Intelligent Agents:
Web-mining Agents

Probabilistic Graphical Models

Lifted Inference

Tanya Braun

UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR INFORMATIONSSYSTEME

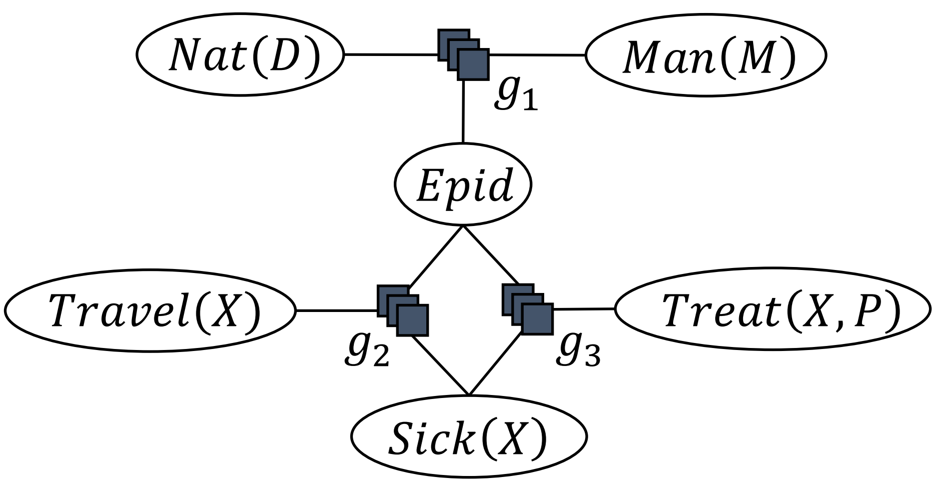
Probabilistic Graphical Models (PGMs)

1. Recap: **Propositional** modelling
   - Factor model, Bayesian network, Markov network
   - Semantics, inference tasks + algorithms + complexity
2. **Probabilistic relational models** (PRMs)
   - Parameterised models, Markov logic networks
   - Semantics, inference tasks
3. **Lifted inference**
   - LVE, LJT, FOKC
   - Theoretical analysis
4. **Lifted learning**
   - Recap: propositional learning
   - From ground to lifted models
   - Direct lifted learning
5. **Approximate Inference: Sampling**
   - Importance sampling
   - MCMC methods
6. **Sequential models & inference**
   - Dynamic PRMs
   - Semantics, inference tasks + algorithms + complexity, learning
7. **Decision making**
   - (Dynamic) Decision PRMs
   - Semantics, inference tasks + algorithms, learning
8. **Continuous Space**
   - Gaussian distributions and Bayesian networks
   - Probabilistic soft logic

Problem: Many Queries

- Set of queries
  - $P(Travel(eve))$
  - $P(Sick(bob))$
  - $P(Treat(eve, m_1))$
  - $P(Epid)$
  - $P(Nat(flood))$
  - $P(Man(virus))$
  - Combinations of variables
- Under evidence
  - $Sick(X') = true$
  - $X' \in \{alice, eve\}$

Build a helper structure to precompute parts



- LVE restarts with initial model for each query

Outline: 3. Lifted Inference

A. *Lifted variable elimination (LVE)*
  - Operators
  - Algorithm
  - Complexity (including first-order dtrees), completeness, tractability
  - Variants

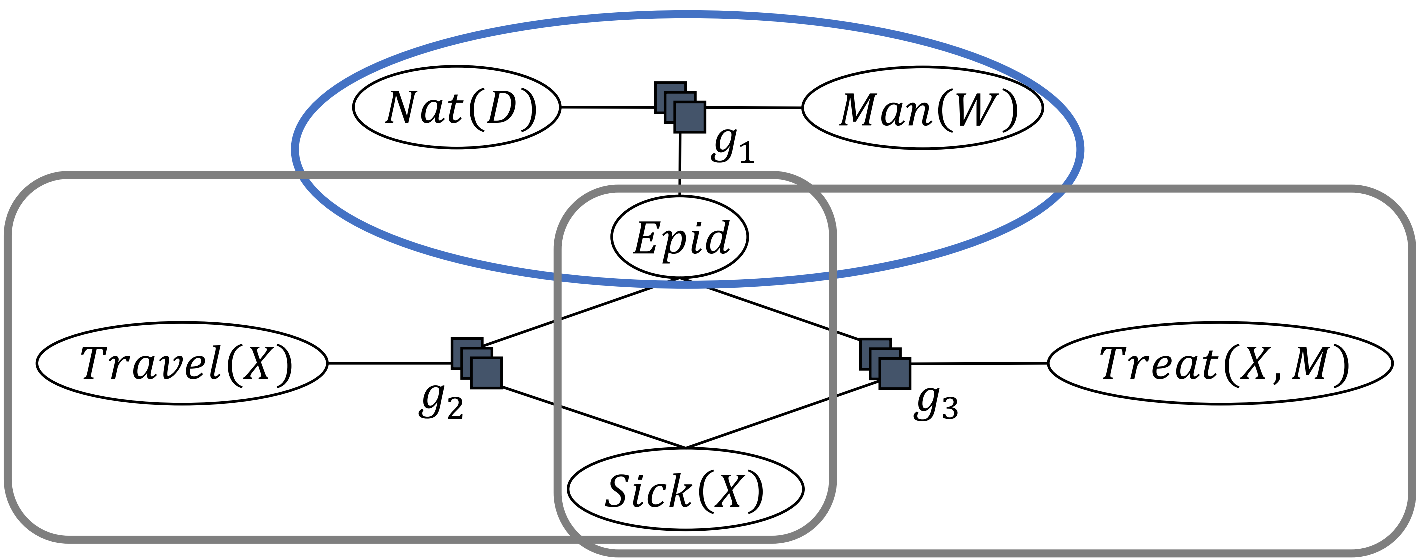
B. ***Lifted junction tree algorithm (LJT)***
  - First-order junction trees (FO jtrees)
  - Algorithm
  - Complexity, completeness
  - Variants

C. *First-order knowledge compilation (FOKC)*
  - Normal form, circuits
  - Algorithm
  - Complexity, completeness

D. *Most probable assignment queries*
  - Distribution vs. assignment queries
  - Most probable explanation (MPE) , Maximum-a-posteriori (MAP) assignments
  - Changes in LVE, LJT, FOKC
  - Complexity, completeness

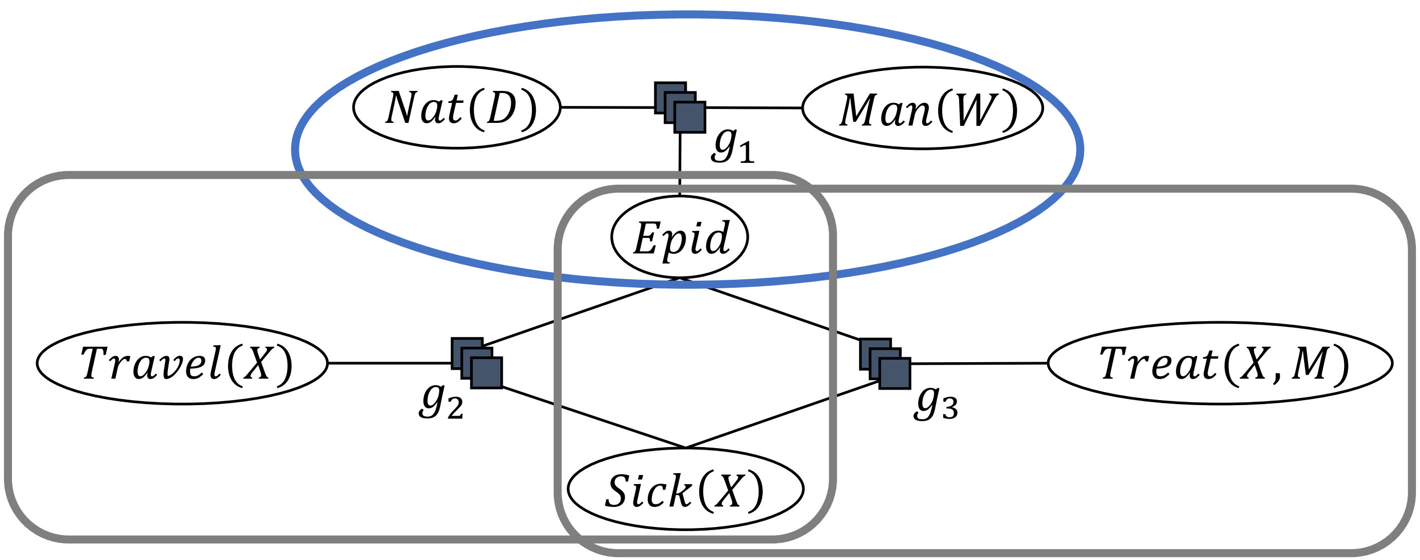
Clustering of Models

- Idea: Find subsets (clusters) of PRVs that are "enough" for certain queries
  - E.g.,
    - For queries about instances of $Nat(D), Man(W), Epid$
      - $Nat(D), Man(W), Epid$ are enough
    - For queries about instances of $Travel(X), Sick(X), Epid$
      - $Travel(X), Sick(X), Epid$ are enough
    - For queries about instances of $Treat(X, M), Sick(X), Epid$
      - $Treat(X, M), Sick(X), Epid$ are enough



