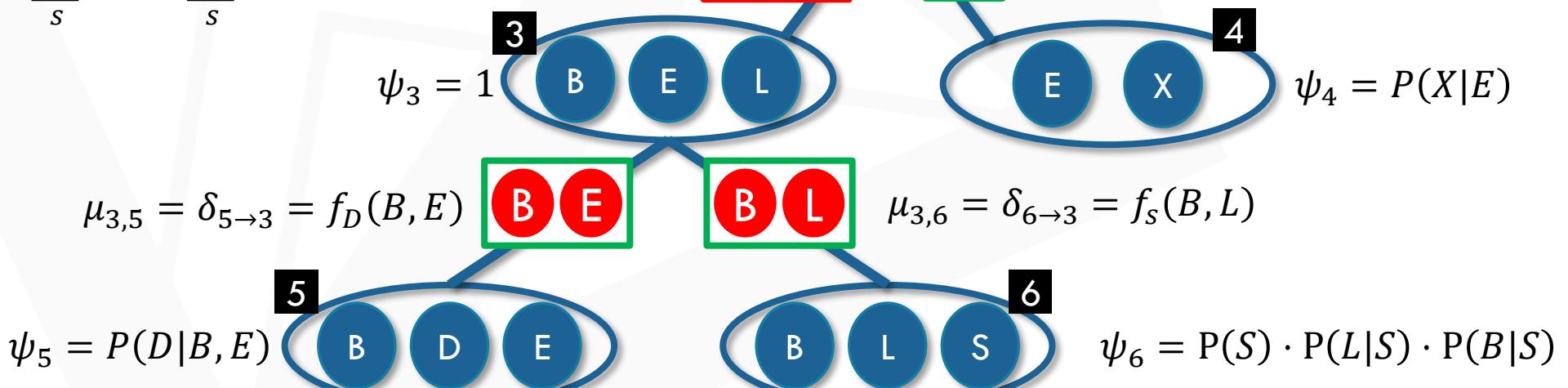
Fase upward (II)

Calcoliamo i messaggi che vanno dalle clique foglia (4, 5 e 6) alle cliques sopra di loro. A questo punto sono gli unici messaggi calcolabili.

$$\delta_{4 \rightarrow 2} = \sum_x \psi_4 = \sum_x P(x|E)$$

$$\delta_{5 \rightarrow 3} = \sum_d \psi_5 = \sum_d P(d|B, E)$$

$$\delta_{6 \rightarrow 3} = \sum_s \psi_6 = \sum_s P(s) \cdot P(L|s) \cdot P(B|s)$$

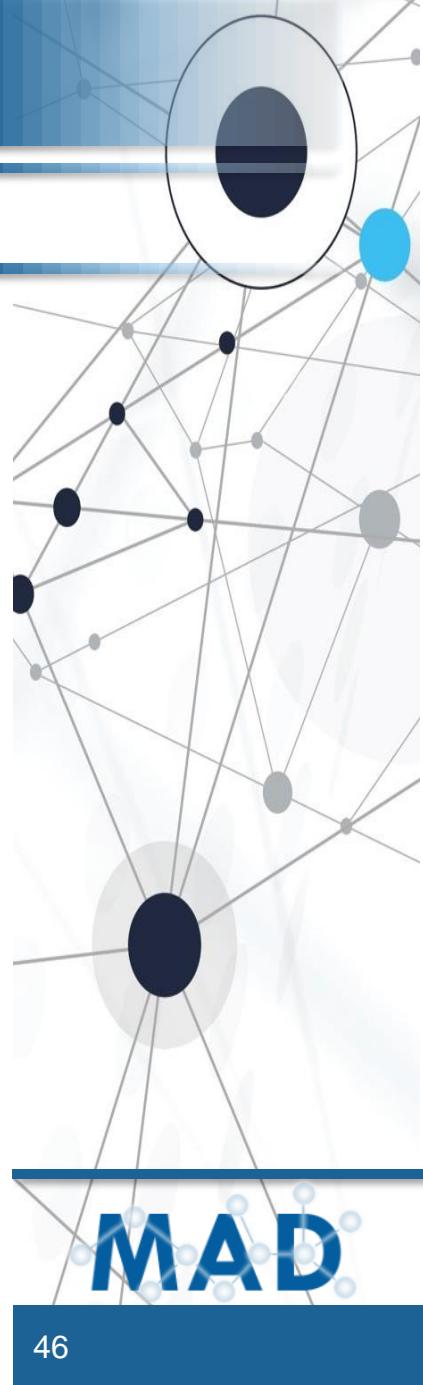
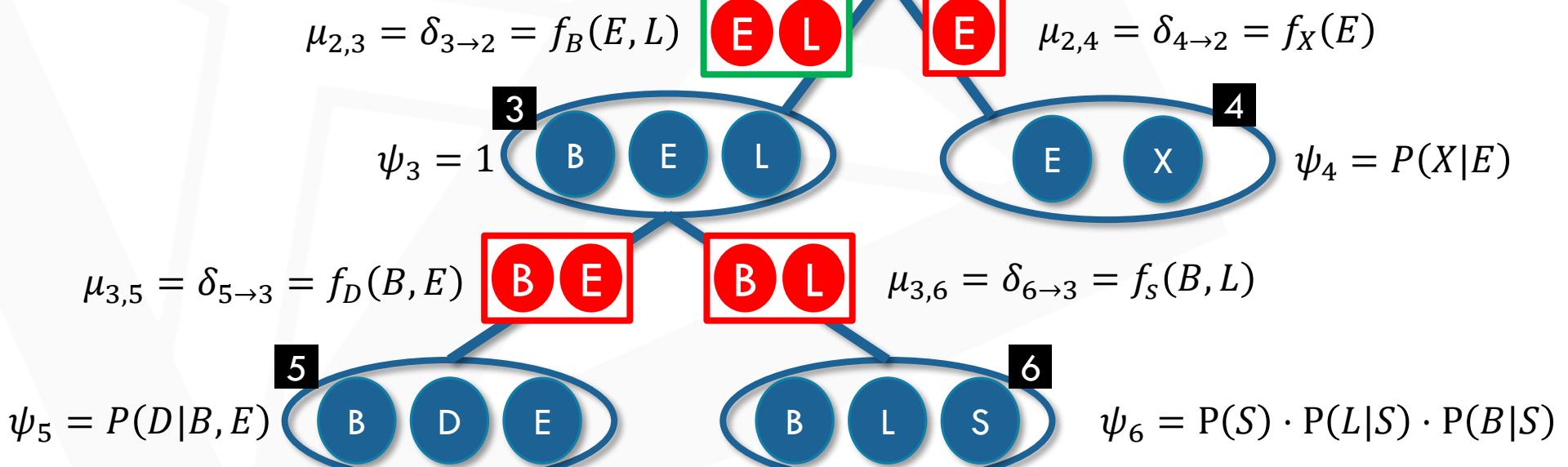


ESEMPIO FINALE: ASIA NETWORK

Fase upward (III)

A questo punto possiamo calcolare il messaggio dalla clique 3 alla clique 2

$$\begin{aligned}\delta_{3 \rightarrow 2} &= \sum_b \psi_3 \cdot \delta_{5 \rightarrow 3} \cdot \delta_{6 \rightarrow 3} \\ &= \sum_b 1 \cdot f_D(b, E) \cdot f_S(b, L)\end{aligned}$$

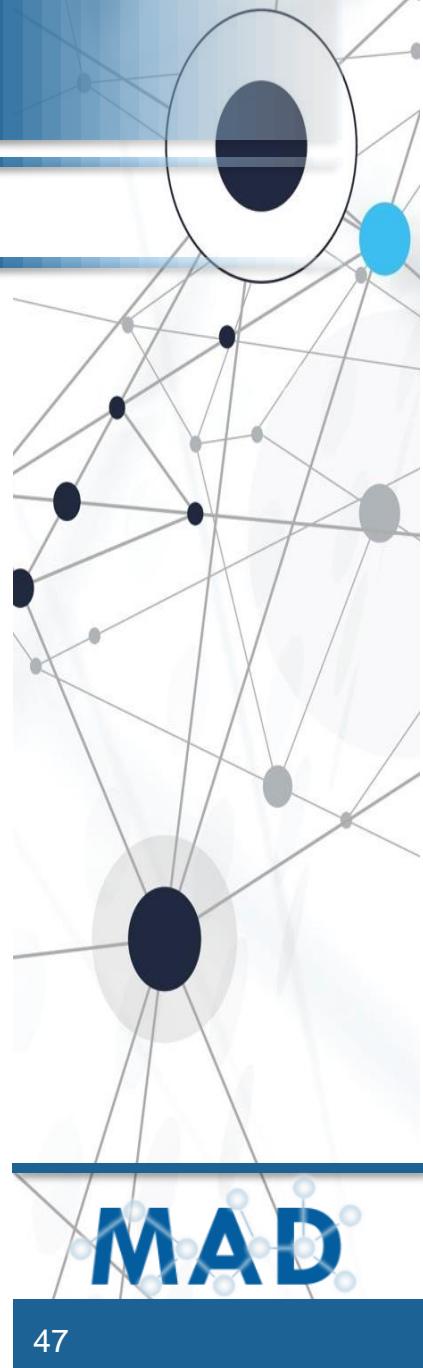
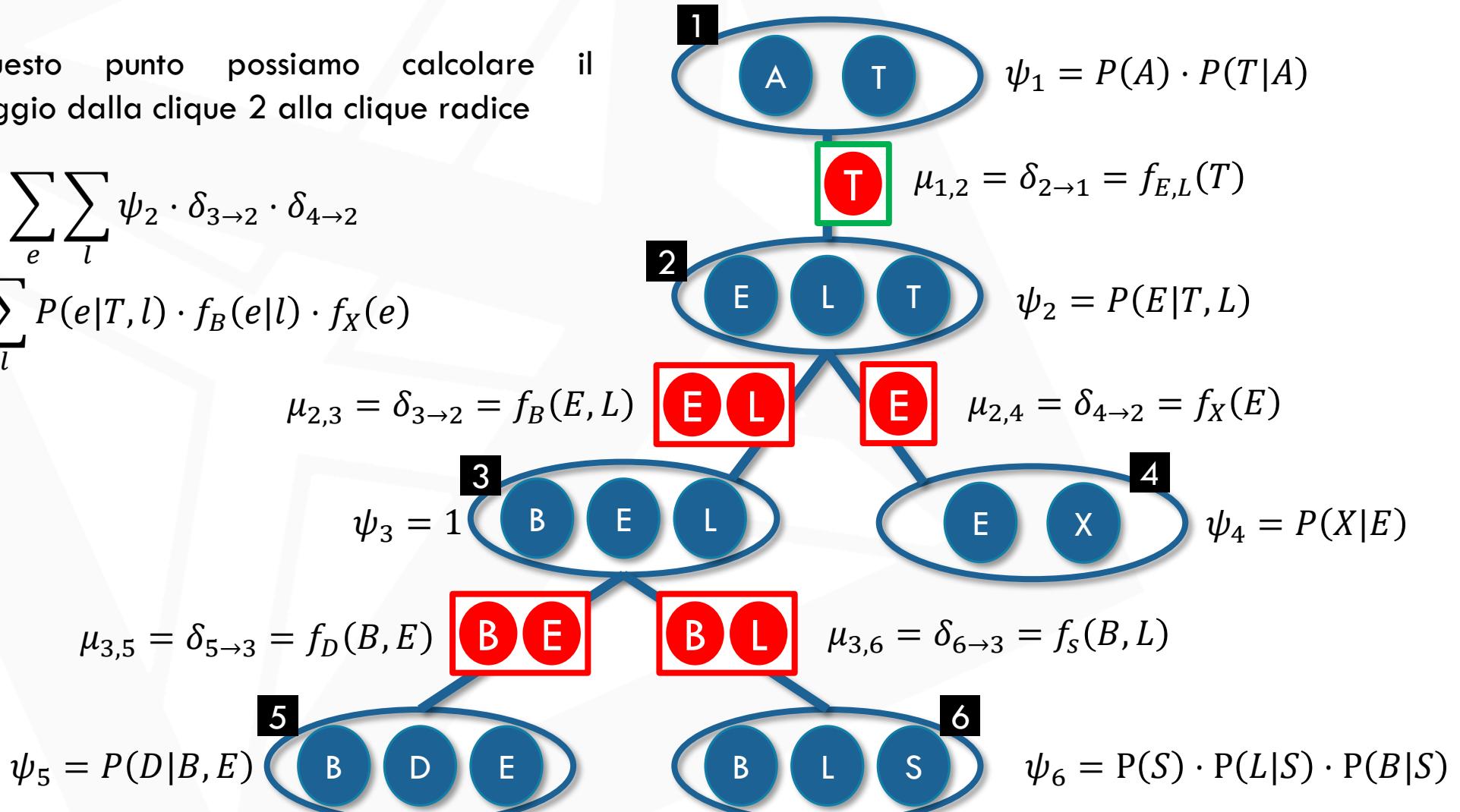