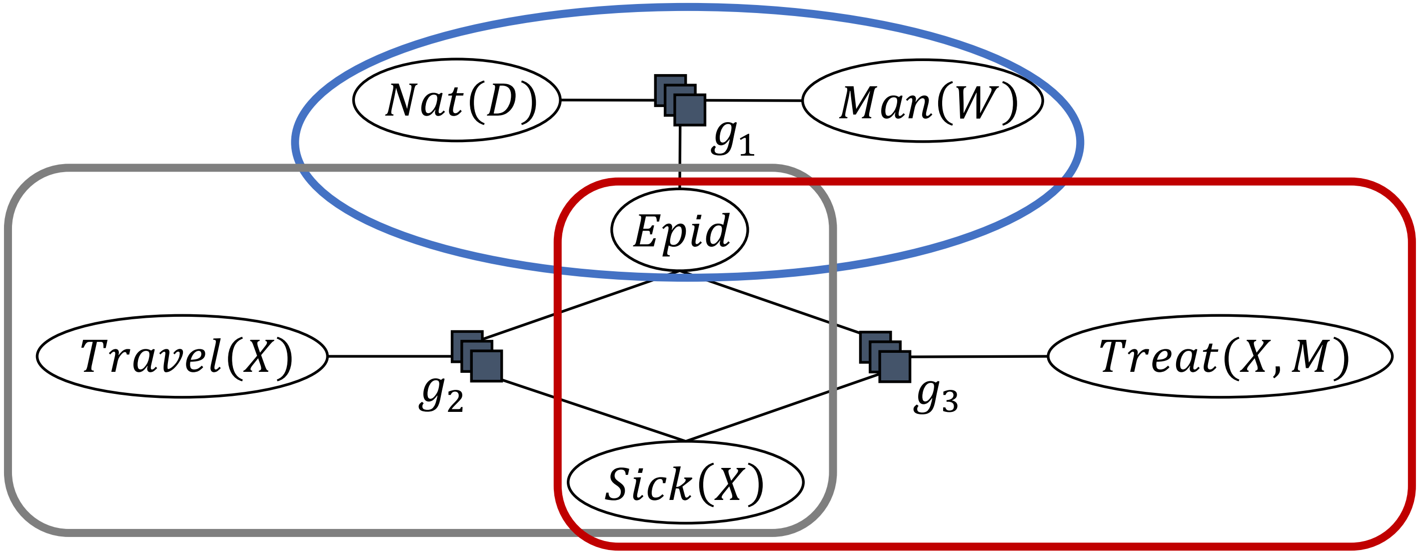
Clustering of Models

- But: If only parfactors used that contain the PRVs of a cluster, information stored in all other parfactors ignored
  - E.g.,
    - $Nat(D), Man(W), Epid$: g_1 → misses g_2, g_3
    - $Travel(X), Sick(X), Epid$: g_2 → misses g_1, g_3
    - $Treat(X, M), Sick(X), Epid$: g_3 → misses g_1, g_2
- Only correct if clusters are *independent* from each other
  - How can we achieve independence?



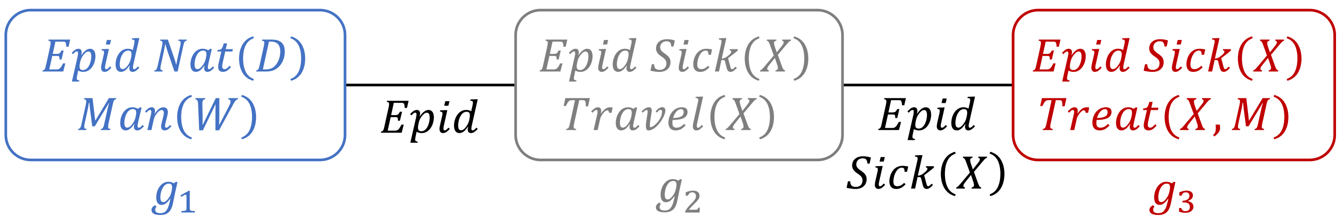
Clustering of Models

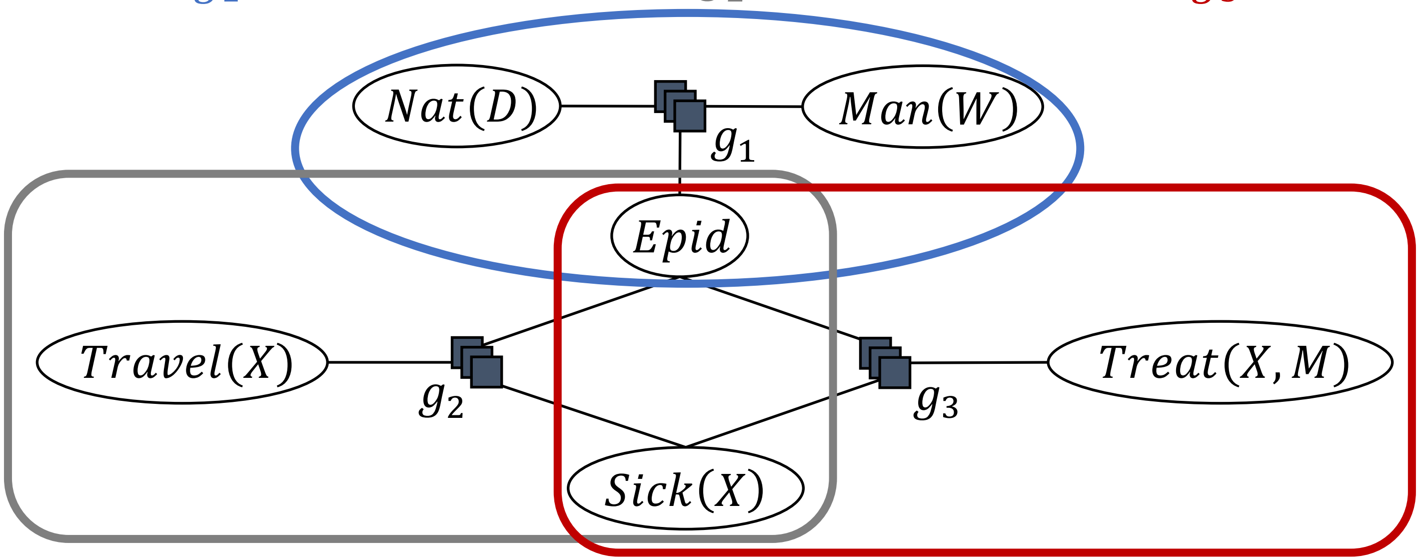
- Remember: Global Markov Property
  - Any two subsets of variables are conditionally independent given a separating subset
  - E.g.,
    - $Nat(D), Man(W), Epid: g_1$
      → independent of the rest given $Epid$
    - $Travel(X), Sick(X), Epid: g_2$
      → independent of the rest given $Epid, Sick(X)$
    - $Treat(X, M), Sick(X), Epid: g_3$
      → independent of the rest given $Epid, Sick(X)$



Clustering of Models

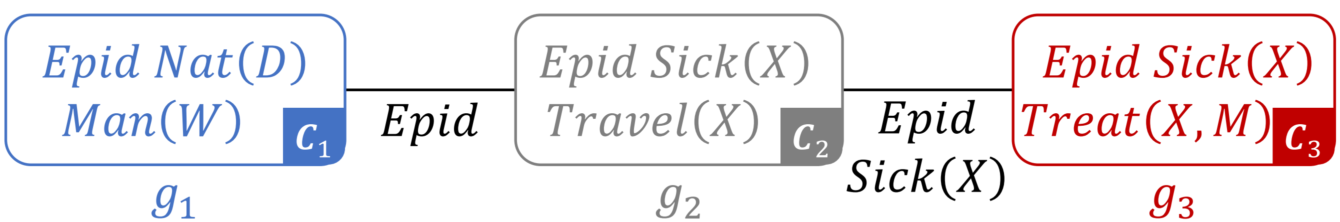
- Put clusters and their separators into a graph structure where
  - Nodes are clusters with parfactors assigned containing the cluster PRVs (*local model*)
  - Edges are labelled with the separator between neighbouring nodes
  - If two nodes contain the same PRV, every node on the path between the two nodes contain the PRV (*running intersection property*)





Clustering of Models

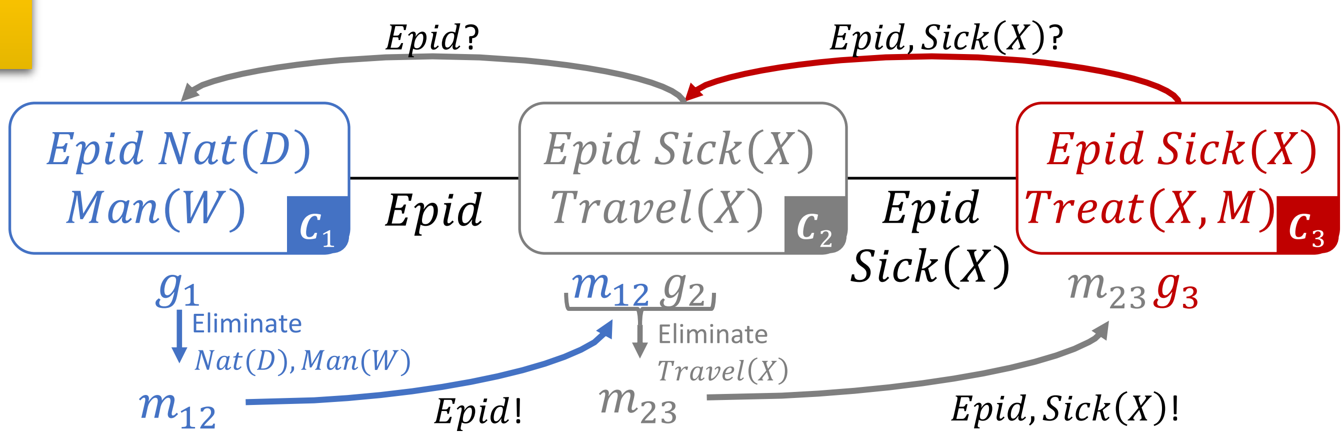
- Next: Make clusters actually independent of each other
  - Each cluster i asks its neighbours $j \in nbs(i)$ for information about the separator $\boldsymbol{S}_{ij}$ between them
    - Other clusters have to collect all the information from the model that lies behind the separator on its part, eliminate the non-separator PRVs from that information using LVE, and send the result in a message m_{ji}, i.e., a set of parfactors, back
  - Having the information on the separators to all neighbours makes a cluster independent from its neighbours and therefore all other parts of the model
    - Ensures that each cluster of PRVs has all model information needed available for query answering on instances of its cluster PRVs



Clustering of Models

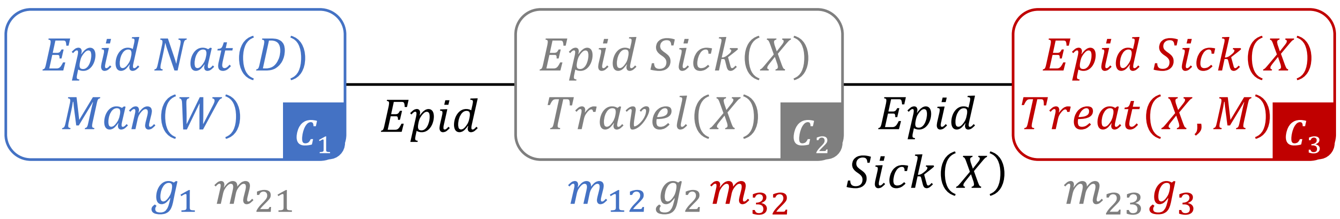
- Next: Make clusters actually independent of each other
  - E.g., $\boldsymbol{C}_3$: g_3 → independent of the rest given $Epid, Sick(X)$
    - Asks neighbour $\boldsymbol{C}_2$ for information on $Epid, Sick(X)$
      - $\boldsymbol{C}_2$ asks neighbour $\boldsymbol{C}_1$ for information on $Epid$
        - $\boldsymbol{C}_1$ sends information on $Epid$ in a message m_{12}
          - Eliminates $Nat(D), Man(M)$ from g_1 for m_{12}
      - $\boldsymbol{C}_2$ sends information on $Epid, Sick(X)$ to $\boldsymbol{C}_3$ in a message m_{23}
        - Eliminates $Travel(X)$ from g_2 and m_{12} for m_{23}
    - With m_{23}, $\boldsymbol{C}_3$ is independent from its neighbour $\boldsymbol{C}_2$ and therefore also from $\boldsymbol{C}_1$
      - As $\boldsymbol{C}_2$ is independent given m_{12} from $\boldsymbol{C}_1$

The same has to be done for $\boldsymbol{C}_2$ and $\boldsymbol{C}_1$



Clustering of Models

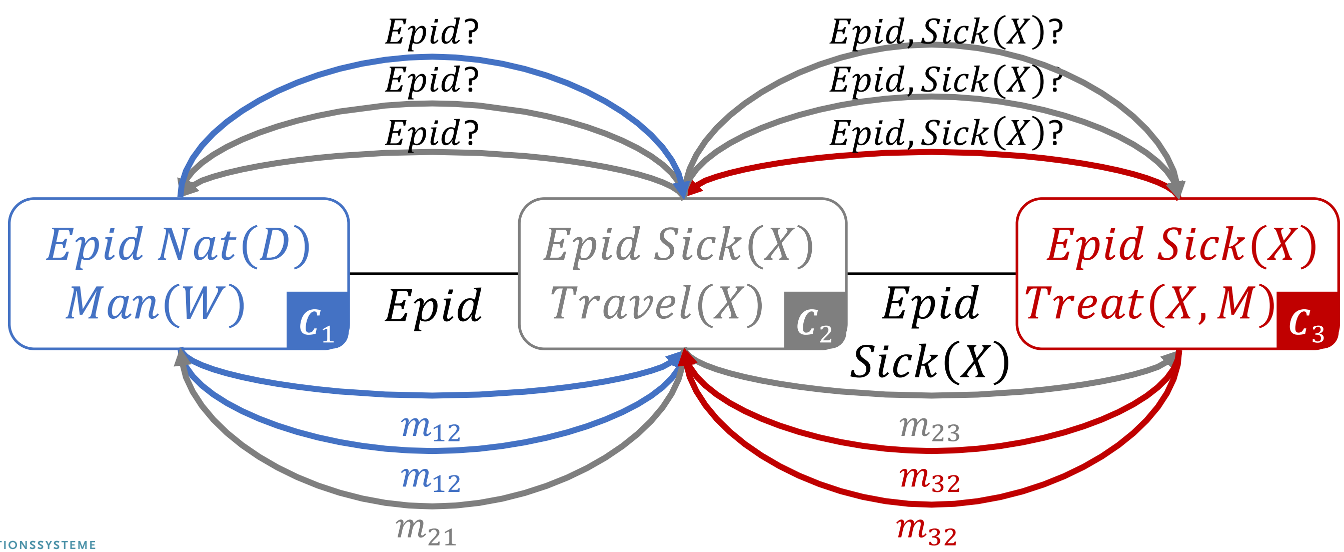
- With each cluster i independent of the rest, each i can answer queries about instances of its PRVs based on its local model and the messages received
  - Query terms: grounded instances or parameterised versions of its PRVs
    - Conjunctive queries if terms only concern the cluster PRVs
  - E.g., $\boldsymbol{C}_3 : g_3 \rightarrow$ independent of the rest given $Epid, Sick(X)$
    - Based on g_3 and m_{23}, $\boldsymbol{C}_3$ can answer queries about $Epid, Sick(X), Treat(X, M)$ such as
      $P\big(Sick(X)\big), P\big(Treat(eve, m_2)\big), P\big(Epid, Sick(alice)\big)$
      - Cannot answer any queries about $Nat(D), Man(W), Travel(X)$ but $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ can



Clustering of Models

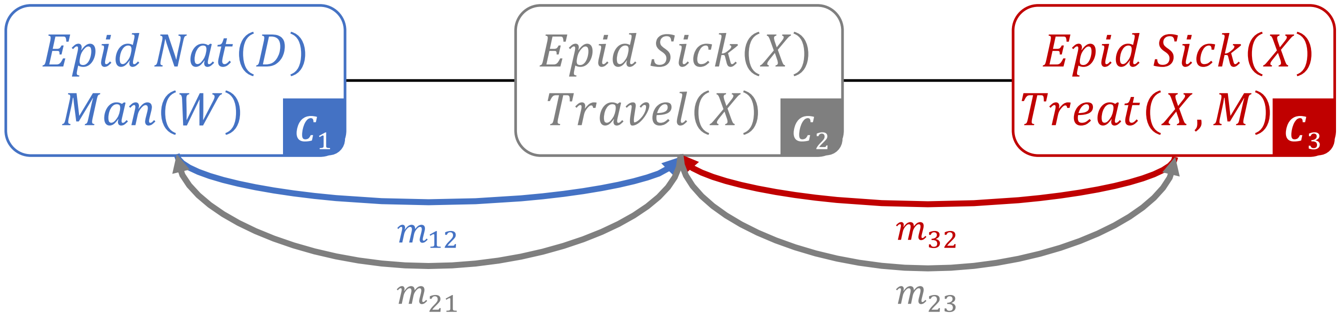
- Problem left: If each cluster asks for information on separators, some messages are sent multiple times
  - E.g.,
    - C_3 asks C_2, which asks C_1
      - Messages calculated and sent: m_{12}, m_{23}
    - C_2 asks C_1 and C_3
      - Messages calculated and sent: m_{12}, m_{32}
    - C_1 asks C_2, which asks C_3
      - Messages calculated and sent: m_{32}, m_{21}

Organise in way that messages are calculated only once



Clustering of Models

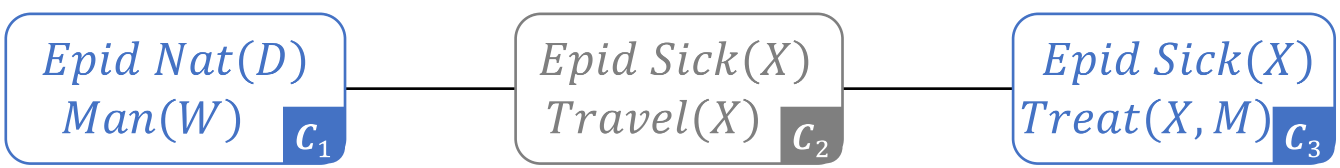
- Use dynamic programming to organise the order of asking or rather sending information in messages:
  - → If a node i has received all information from neighbours but one, j, node i sends a message with its information on the separator $\boldsymbol{S}_{ij}$ to j
  - → If a node i has received all messages, then it sends messages to all neighbours j that have not received a message yet
- When computing the message, i takes into consideration its local model as well as the messages received from all other neighbours but the receiving neighbour j



Clustering of Models

- Observations:
  → If a node i has received all information from neighbours but one, j, node i sends a message with its information on the separator $\boldsymbol{S}_{ij}$ to j
    - Trivially true at leaf nodes (*periphery*), can start sending immediately to its only neighbour (in *parallel*!)
      - From periphery inbound, new nodes trigger this first condition

  → If a node i has received all messages, then it sends messages to all neighbours j that have not received a message yet
    - As messages are sent further inwards, a first node at the centre will have received all messages and will start sending messages outbound, leading to new nodes triggering this second condition

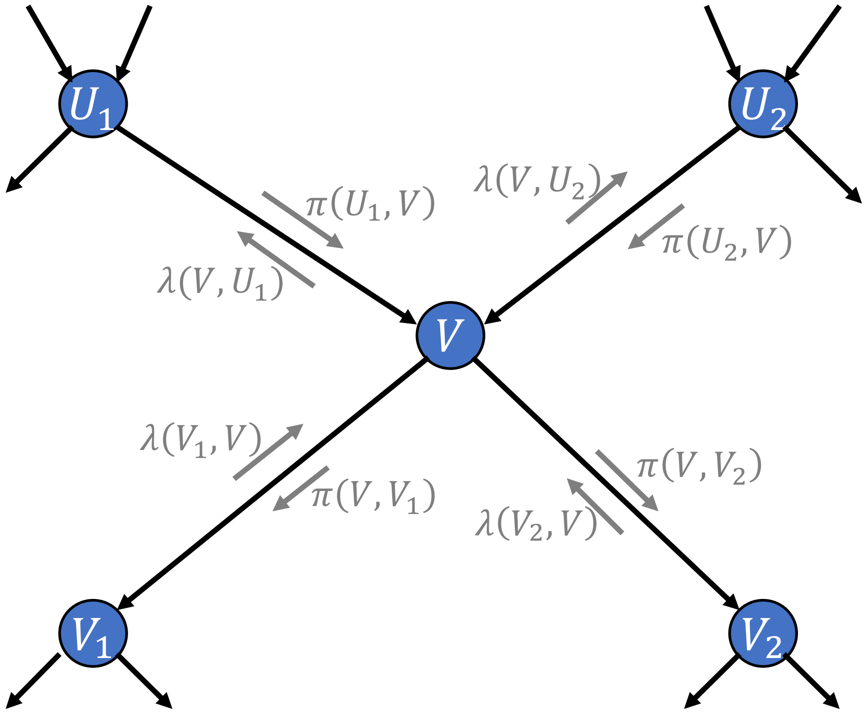


These two passes from periphery inbound and back suffice to distribute all information and make the clusters independent from each other*

* Shown by Steffen L. Lauritzen and David J. Spiegelhalter: Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems. In: *Journal of the Royal Statistical Society. Series B: Methodological*, 1988.