ESEMPIO FINALE: ASIA NETWORK

Fase upward (IV)

A questo punto possiamo calcolare il messaggio dalla clique 2 alla clique radice

$$\begin{aligned}\delta_{2 \rightarrow 1} &= \sum_e \sum_l \psi_2 \cdot \delta_{3 \rightarrow 2} \cdot \delta_{4 \rightarrow 2} \\ &= \sum_e \sum_l P(e|T,l) \cdot f_B(e|l) \cdot f_X(e)\end{aligned}$$

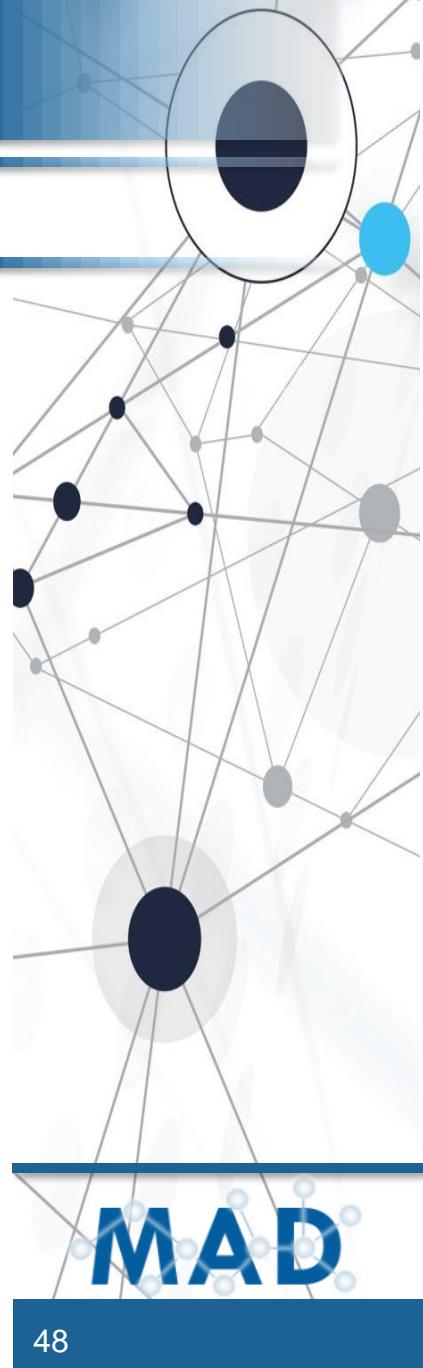
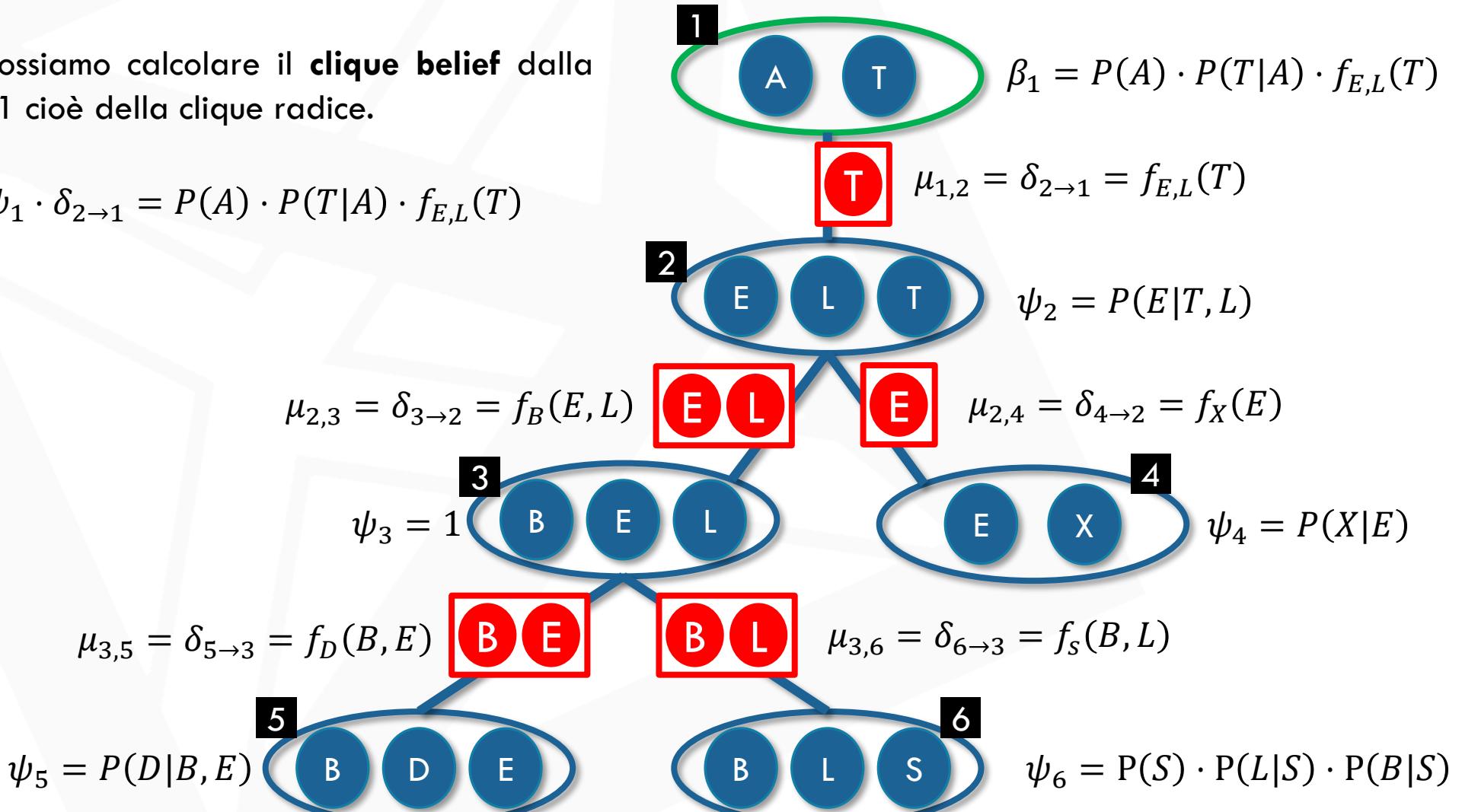


ESEMPIO FINALE: ASIA NETWORK

Fase upward (V)

Ora possiamo calcolare il **clique belief** dalla clique 1 cioè della clique radice.

$$\beta_1 = \psi_1 \cdot \delta_{2 \rightarrow 1} = P(A) \cdot P(T|A) \cdot f_{E,L}(T)$$

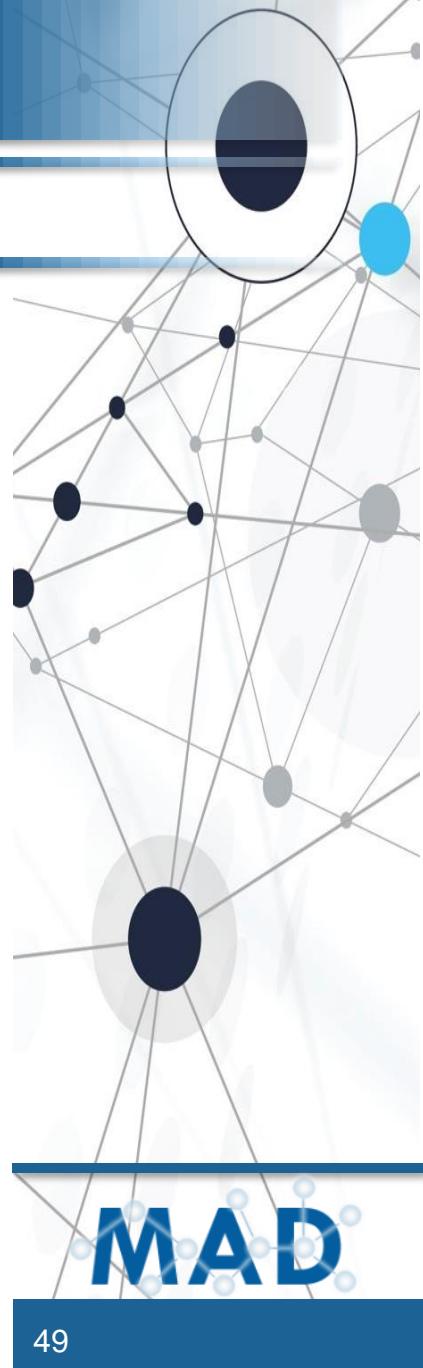
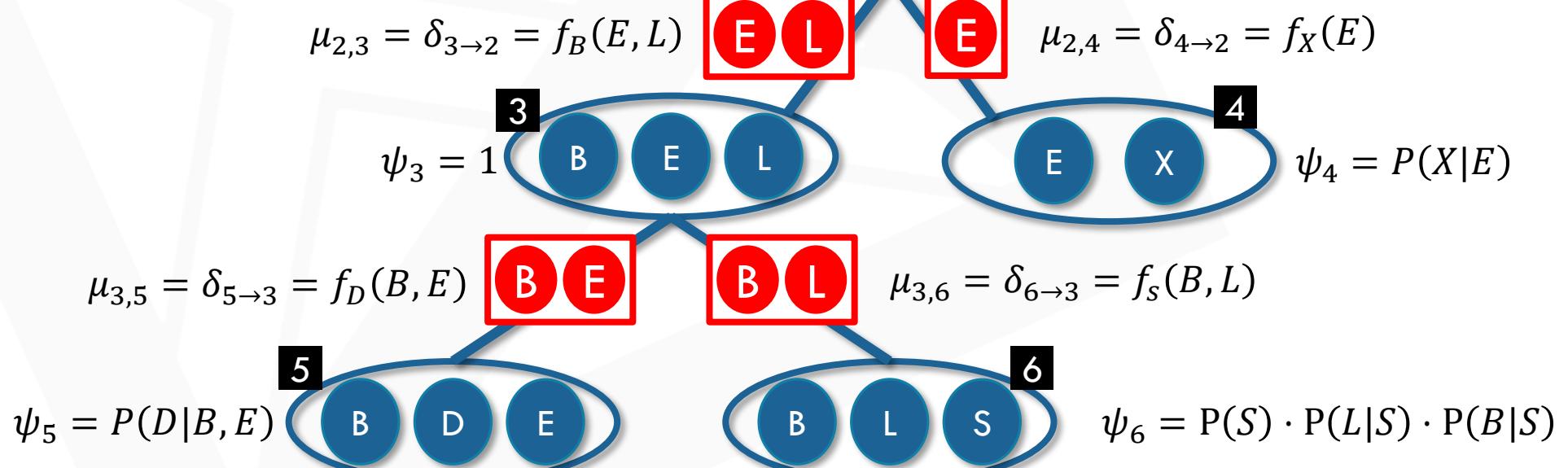