Foundations of Clustering

- History in propositional probabilistic inference:
  - Based on probability propagation introduced by Pearl (1988)



- If a BN is a polytree, i.e., the underlying undirected graph has no trivial cycles, then
  - Treat each node in a BN as a cluster with the random variables (randvars) of the accompanying CPT as the cluster randvars
  - Send messages along the edges (to parents and children), eliminating randvars not occurring in the parent or child nodes

Judea Pearl: Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach. In: *AAAI-82 Proceedings of the 2nd National Conference on Artificial Intelligence*, 1982.

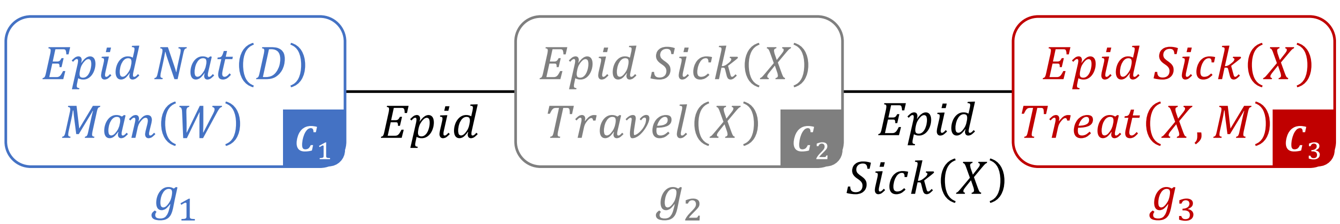
Foundations of Clustering

- History in propositional probabilistic inference:
  - If no polytree, the cycles mess up the message passing along the edges (information arrives multiple times)
    - Send messages nonetheless (exact if polytree, approximate otherwise): called belief propagation as an algorithm for *approximate* inference
    - Exact inference required → put the cycles into one cluster
    - Graph formed called a junction tree (jtree)
      - A first-order version of a jtree was induced on the previous slides
      - Also known as clique tree (since the cycles often form cliques in the model graph) or join tree
      - Propositional version introduced by Lauritzen and Spiegelhalter (1988)
      - Shenoy and Shafer (1989) introduce three axioms of *local computations* to show correctness of doing computations locally

Steffen L. Lauritzen and David J. Spiegelhalter: Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems. In: *Journal of the Royal Statistical Society. Series B: Methodological*, 1988.
Prakash P. Shenoy and Glenn R. Shafer: Axioms for Probability and Belief-Function Propagation. In: *Uncertainty in Artificial Intelligence 4*, 1990.

First-order Jtree (FO Jtree)

- As seen on the earlier slides
  - Acyclic graph
  - Nodes contain PRVs, which form clusters
  - Edges are based on the separators between the clusters
  - Nodes have parfactors assigned
- Next slides:
  - Formal definition
  - Construction
    - Get an *acyclic* structure with valid *separators* and each parfactor of a model assigned to a *local model*



Parameterised Clusters

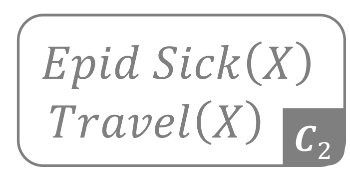
- Node of an FO jtree:
  Set of PRVs called parameterised cluster (parcluster)
- Let $\boldsymbol{X}$ be a set of logvars, $\boldsymbol{A}$ a set of PRVs with $lv(\boldsymbol{A}) \subseteq \boldsymbol{X}$, and $(\mathcal{X}, C_{\mathcal{X}})$ a constraint on $\boldsymbol{X}$ with $\mathcal{X}$ being a sequence of the logvars of $\boldsymbol{X}$
- Then, a parcluster $\boldsymbol{C}$ is given by

$$\forall \boldsymbol{x} \in C_{\mathcal{X}} : \boldsymbol{A}_{|(\mathcal{X}, C_{\mathcal{X}})}$$

  - $\boldsymbol{A}_{|(\mathcal{X}, C_{\mathcal{X}})}$ for short
  - Again, $(\mathcal{X}, C_{\mathcal{X}})$ can be omitted if $\top$ constraint encoded
  - Depicted as a round shape containing $\boldsymbol{A}$ or just $\boldsymbol{A}$
    - Again, constraint usually not depicted
  - E.g., parcluster $\boldsymbol{C}_2$

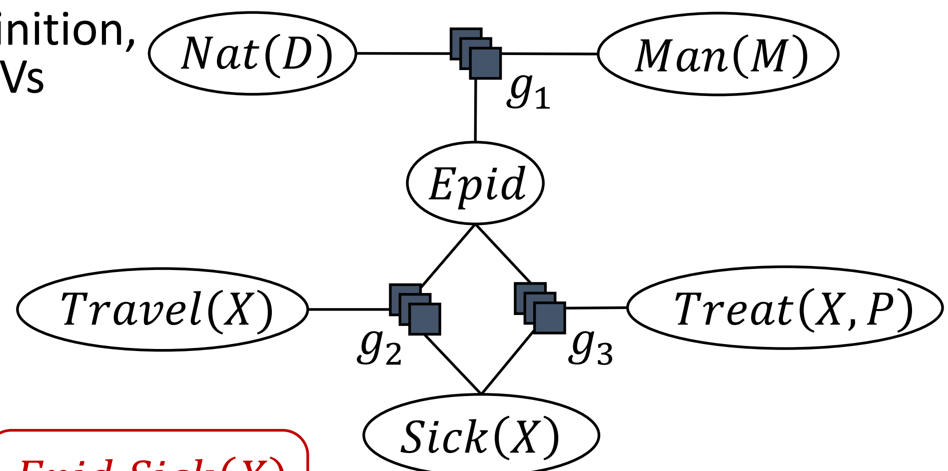
$$\forall x \in \mathcal{D}(X) : \{Epid, Sick(x), Travel(x)\}_{|(X, \mathcal{D}(X))}$$
$$= \{Epid, Sick(X), Travel(X)\}_{|(X, \mathcal{D}(X))}$$
$$= \{Epid, Sick(X), Travel(X)\}$$

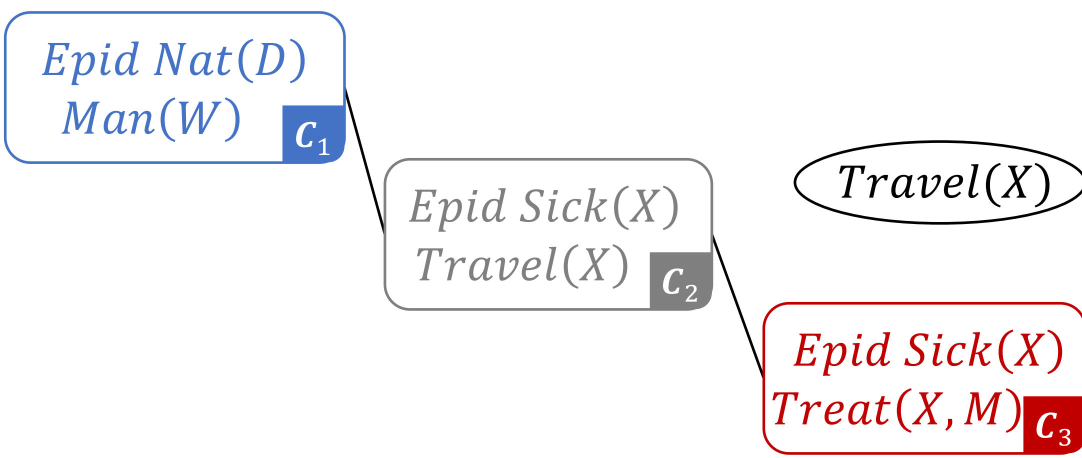
$Epid\ Sick(X)$
$Travel(X)$



FO Jtree

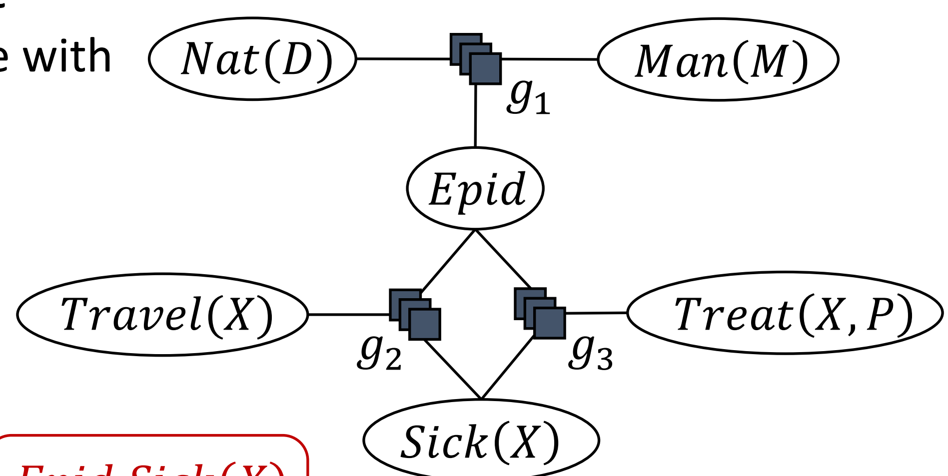
- An FO jtree J for a model G is a cycle-free graph (V, E)
  - Set of nodes $V \subseteq 2^{rv(G)}$
    - I.e., nodes are sets of PRVs (parclusters)
    - $2^{rv(G)}$ denotes the power set of $rv(G)$
  - Set of edges $E \subseteq \{\{i,j\} \mid i,j \in V, i \neq j\}$,
    - Has to be cycle free, which includes no self-loops
  - E.g., as depicted on the left
    - But at this point in the definition, could be any subsets of PRVs