ESEMPIO FINALE: ASIA NETWORK

Fase downward (I)

Aggiorniamo il sepset belief $\mu_{1,2}$ calcolando il messaggio $\delta_{1 \rightarrow 2}$

$$\begin{aligned}\delta_{1 \rightarrow 2} &= \frac{\sum_a \beta_1}{\delta_{2 \rightarrow 1}} = \frac{\sum_a \beta_1}{\mu_{1,2}} \\ &= \frac{\sum_a P(a) \cdot P(T|a) \cdot f_{E,L}(T)}{f_{E,L}(T)} = f_A(T)\end{aligned}$$

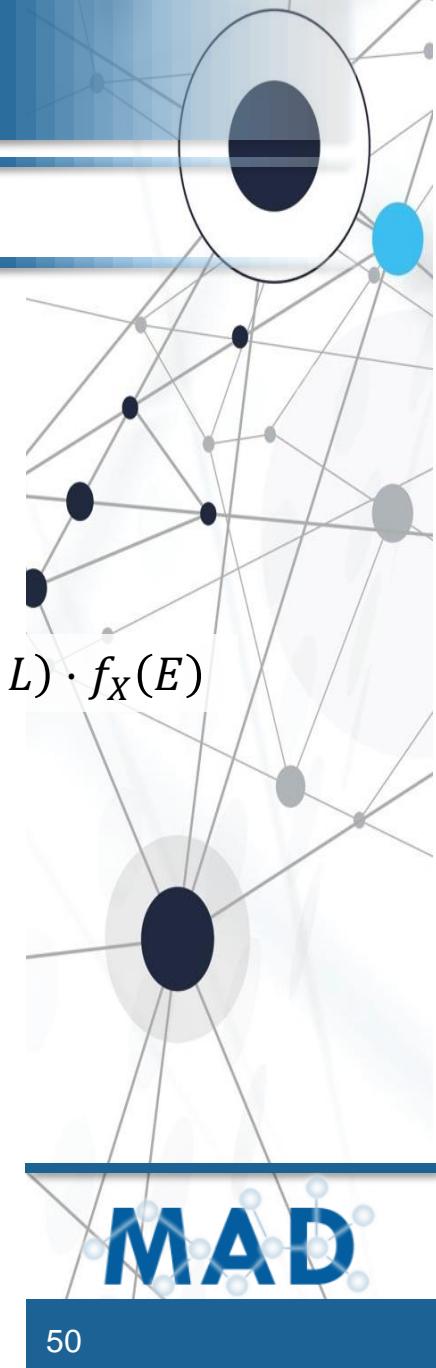
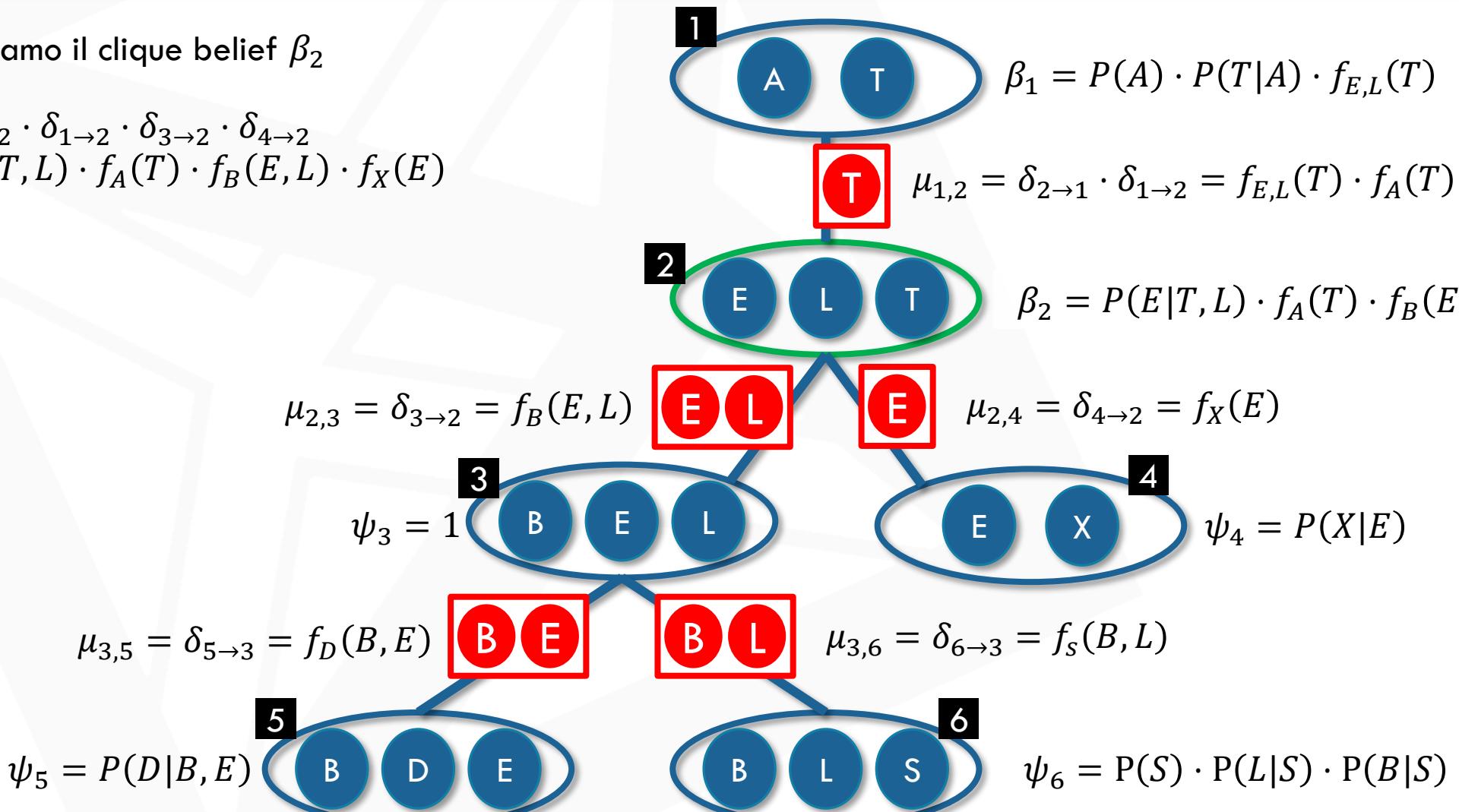


ESEMPIO FINALE: ASIA NETWORK

Fase downward (II)

Calcoliamo il clique belief β_2

$$\begin{aligned}\beta_2 &= \psi_2 \cdot \delta_{1 \rightarrow 2} \cdot \delta_{3 \rightarrow 2} \cdot \delta_{4 \rightarrow 2} \\ &= P(E|T, L) \cdot f_A(T) \cdot f_B(E, L) \cdot f_X(E)\end{aligned}$$

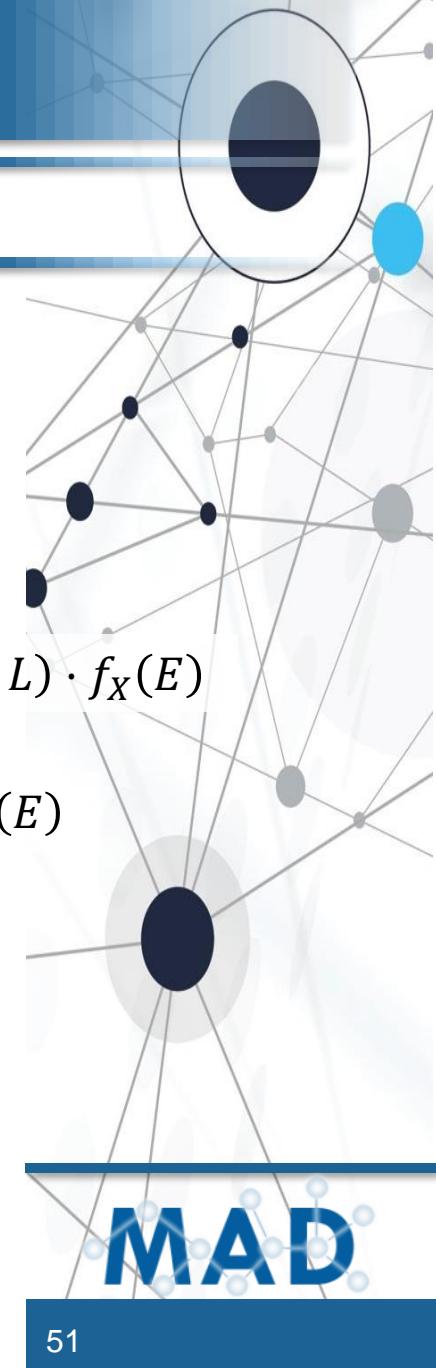
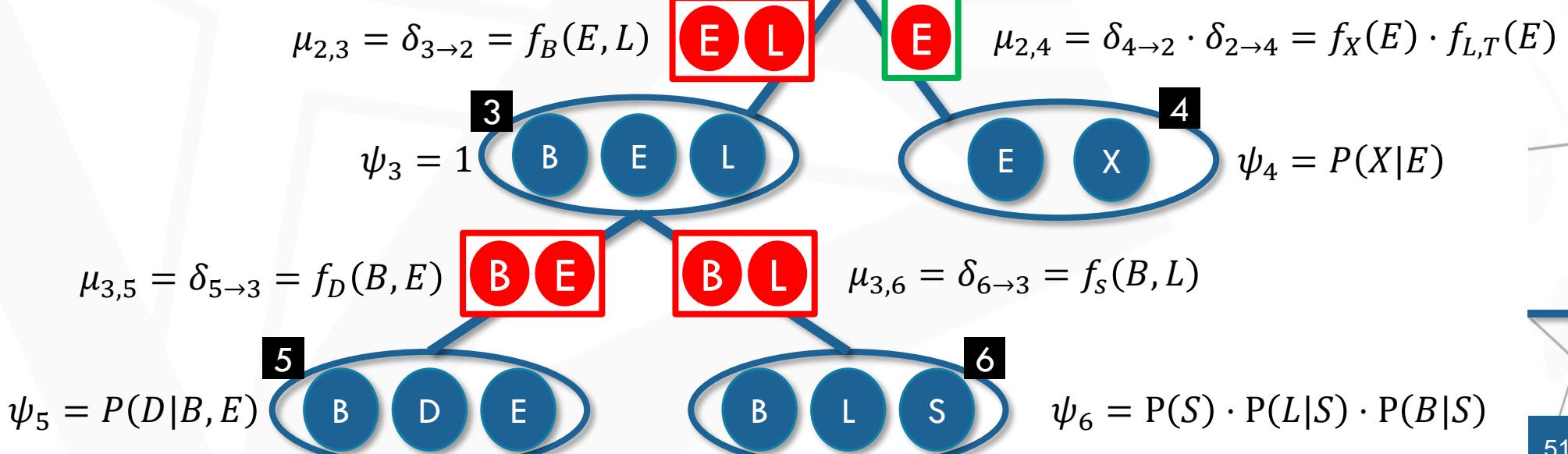


ESEMPIO FINALE: ASIA NETWORK

Fase downward (III)

Aggiorniamo il sepset belief $\mu_{2,4}$ calcolando il messaggio $\delta_{2 \rightarrow 4}$

$$\begin{aligned}\delta_{2 \rightarrow 4} &= \frac{\sum_l \sum_t \beta_2}{\delta_{4 \rightarrow 2}} \\ &= \frac{\sum_l \sum_t P(E|t, l) \cdot f_A(t) \cdot f_B(E, l) \cdot f_X(E)}{f_X(E)} \\ &= f_{L,T}(E)\end{aligned}$$

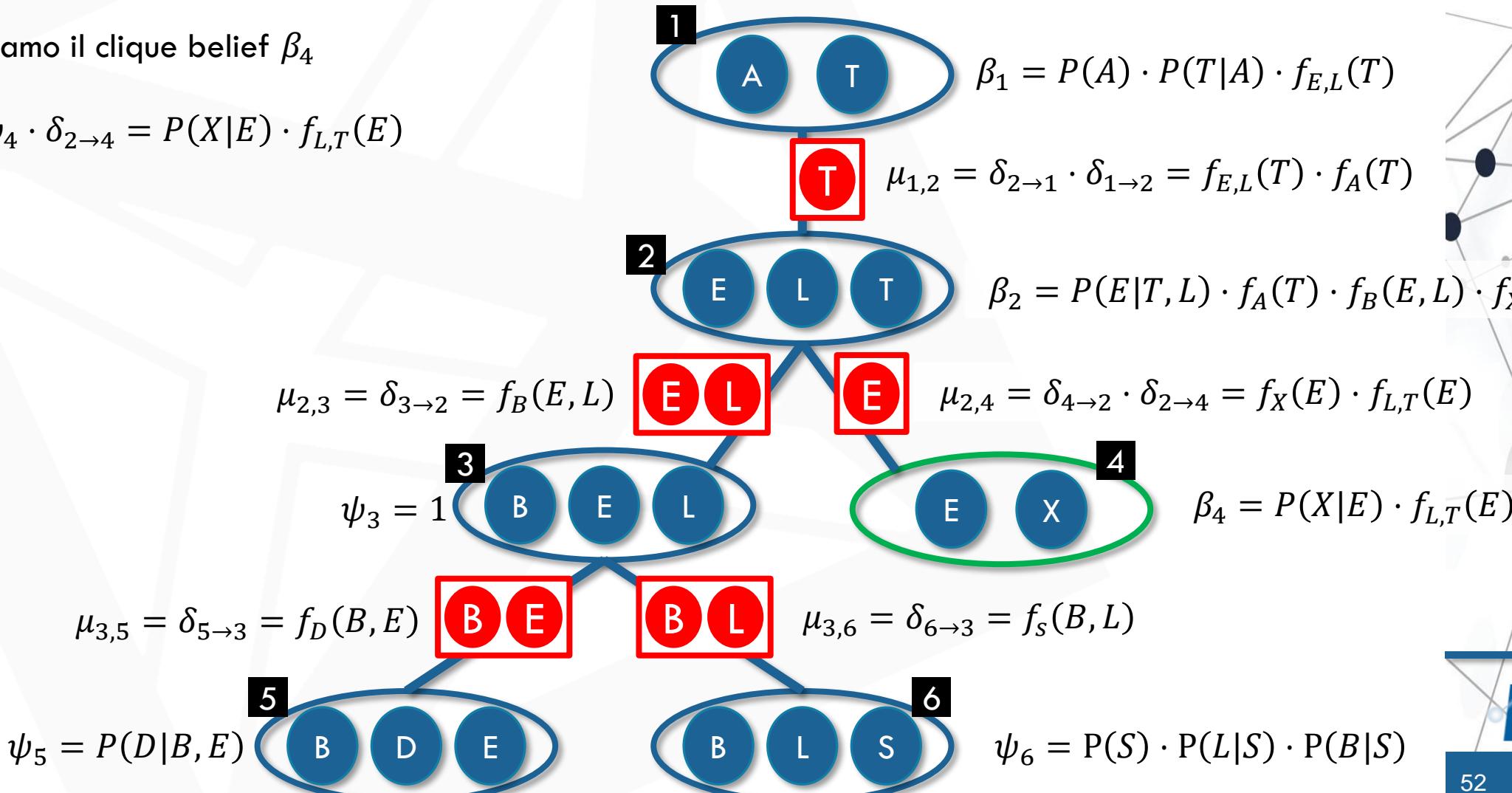


ESEMPIO FINALE: ASIA NETWORK

Fase downward (IV)

Calcoliamo il clique belief β_4

$$\beta_4 = \psi_4 \cdot \delta_{2 \rightarrow 4} = P(X|E) \cdot f_{L,T}(E)$$

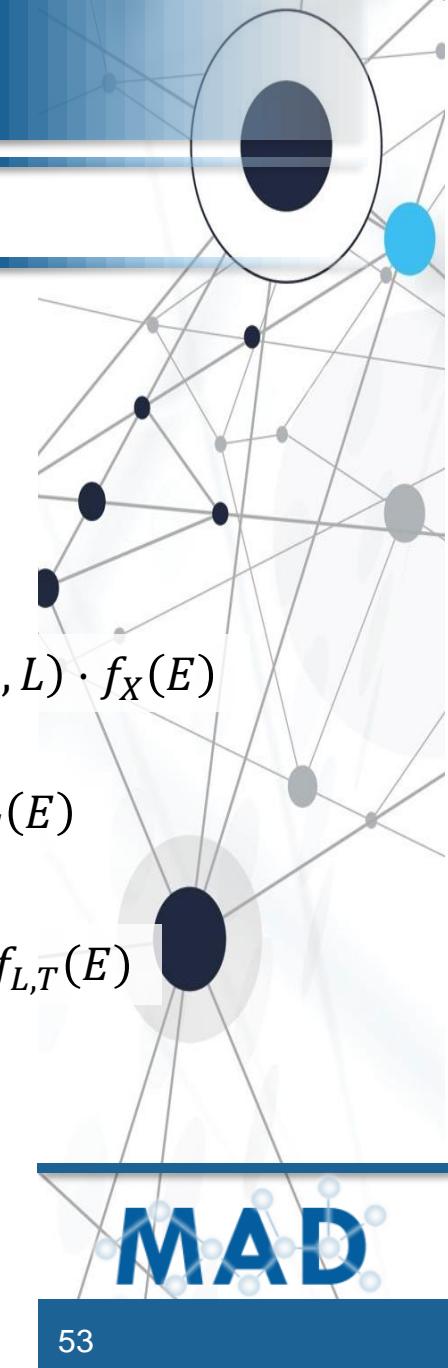
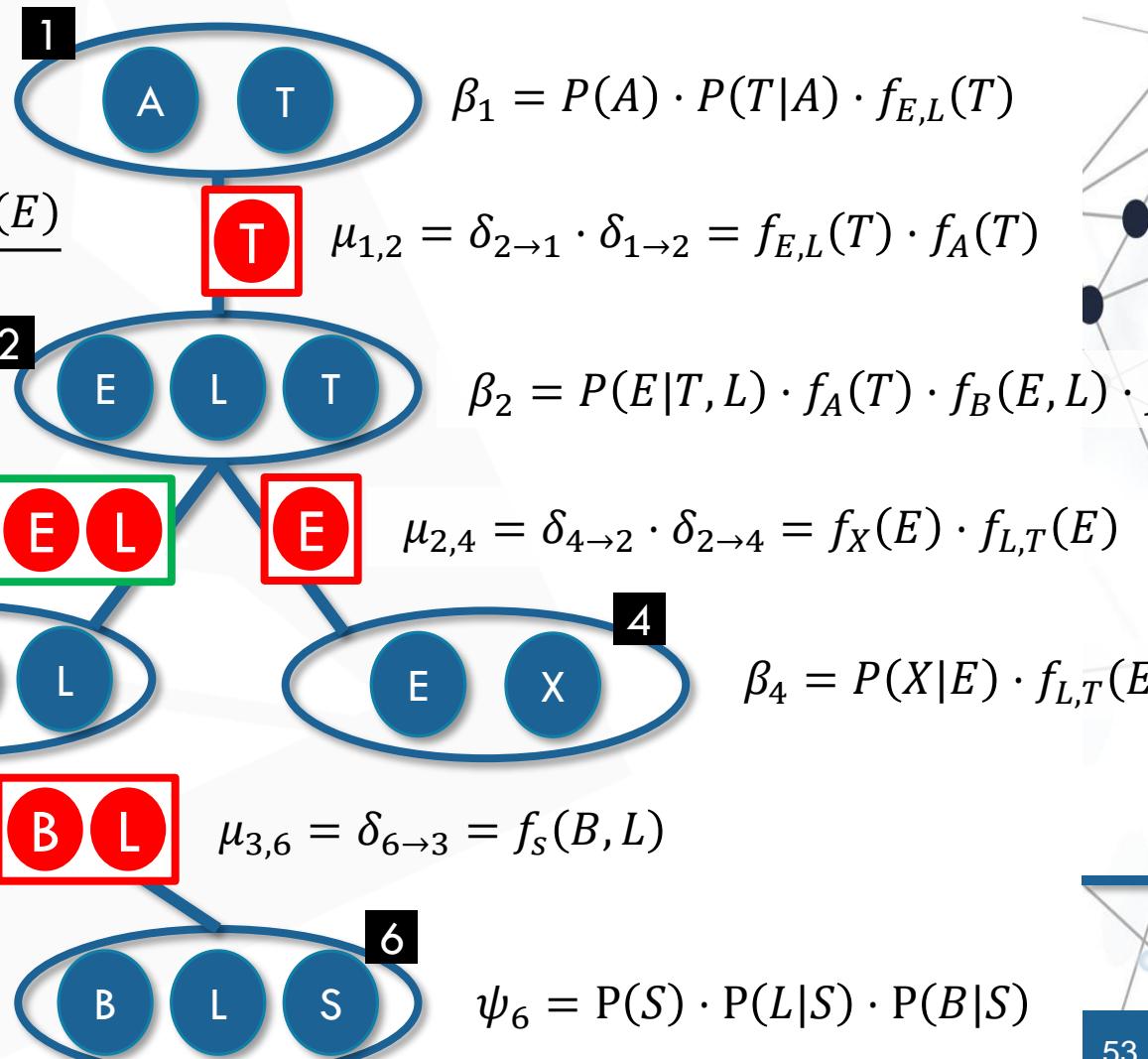