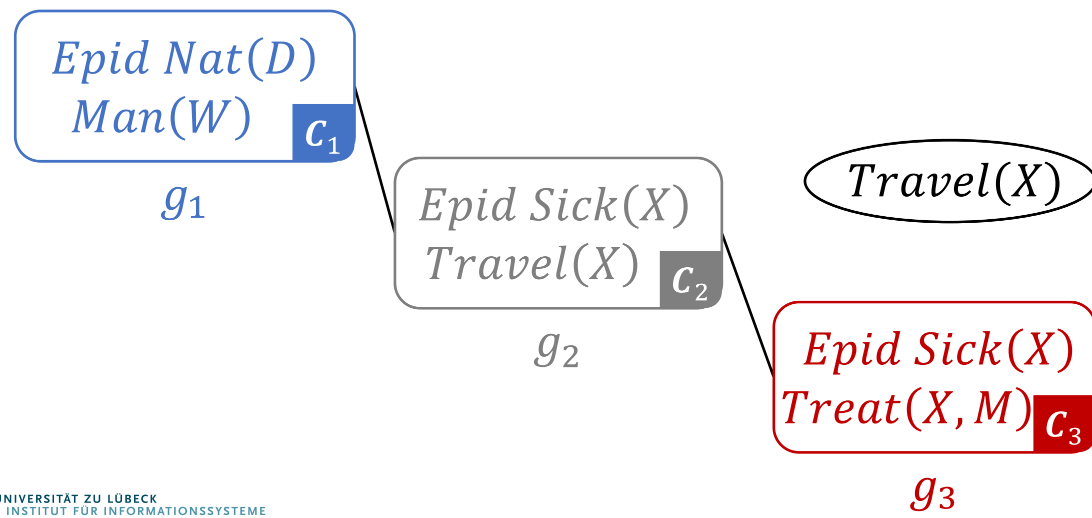




FO Jtree

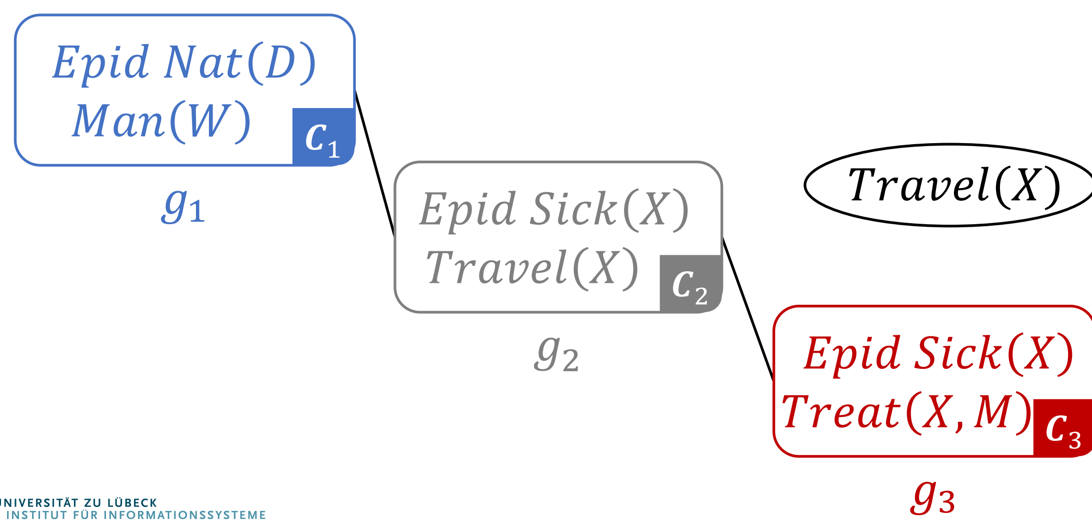
- An FO jtree J for a model G is a cycle-free graph (V, E)
  - Has to satisfy three properties:
    1. $\forall \boldsymbol{C} \in V : \boldsymbol{C} \subseteq rv(G)$
    2. $\forall g \in G : \exists \boldsymbol{C} \in V : rv(g) \subseteq \boldsymbol{C}$
    3. If $\exists A \in rv(G) : A \in \boldsymbol{C}_i \wedge A \in \boldsymbol{C}_j$ with $\boldsymbol{C}_i, \boldsymbol{C}_j \in V$, then $\forall \boldsymbol{C}_k \in V$ on the path between $\boldsymbol{C}_i, \boldsymbol{C}_j : A \in \boldsymbol{C}_k$ (running intersection property)
  - E.g., as depicted on the left
    - Only the following and one with $\boldsymbol{C}_3$ at the centre are valid

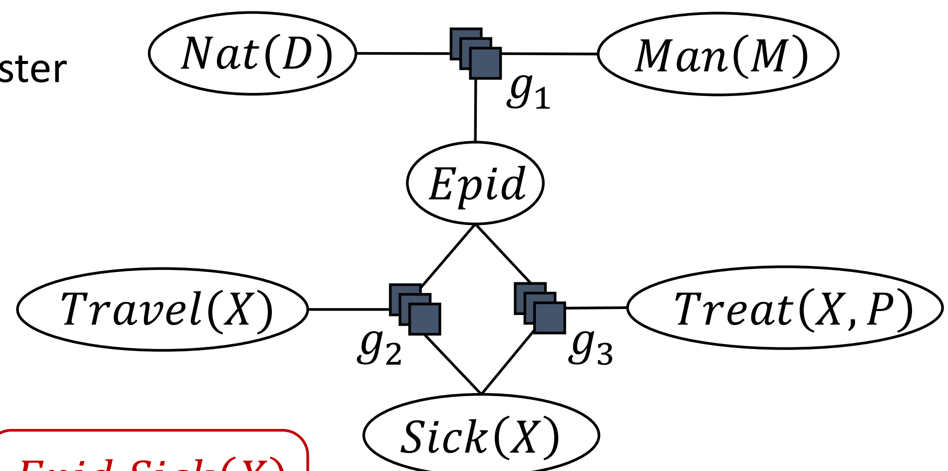




FO Jtree

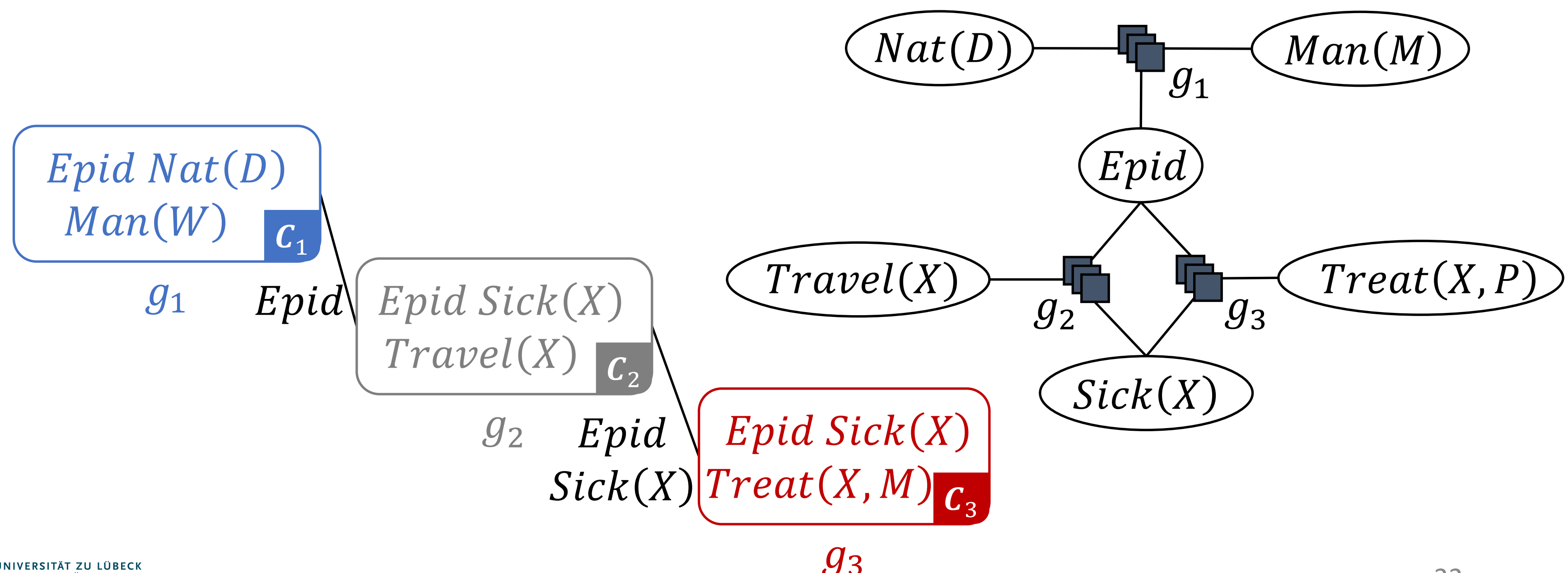
- An FO jtree J for a model G is a cycle-free graph (V, E)
  - Is minimal if by removing a PRV from a parcluster, the FO jtree ceases to be an FO jtree, i.e., no longer fulfils at least one property
  - E.g., depicted on the left
    - Cannot remove any PRV from any parcluster
      - Otherwise, a parfactor would no longer have its arguments in one parcluster