ESEMPIO FINALE: ASIA NETWORK

Fase downward (V)

Aggiorniamo il sepset belief $\mu_{2,3}$ calcolando il messaggio $\delta_{2 \rightarrow 3}$

$$\delta_{2 \rightarrow 3} = \frac{\sum_t \beta_2}{\delta_{3 \rightarrow 2}} = \frac{\sum_t P(E|t, L) \cdot f_A(t) \cdot f_B(E, L) \cdot f_X(E)}{f_B(E, L)} \\ = f_T(E, L)$$

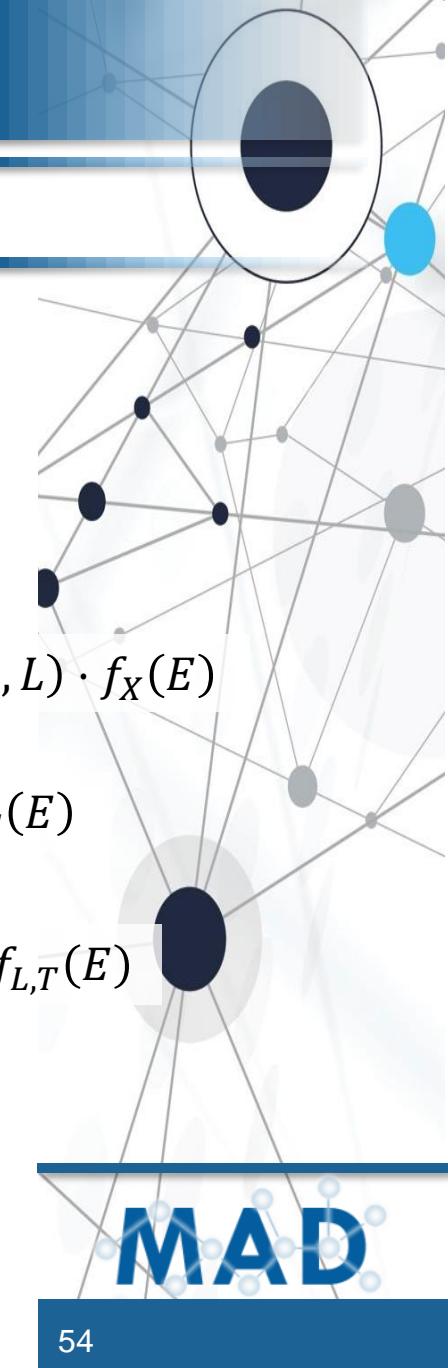
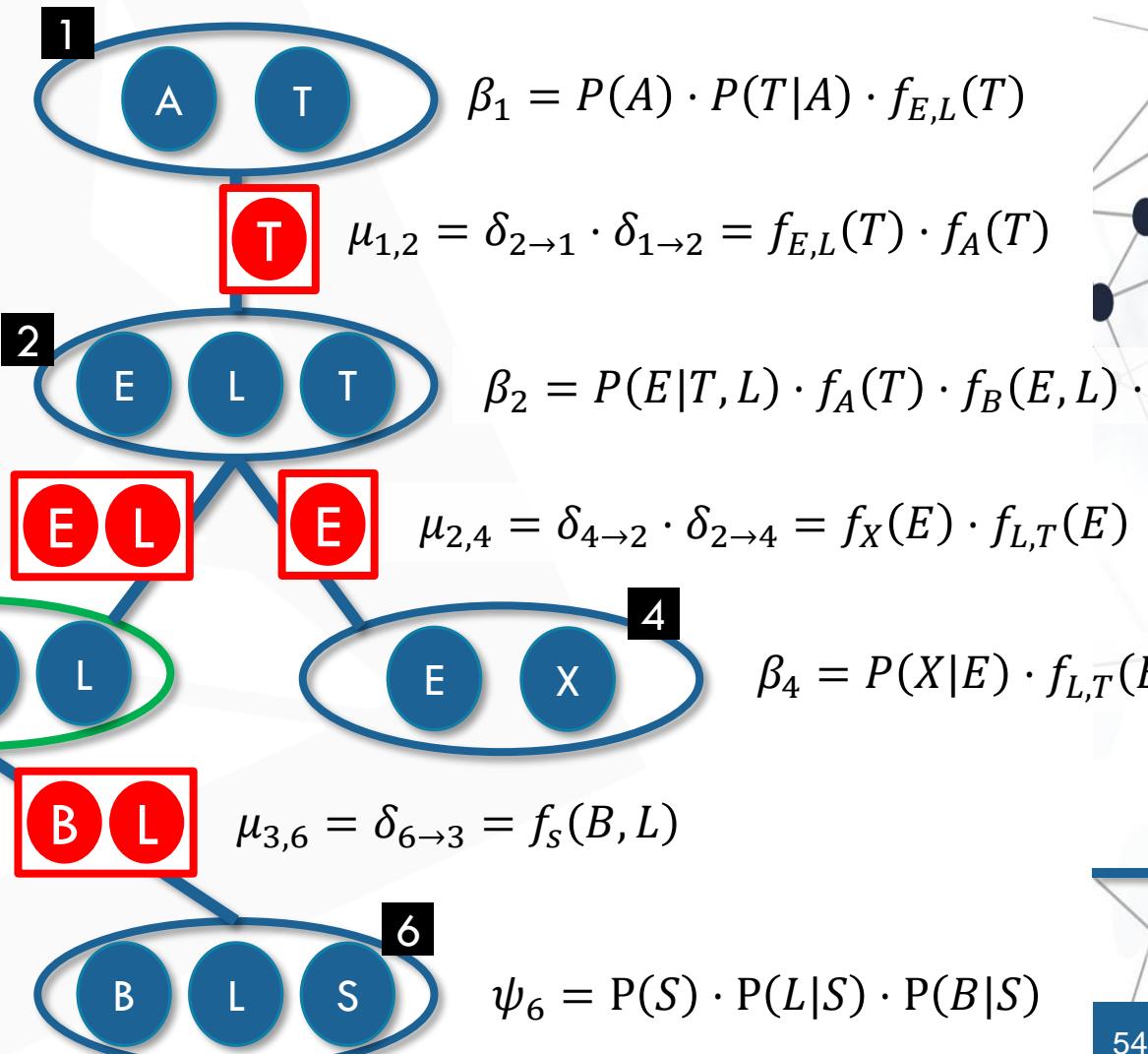


ESEMPIO FINALE: ASIA NETWORK

Fase downward (VI)

Calcoliamo il clique belief β_3

$$\begin{aligned}\beta_3 &= \psi_3 \cdot \delta_{2 \rightarrow 3} \cdot \delta_{5 \rightarrow 3} \cdot \delta_{6 \rightarrow 3} \\ &= 1 \cdot f_T(E, L) \cdot f_D(B, E) \cdot f_S(B, L)\end{aligned}$$



ESEMPIO FINALE: ASIA NETWORK

Fase downward (VII)

Aggiorniamo il sepset belief $\mu_{3,5}$ calcolando il messaggio $\delta_{3 \rightarrow 5}$

$$\begin{aligned}\delta_{3 \rightarrow 5} &= \frac{\sum_l \beta_3}{\delta_{5 \rightarrow 3}} = \frac{\sum_l f_T(E, l) \cdot f_D(B, E) \cdot f_S(B, l)}{f_D(B, E)} \\ &= f_L(B, E)\end{aligned}$$

$$\mu_{2,3} = \delta_{3 \rightarrow 2} \cdot \delta_{2 \rightarrow 3} = f_B(E, L) \cdot f_T(E, L)$$

$$\beta_3 = f_T(E, L) \cdot f_D(B, E) \cdot f_S(B, L)$$

$$\mu_{3,5} = \delta_{5 \rightarrow 3} \cdot \delta_{3 \rightarrow 5} = f_D(B, E) \cdot f_L(B, E)$$

$$\psi_5 = P(D|B, E)$$

$$\beta_1 = P(A) \cdot P(T|A) \cdot f_{E,L}(T)$$

$$\mu_{1,2} = \delta_{2 \rightarrow 1} \cdot \delta_{1 \rightarrow 2} = f_{E,L}(T) \cdot f_A(T)$$

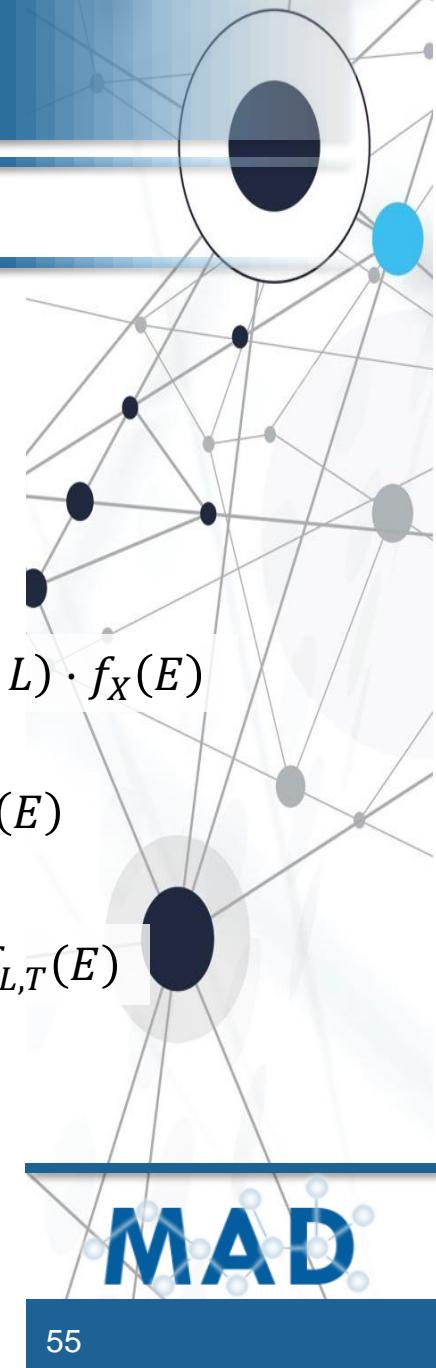
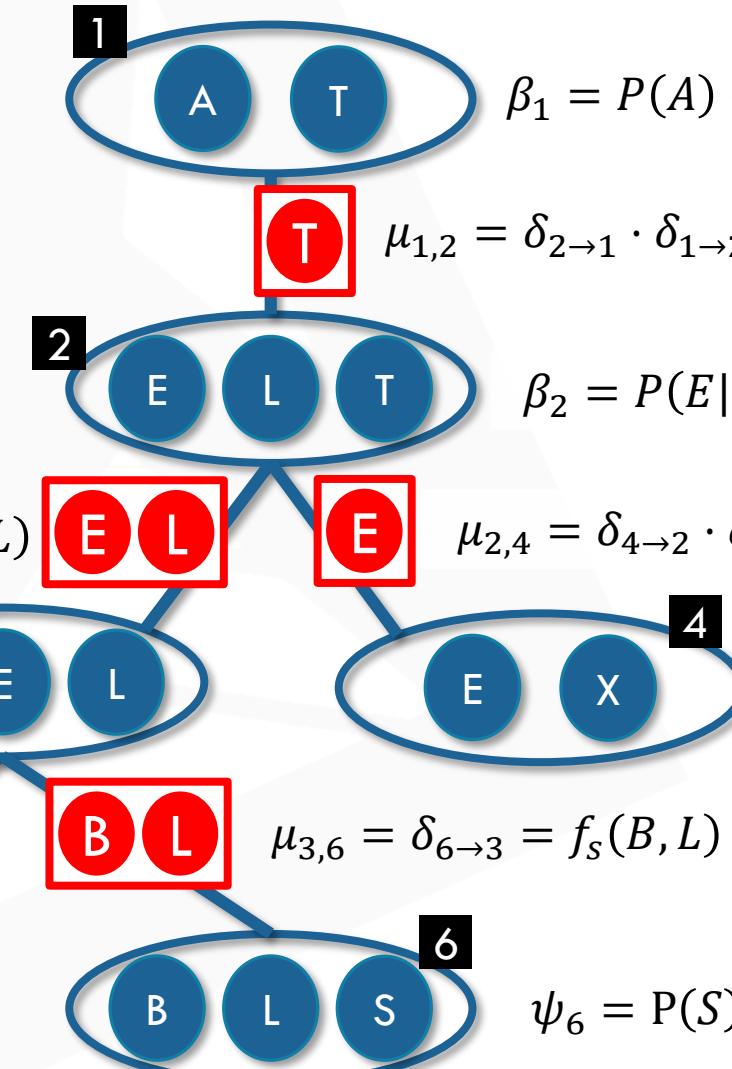
$$\beta_2 = P(E|T, L) \cdot f_A(T) \cdot f_B(E, L) \cdot f_X(E)$$

$$\mu_{2,4} = \delta_{4 \rightarrow 2} \cdot \delta_{2 \rightarrow 4} = f_X(E) \cdot f_{L,T}(E)$$

$$\beta_4 = P(X|E) \cdot f_{L,T}(E)$$

$$\mu_{3,6} = \delta_{6 \rightarrow 3} = f_S(B, L)$$

$$\psi_6 = P(S) \cdot P(L|S) \cdot P(B|S)$$



ESEMPIO FINALE: ASIA NETWORK

Fase downward (VIII)

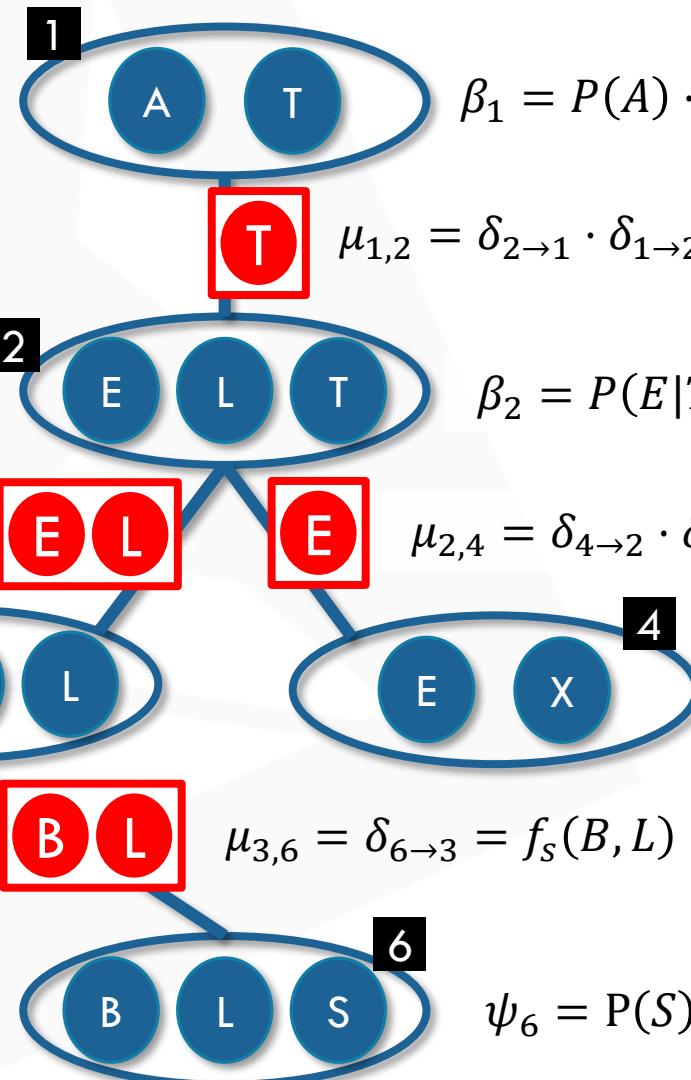
Calcoliamo il clique belief β_5

$$\beta_5 = \psi_5 \cdot \delta_{3 \rightarrow 5} = P(D|B,E) \cdot f_L(B,E)$$

$$\beta_3 = f_T(E,L) \cdot f_D(B,E) \cdot f_S(B,L)$$

$$\mu_{3,5} = \delta_{5 \rightarrow 3} \cdot \delta_{3 \rightarrow 5} = f_D(B,E) \cdot f_L(B,E)$$

$$\beta_5 = P(D|B,E) \cdot f_L(B,E)$$



$$\beta_1 = P(A) \cdot P(T|A) \cdot f_{E,L}(T)$$

$$\mu_{1,2} = \delta_{2 \rightarrow 1} \cdot \delta_{1 \rightarrow 2} = f_{E,L}(T) \cdot f_A(T)$$

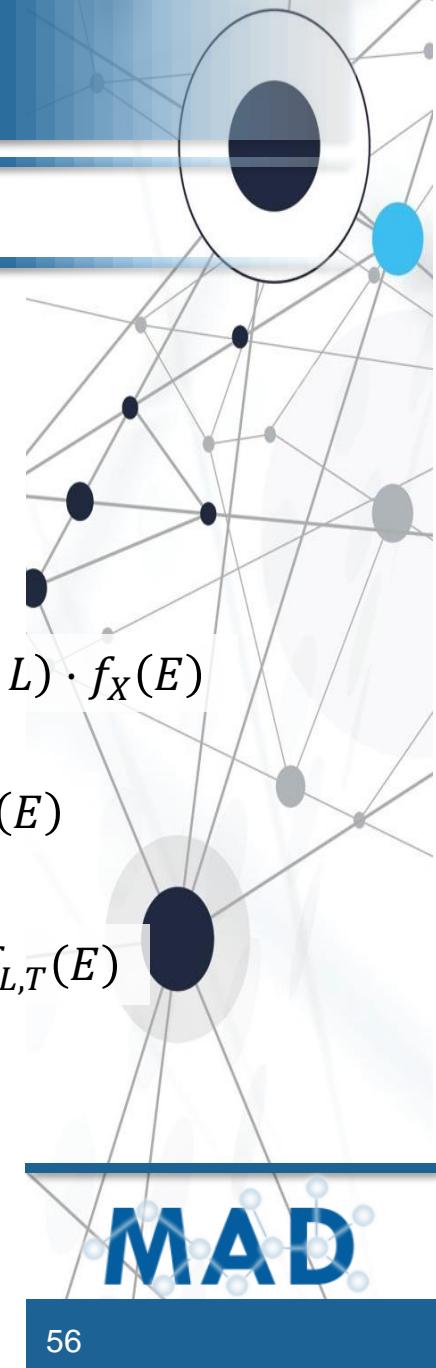
$$\beta_2 = P(E|T,L) \cdot f_A(T) \cdot f_B(E,L) \cdot f_X(E)$$

$$\mu_{2,4} = \delta_{4 \rightarrow 2} \cdot \delta_{2 \rightarrow 4} = f_X(E) \cdot f_{L,T}(E)$$

$$\beta_4 = P(X|E) \cdot f_{L,T}(E)$$

$$\mu_{3,6} = \delta_{6 \rightarrow 3} = f_S(B,L)$$

$$\psi_6 = P(S) \cdot P(L|S) \cdot P(B|S)$$

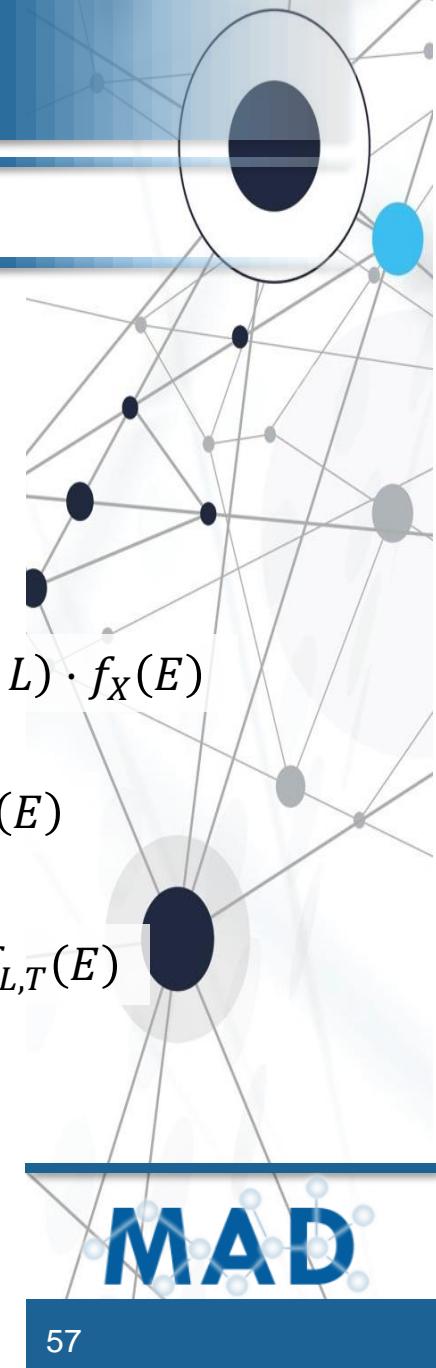
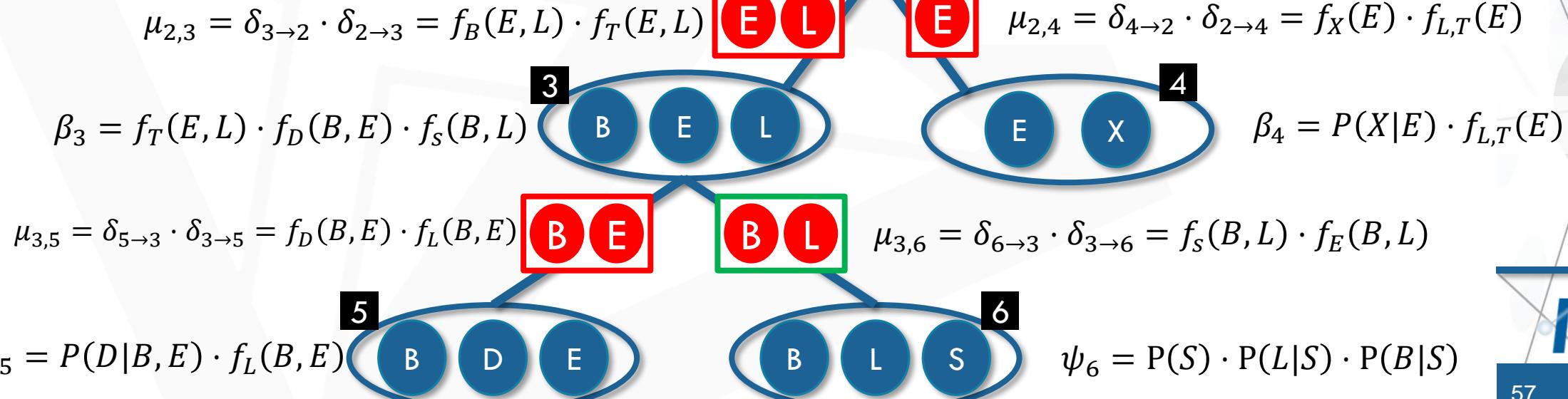