FO Jtree

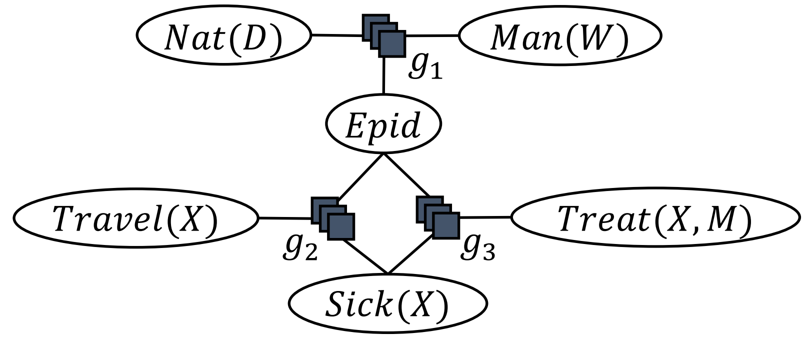
- An FO jtree J for a model G is a cycle-free graph (V, E)
  - Set $\boldsymbol{S}_{ij}$ called separator of edge $\{i, j\} \in E$, defined by
    $$\boldsymbol{S}_{ij} = \boldsymbol{C}_i \cap \boldsymbol{C}_j$$
  - Term $nbs(i)$ refers to the neighbours of $\boldsymbol{C}_i$, defined by
    $$nbs(i) = \{j \mid \{i, j\} \in E\}$$
  - Each $\boldsymbol{C}_i$ has a local model G_i and $\forall g \in G_i : rv(g) \subseteq \boldsymbol{C}_i$
    - Local models G_i partition G, i.e., $G = \bigcup_{i \in V} G_i$

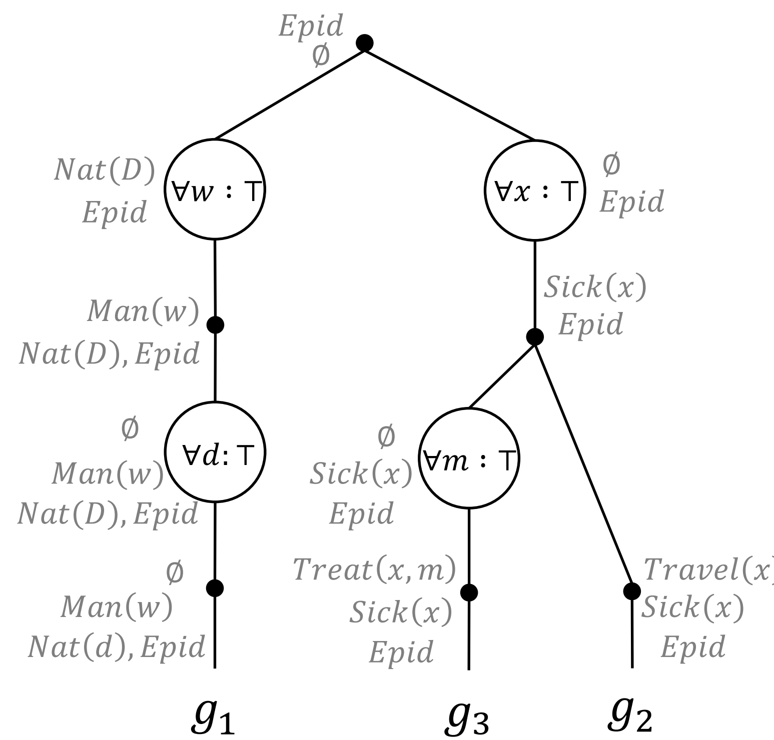


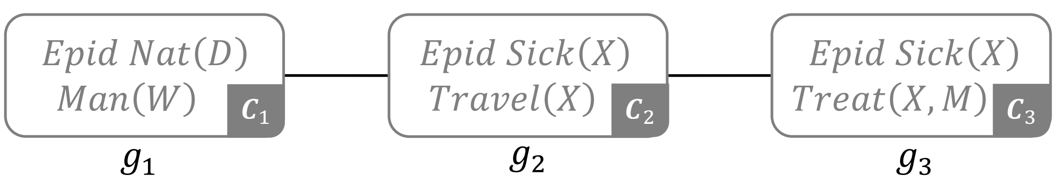
Construction

- Where do we get the FO jtree from s.t. the jtree
  - is acyclic
  - fulfils the three FO jtree properties
  - has the model parfactors automatically assigned to fitting parclusters?

→Clusters of an FO dtree
+ undirected dtree edges
+ minimisation
= FO jtree



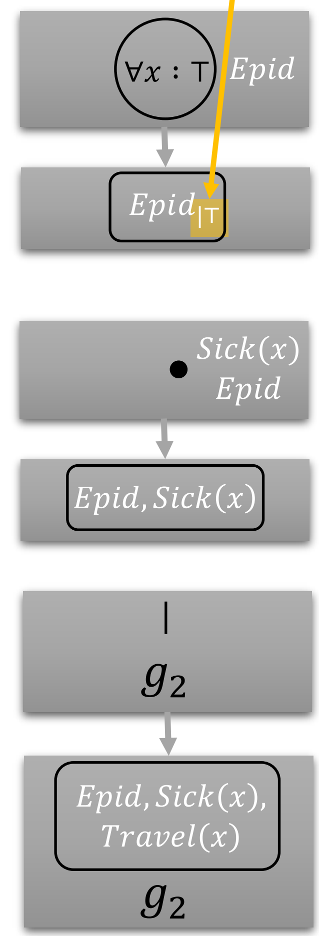




Clusters → Parclusters

- Given an FO dtree T for a model G with clusters for each node
- Given a cluster $\{A_1, \dots, A_n\}$ of a DPG node (X, x, C)
  - Resulting parcluster $\boldsymbol{C}_j = \{A_1, \dots, A_n\}_{|C}$
    - Local model $G_j = \emptyset$
- Given a cluster $\{A_1, \dots, A_n\}$ of a VE node
  - Resulting parcluster $\boldsymbol{C}_j = \{A_1, \dots, A_n\}_{|\top}$
    - Local model $G_j = \emptyset$
- Given a cluster $\{A_1, \dots, A_n\}$ from a leaf node with parfactor g_i
  - Resulting parcluster $\boldsymbol{C}_j = \{A_1, \dots, A_n\}_{|\top}$
    - Local model $G_j = \{g_i\}$

Let's carry the constraint around for a bit to make it explicit

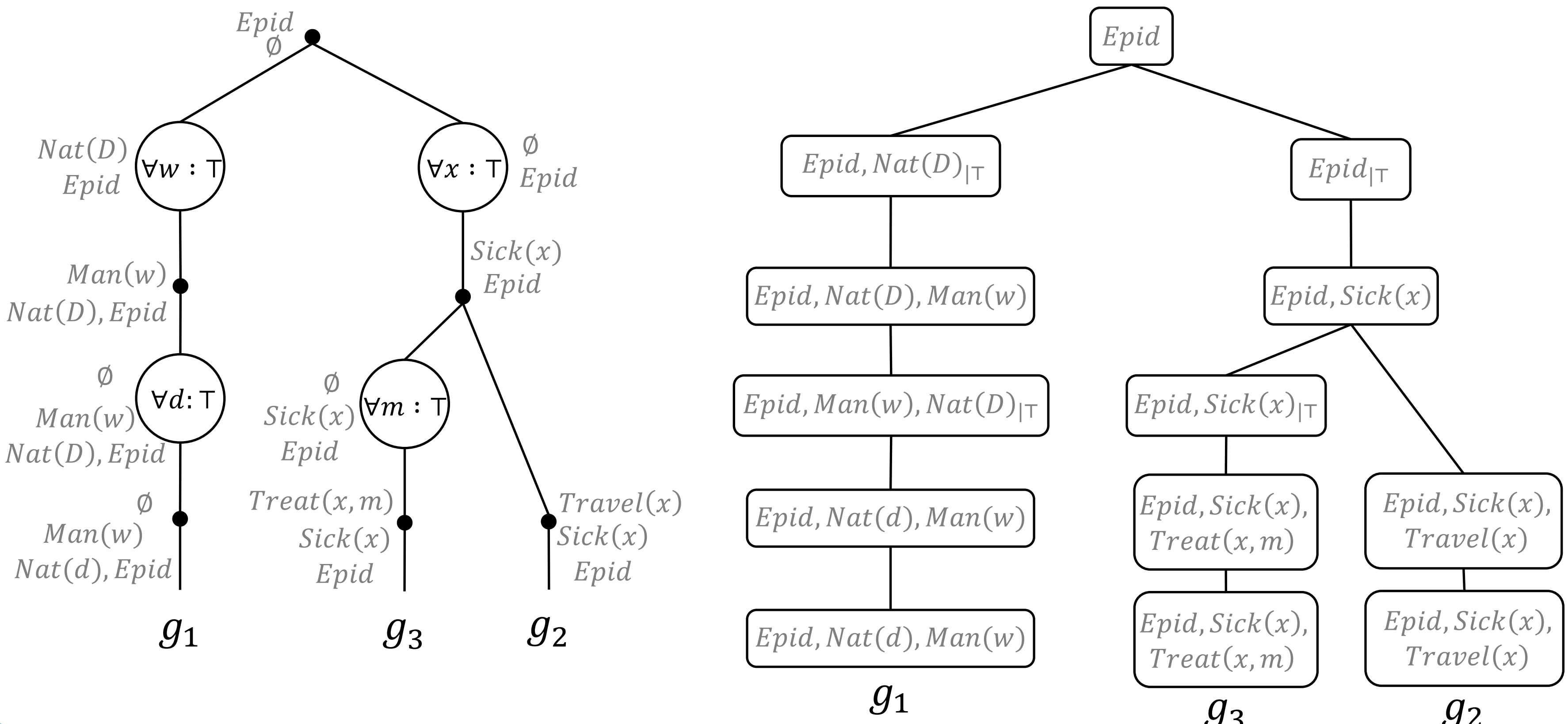


FO Dtree → FO Jtree

- Forming an FO jtree J from an FO dtree T of a model G
- Nodes of J
  - Parclusters resulting from clusters of T as shown on previous slide
    - Each parcluster has a source node in T
- Edges of J
  - Add an edge between two parclusters whenever there is an edge between the source nodes of the two parclusters in T