ESEMPIO FINALE: ASIA NETWORK

Fase downward (IX)

Aggiorniamo il sepset belief $\mu_{3,6}$ calcolando il messaggio $\delta_{3 \rightarrow 6}$

$$\begin{aligned}\delta_{3 \rightarrow 6} &= \frac{\sum_e \beta_3}{\delta_{5 \rightarrow 3}} \\ &= \frac{\sum_e f_T(e, L) \cdot f_D(B, e) \cdot f_S(B, L)}{f_S(B, L)} = f_E(B, L)\end{aligned}$$

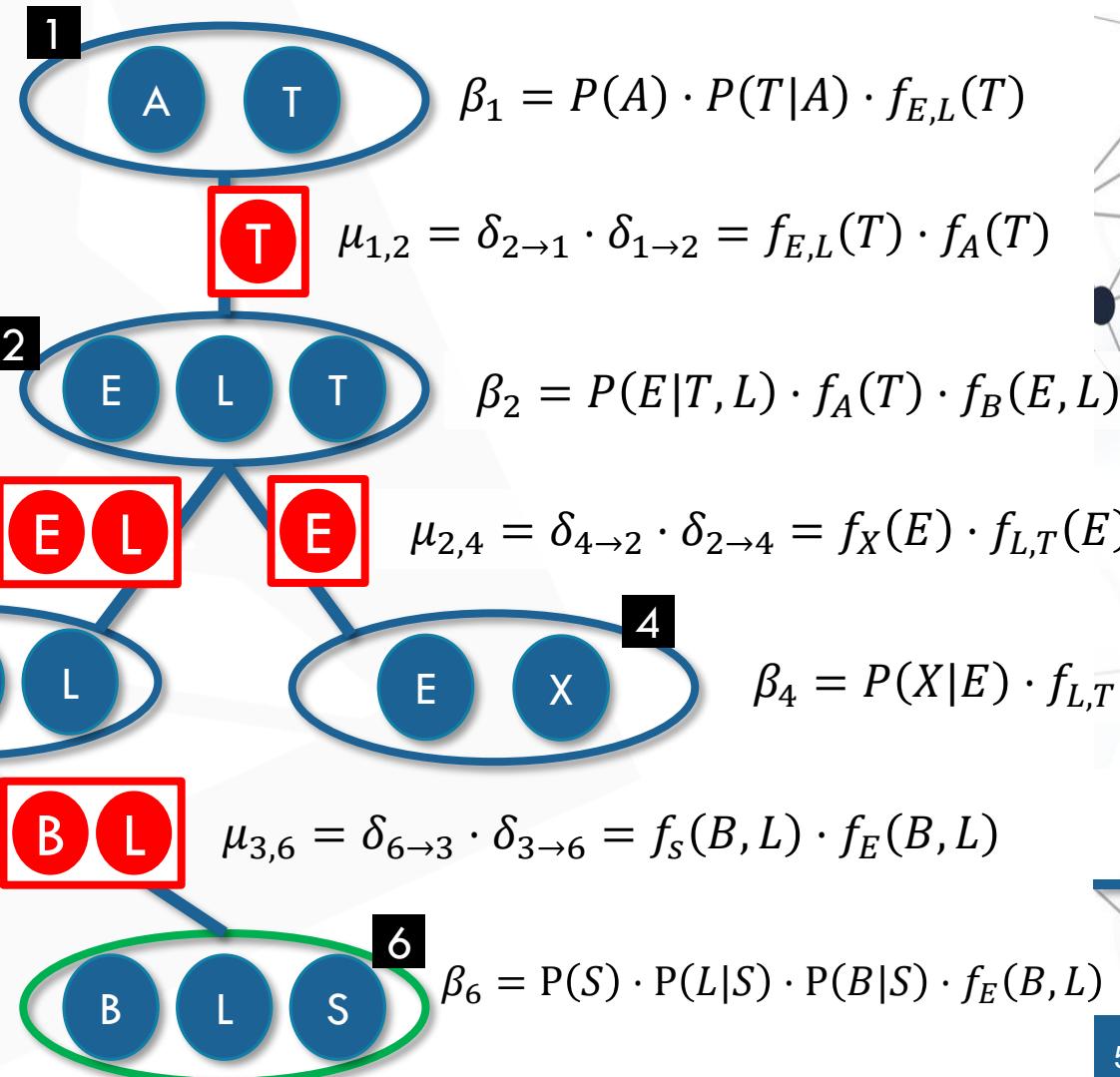


ESEMPIO FINALE: ASIA NETWORK

Fase downward (X)

Calcoliamo il clique belief β_6

$$\beta_6 = \psi_6 \cdot \delta_{3 \rightarrow 6} = P(S) \cdot P(L|S) \cdot P(B|S) \cdot f_E(B, L)$$



AGGIUNTA DI EVIDENZE

Evidenza di una variabile

MAD

AGGIUNTA DI EVIDENZE

L'aggiunta di un'evidenza, per esempio durante una query, viene fatta creando un nuovo Junction Tree con la stessa struttura e i **clique belief** aggiornati. In questo modo non perderemo il JT calcolato precedentemente che potrà essere usato per altre operazioni ma verrà modificata solo la copia.

Procediamo quindi per i seguenti step:

- Cerchiamo una clique C_i in cui sia presente la variabile E di cui si vuole settare l'evidenza. L'evidenza ci fornirà il livello della variabile E che avrà valore 1 oppure l'intera distribuzione di probabilità della variabile secondo la nuova evidenza. Indichiamo con E_{new} questa probabilità.

$$E_{new} \leftarrow \{X_{ik} = 1\} \quad \text{oppure} \quad E_{new} \leftarrow \{X_{i_1} = e_1, X_{i_2} = e_2, \dots, X_{ik} = e_k\}$$

- Otteniamo dalla clique C_i la probabilità marginale di E prima che ne venga settata l'evidenza. Questa probabilità verrà chiamata E_{old} .

$$E_{old} \leftarrow \sum_{C_i - E} \beta_i(C_i)$$



AGGIUNTA DI EVIDENZE

- Dividiamo il **clique belief** β_i per la probabilità E_{old} e moltiplichiamola per la nuova probabilità E_{new} . Chiamiamo il nuovo **clique belief** β_i^{new} .

$$\beta_i^{new} \leftarrow \frac{\beta_i(C_i)}{E_{old}} \cdot E_{new}$$

- A questo punto il **nuovo clique belief** β_i^{new} della clique C_i è coerente con la nuova evidenza aggiunta. Dobbiamo quindi propagare il cambiamento a tutte le altre clique del JT. Per farlo è sufficiente eseguire una **fase downward** dell'algoritmo di message passing descritto precedentemente utilizzando come clique radice la clique C_i già aggiornata.
- Calcoliamo quindi il nuovo **sepset belief** $\mu_{i,k}^{new}$ come marginalizzazione di β_i^{new} :

$$\mu_{i,k}^{new} = \sum_{C_i - S_{i,k}} \beta_i^{new}(C_i)$$

- Il generico **clique belief** β_k^{new} della generica clique C_k verrà calcolato quindi come:

$$\beta_k^{new} \leftarrow \frac{\beta_k(C_k)}{\mu_{i,k}} \cdot \mu_{i,k}^{new}$$



INFERENZA SUL JUNCTION TREE

Singola variabile

Più variabili in una clique

Più variabili in più cliques

MAD

INFERENZA SUL JUNCTION TREE

Introduzione

Una volta costruita la struttura del Junction Tree, inizializzate le clique e calibrato l'albero possiamo eseguire query direttamente al JT senza più considerare la struttura del DAG iniziale.

Per implementare il meccanismo delle query le dividiamo in tre tipologie in base ai nodi e alle clique:

- **Query di una variabile:** viene chiesta la probabilità marginale di una variabile.
- **Query di più variabili nella stessa clique:** vengono chieste le probabilità di più variabili che sono contenute in una stessa clique.
- **Query a più variabili in più cliques:** vengono chieste le probabilità di più variabili che **non** sono tutte contenute in una stessa clique del Junction Tree.

Similmente a quanto fatto nella libreria **gRain**[7] vengono rese disponibili tre tipologie di query:

- **Marginal:** restituisce le probabilità marginali delle variabili richieste
- **Joint:** restituisce le probabilità congiunte delle variabili richieste
- **Conditional:** restituisce le probabilità condizionate rispetto alle variabili richieste

INFERENZA SUL JUNCTION TREE

Singola variabile e più variabili in una stessa clique

Trattiamo insieme i casi in cui venga chiesta una singola variabile oppure vengano chieste più variabili che sono presenti nella stessa clique. Questo perché è sufficiente utilizzare il **Clique Belief** della singola clique.

Individuata la clique C_i che conterrà il **Clique Belief** $\beta_i(C_i)$ che contiene la/le variabile/i richieste dalla query $Q(X_1, X_2, \dots, X_n)$, distinguiamo per i tre tipi di query disponibili:

- **Marginal:** in questo caso basterà marginalizzare rispetto ad una variabile alla volta:
- **Joint:** in questo caso basterà marginalizzare rispetto alla/alle variabile/i:

$$Q(X_1, X_2, \dots, X_n) = \sum_{X \notin Q} \beta_i(C_i)$$

- **Conditional:** in questo caso la query sarà nella forma $Q(X_1 | Y_1, Y_2, \dots, Y_m)$. Chiediamo che ci venga restituita la **joint** di tutte le variabili e la **joint** delle variabili Y e, applicando la definizione di probabilità condizionata, otterremo:

$$Q(X_1 | Y_1, Y_2, \dots, Y_m) = \frac{\sum_{C_i - (X+Y)} \beta_i(C_i)}{\sum_{C_i - Y} \beta_i(C_i)}$$



MAD

INFERENZA SUL JUNCTION TREE

Più variabili in più cliques

In questo caso la soluzione migliore che sfrutti il Junction Tree costruito è quella che esegue l'algoritmo **Variable Elimination**. Il vantaggio computazionale qui risiede nel poter evitare di eseguire l'algoritmo su tutti i fattori ma sul poterlo eseguire sui fattori presenti nel sotto albero formato dal minor numero di cliques che contengono le variabili richieste.

L'algoritmo Variable Elimination non può essere applicato direttamente con i Belief, per cui prima di eseguire l'algoritmo calcoliamo i parametri Φ su cui eseguirlo.

Dove:

- T è il JT su Φ
- Q è la query

```
def CTree-Query( $T, \{\beta_i\}, \{\mu_{i,j}\}, Q$ ):  
    Let  $T'$  be a subtree of  $T$  such that  $Q \subseteq \text{Scope}[T']$   
    Select a clique  $r \in V_{T'}$ , to be the root  
     $\Phi \leftarrow \beta_r$   
    for each  $i \in V_{T'} - \{r\}$   
         $\phi \leftarrow \frac{\beta_i}{\mu_{i,pr(i)}}$   
         $\Phi \leftarrow \Phi \cup \{\phi\}$   
     $Z \leftarrow \text{Scope}[T'] - Q$   
    Let  $\prec$  be some ordering over  $Z$   
  
    return Sum-Product-VE( $\Phi, Z, \prec$ )
```



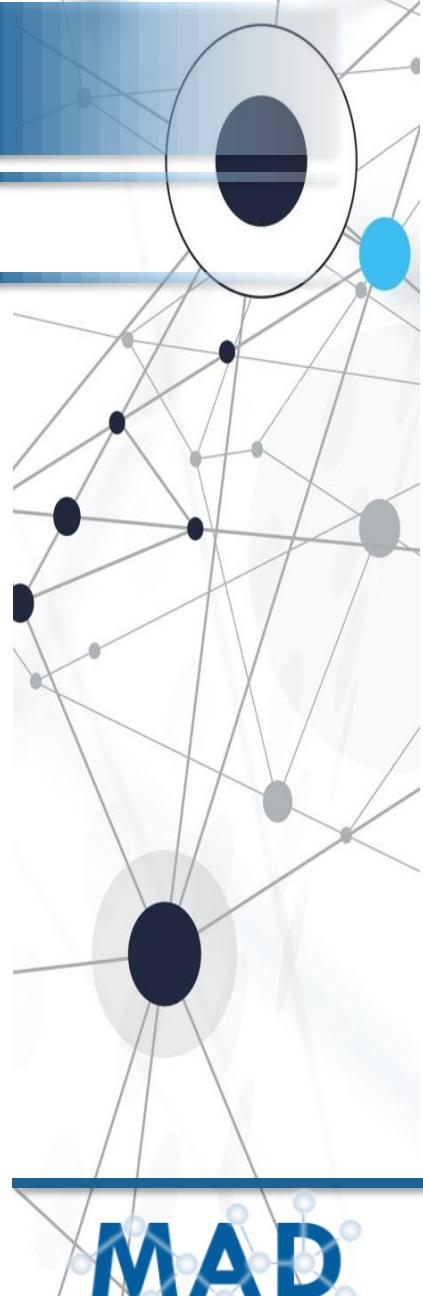
INFERENZA SUL JUNCTION TREE

Più variabili in più cliques

L'algoritmo Variable Elimination proposto[6] è formato da due parti:

```
def Sum-Product-VE( $\Phi, Z, \prec$ ):  
    Let  $Z_1, \dots, Z_k$  be an ordering of  $Z$  such that  $Z_i \prec Z_j$  if and only if  $i < j$   
    for  $i = 1, \dots, k$   
         $\Phi \leftarrow$  Sum-Product-Eliminate-Var( $\Phi, Z_i$ )  
     $\phi^* \leftarrow \prod_{\phi \in \Phi} \phi$   
    return  $\phi^*$ 
```

```
def Sum-Product-Eliminate-Var( $\Phi, Z$ ):  
     $\Phi' \leftarrow \{\phi \in \Phi : Z \in \text{Scope}[\phi]\}$   
     $\Phi'' \leftarrow \Phi - \Phi'$   
     $\psi \leftarrow \prod_{\phi \in \Phi'} \phi$   
     $\tau \leftarrow \sum_Z \psi$   
    return  $\Phi'' \cup \{\tau\}$ 
```



INFERENZA SUL JUNCTION TREE

Più variabili in più cliques

Presentiamo però una versione dell'algoritmo precedente che consente di sfruttare il Junction Tree per eseguire l'algoritmo Variable Elimination direttamente sul JT così come visto precedentemente.

Supponendo di avere un algoritmo che estragga un sottoalbero valido dal JT, anche in questo caso sceglieremo una clique che funga da radice del sottoalbero che stiamo considerando.

Come visto non possiamo eseguire VE direttamente con i **clique belief** quindi recuperiamo questi fattori risalendo dai nodi foglia del sottoalbero fino alla radice dividendo i **clique belief** per i **sepset belief** direttamente sopra di loro.

Prima di passare il «messaggio» verso la clique soprastante marginalizziamo, come sempre, rispetto alle variabili che **non** sono presenti nella clique sopra avendo però cura di mantenere, diversamente dal solito, le variabili richieste dalla query, escludendole quindi dalla marginalizzazione.

Il vantaggio rispetto all'algoritmo precedente è duplice:

- **In termini di tempo:** evitiamo di dover scorrere la lista dei fattori Φ ogni volta per cercare i fattori che appartengono allo scope della variabile da marginalizzare.
- **In termini di spazio:** evitiamo di mantenere in memoria una CPD con tutte le variabili incontrate ma manteniamo solamente quelle che non possono essere marginalizzate subito.



INFERENZA SUL JUNCTION TREE

Più variabili in più cliques

Resta da trovare il sotto albero che contenga tutte le variabili richieste nella query, preferibilmente il sotto albero minimale.

Per eseguire rapidamente questa ricerca sfruttiamo la proprietà di Running Intersection Property dei JT.

Notiamo che per rispondere alla query $Q(X)$ è sufficiente che il sotto albero contenga almeno una clique, e non tutte, con la variabile X .

```
def build_sub_tree(variables, clique):
    if clique is a separator:
        message = build_sub_tree(variables, clique.children)
        if message is a Probability Table:
            message = message / clique[belief]
    return message

    message = 1; found = False
    for variable in target_clique[nodes]:
        if variable in variables:
            variables.remove(variable)
            found = True
    for sep in target_clique.children():
        if len(variables) > 0:
            message = message * build_sub_tree(variables, sep)
    if (message is a Probability Table) or (found == True):
        message = message * target_clique[belief]

    return message
```

INFERENZA SUL JUNCTION TREE

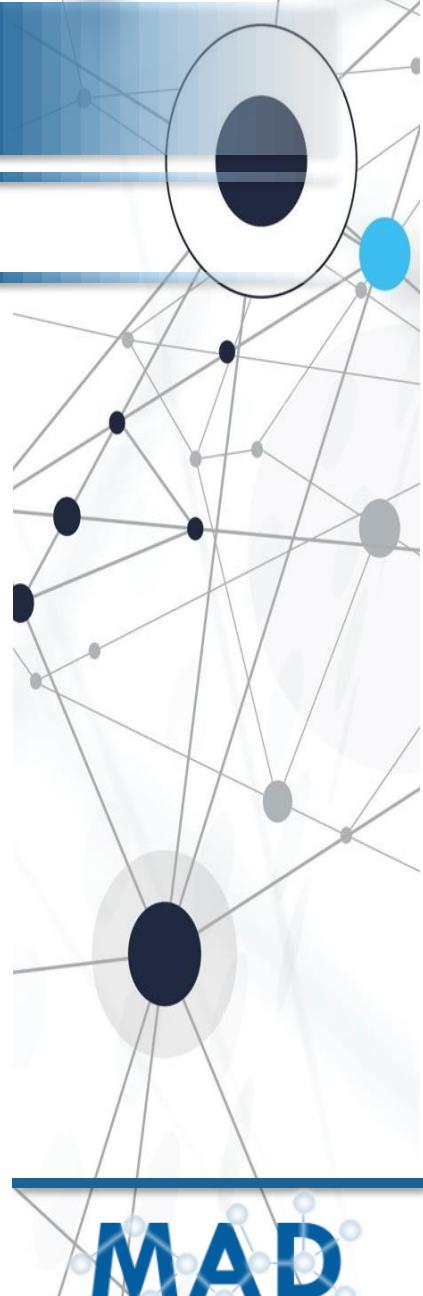
Più variabili in più cliques

Come ulteriore **miglioramento** notiamo che è possibile unificare le due fasi di **costruzione e calcolo della CPT** finale svolgendo entrambe le funzioni contemporaneamente.

Questo può essere fatto durante la fase di risalita della ricerca DFS, possiamo calcolare il «messaggio» che deve essere portato dalla clique sottostante a quella sopra senza aspettare di doverlo fare, nello stesso identico modo una volta creato fisicamente il sotto albero.

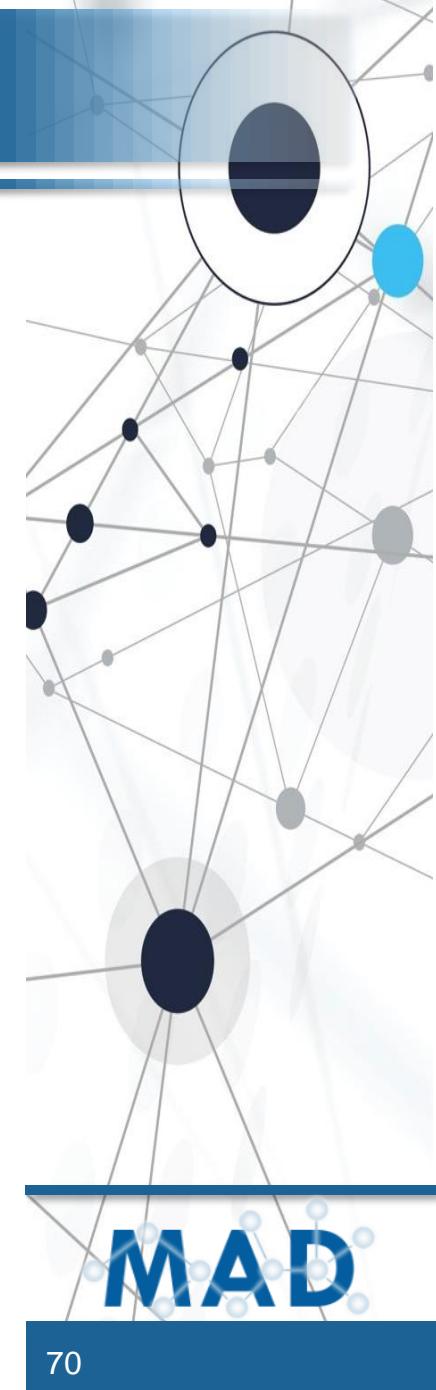
Al termine di questo algoritmo modificato, come nella versione precedente, avremo una CPT che conterrà le variabili della Query e le variabili della clique radice che andranno semplicemente eliminate tramite marginalizzazione.

In questa versione evitiamo di dover creare una copia del sotto albero e sarà sufficiente scorrere il JT salvandosi la CPT che si crea man mano senza ovviamente apportare alcuna modifica al JT. Inoltre risparmiamo anche il tempo necessario per scorrere nuovamente il sotto albero per il calcolo dei messaggi, anche perché questo non esisterà mai se non logicamente durante la visita del JT originale.



BIBLIOGRAFIA

- [1] NetworkX: <https://networkx.github.io/>
- [2] Numba: <http://numba.pydata.org/>
- [3] BNlearn: <https://www.bnlearn.com/>
- [4] Expert Systems and Probabilistic Network Models (E. Castillo, J.M. Gutierrez, A.S. Hadi)
- [5] Finding all cliques of an undirected graph (<https://doi.org/10.1145/362342.362367>)
- [6] Probabilistic Graphical Models, Principles and Techniques (D. Koller and N. Friedman)
- [7] gRain: <https://cran.r-project.org/web/packages/gRain/>



THANK YOU

*Cavenaghi Emanuele
Zanga Alessio*

MAD