FO Dtree → FO Jtree

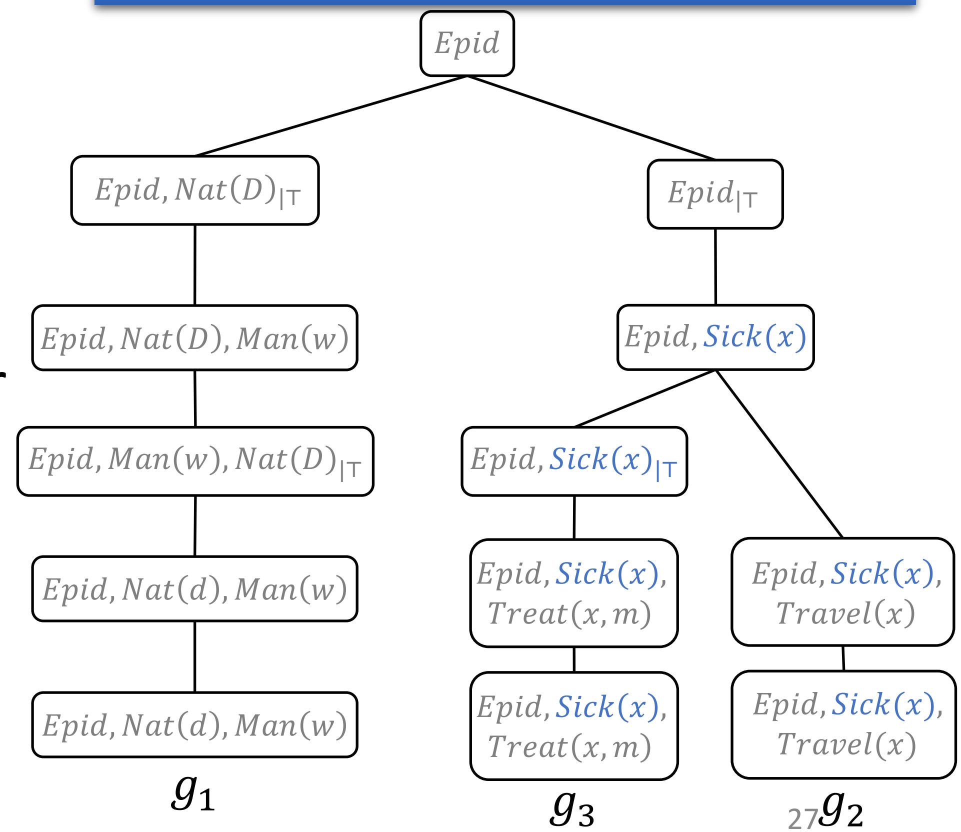
- Result after transformation
  - Fulfils the three jtree properties
  - But is not minimal



FO Dtree → FO Jtree

- Result after transformation fulfils the three jtree properties
- Hold by construction
  1. Parclusters can only contain model PRVs
  2. Each parfactor occurs at a dtree leaf, which is turned into a parcluster
  3. Based on how cutset/ context are calculated*
     - E.g., $Sick(X)$

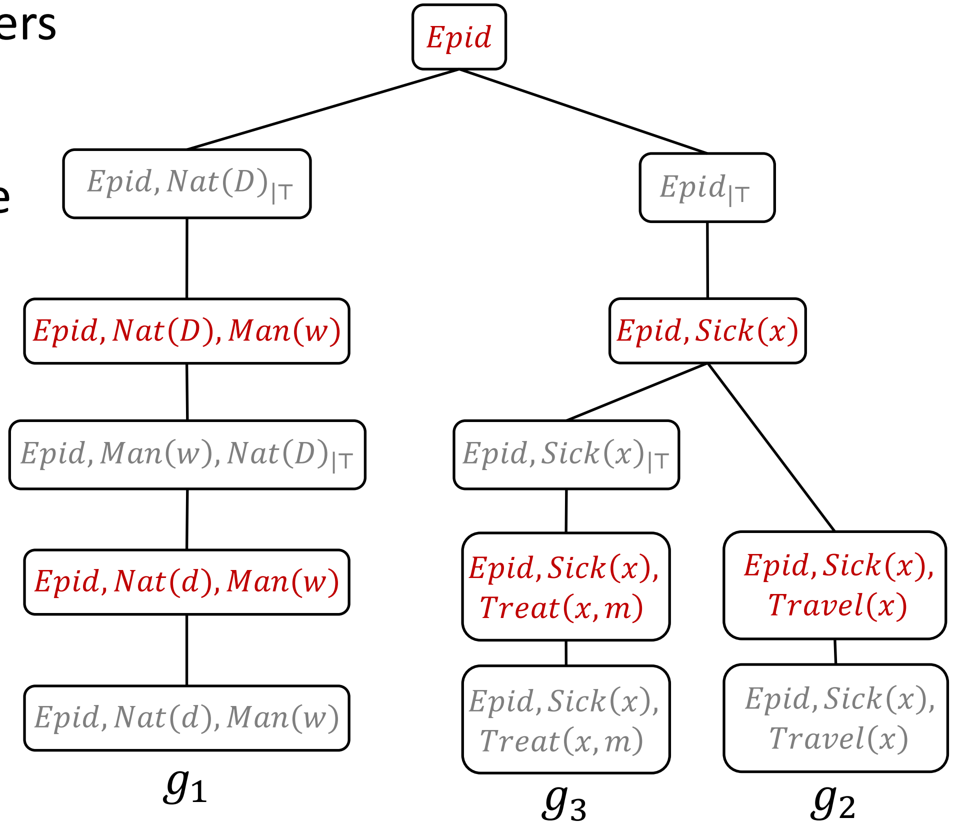
1. $\forall \boldsymbol{C} \in V : \boldsymbol{C} \subseteq rv(G)$
2. $\forall g \in G : \exists \boldsymbol{C} \in V : rv(g) \subseteq \boldsymbol{C}$
3. If $\exists A \in rv(G) : A \in \boldsymbol{C}_i \wedge A \in \boldsymbol{C}_j$ with $\boldsymbol{C}_i, \boldsymbol{C}_j \in V$, then $\forall \boldsymbol{C}_k \in V$ on the path between $\boldsymbol{C}_i, \boldsymbol{C}_j : A \in \boldsymbol{C}_k$



* Proof for jtrees: Adnan Darwiche: Recursive Conditioning. In: *Artificial Intelligence*, 2001.
Proof for FO jtrees: Tanya B: Rescued from a Sea of Queries: Exact Inference in Probabilistic Relational Models. PhD thesis, 2020.

FO Dtree → FO Jtree

- Result after transformation not minimal
  - Can remove complete parclusters and still have an FO jtree
    - Even if we keep parclusters that carry constraint information that we would otherwise lose
  - E.g.,
    - Parclusters marked
- Observation
  - Parclusters are subsets of other parclusters
    - Use for minimisation



UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR INFORMATIONSSYSTEME

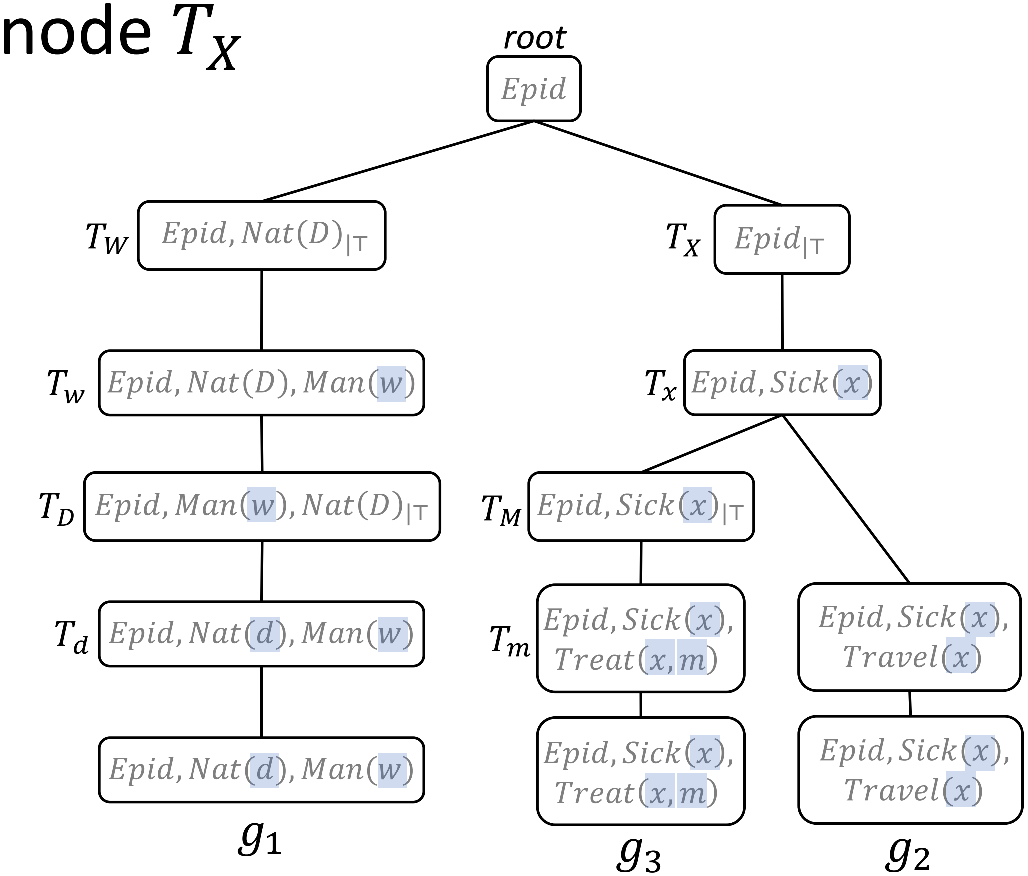
Minimisation

- Merge parclusters $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ with local models G_i and G_j iff
$$gr(\boldsymbol{C}_i) \subseteq gr(\boldsymbol{C}_j) \vee gr(\boldsymbol{C}_j) \subseteq gr(\boldsymbol{C}_i)$$
  - Assuming ⊤ constraints and same logvar names if the same domain is referenced (from normal form of FO dtree), then the following suffices:
$$\boldsymbol{C}_i \subseteq \boldsymbol{C}_j \vee \boldsymbol{C}_j \subseteq \boldsymbol{C}_i$$
    - Checking on a PRV and logvar level instead of a grounded level

Minimisation

- Pre-processing necessary:
  - Parclusters may contain a logvar X or a representative x
- For each source DPG node T_X
  - Apply the inverse substitution θ^{-1} to the one applied during FO dtree construction to all parclusters that come from descendants of T_X :

$$\theta^{-1} = \{X \to x\}^{-1} = \{x \to X\}$$

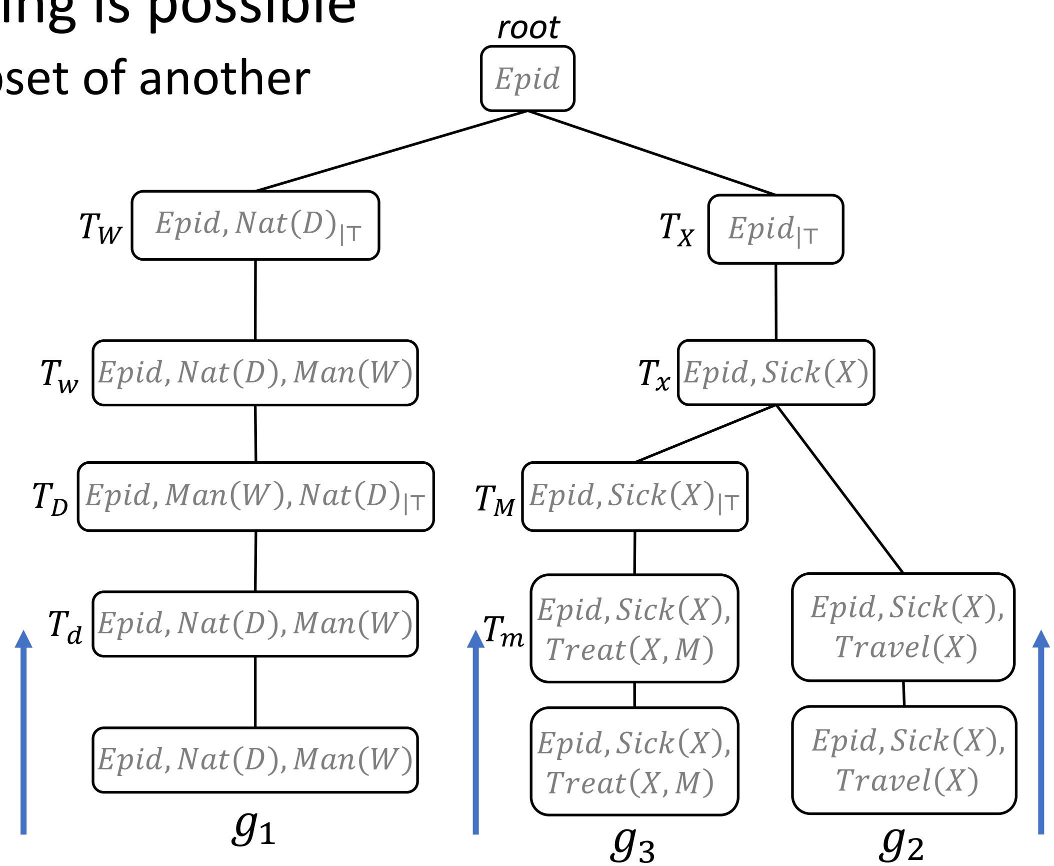


Minimisation

- Merging parclusters $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ into parcluster $\boldsymbol{C}_k$
  - $\boldsymbol{C}_k = \boldsymbol{C}_i \cup \boldsymbol{C}_j$
  - $G_k = G_i \cup G_j$
- Changes in FO jtree (V, E)
  - $V = V \setminus \{\boldsymbol{C}_i, \boldsymbol{C}_j\} \cup \boldsymbol{C}_k$
  - $E = E \setminus \{\{i, l\} \mid l \in nbs(i)\} \setminus \{\{j, l\} \mid l \in nbs(j)\}$
    $\cup \{\{k, l\} \mid l \in nbs(i) \lor l \in nbs(j), l \neq i, l \neq j\}$

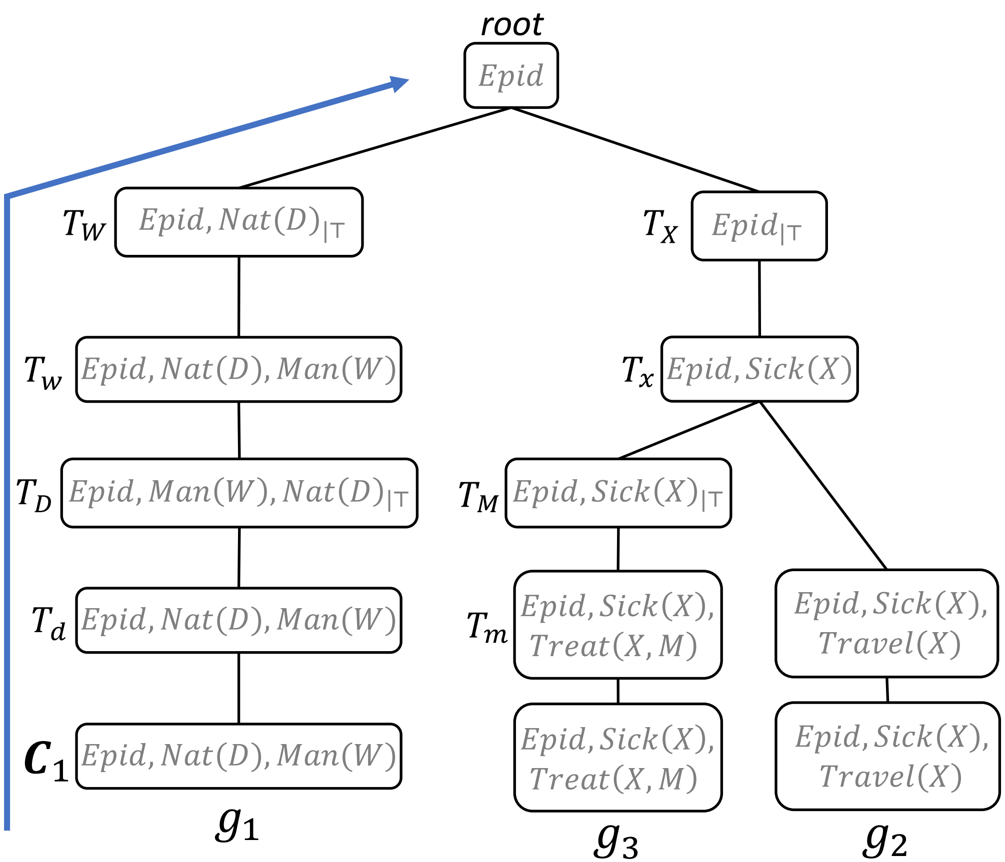
Minimisation

- Possible merging strategy
  - Start at the leaves and merge inbound
  - Until no further merging is possible
    - No parcluster is a subset of another
- After merging, the resulting FO jtree is minimal
- E.g.,
  - Start at leaves with
    - local model $\{g_1\}$
    - local model $\{g_2\}$
    - local model $\{g_3\}$



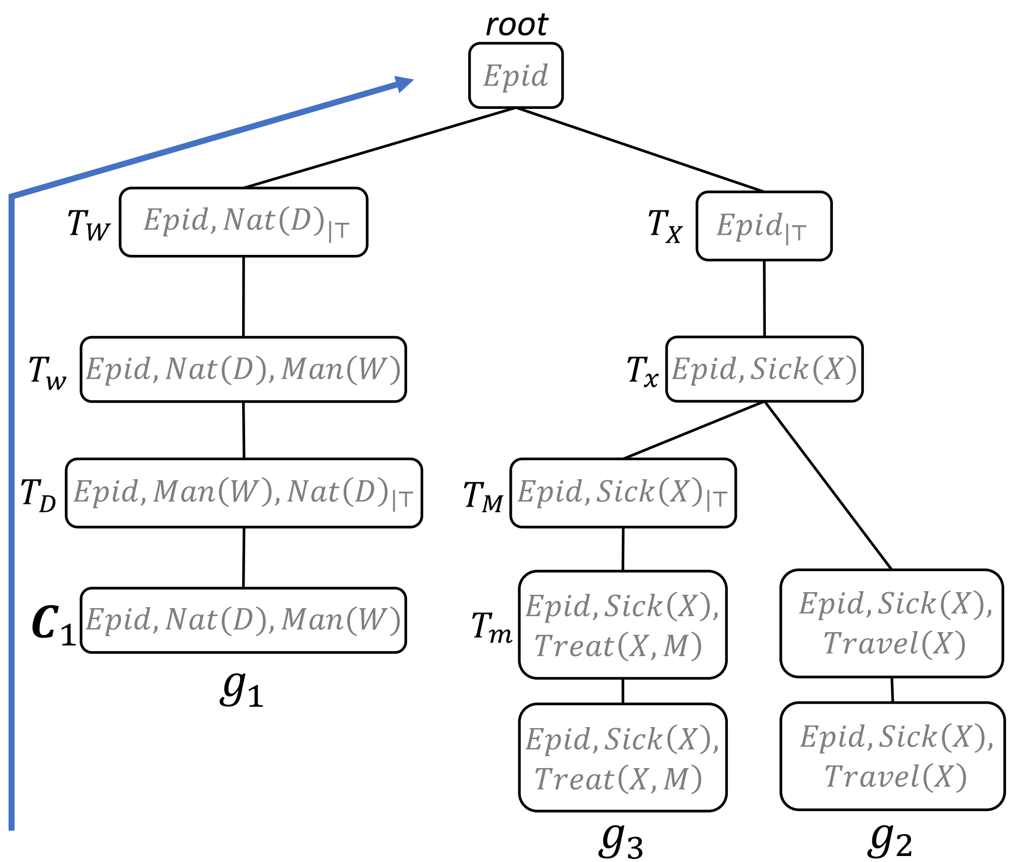
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_1\}$
  - Let us call it $\boldsymbol{C}_1$
  - Merge inbound
    - $\boldsymbol{C}_1$ and T_d parcluster identical → merge (call result $\boldsymbol{C}_1$ again)



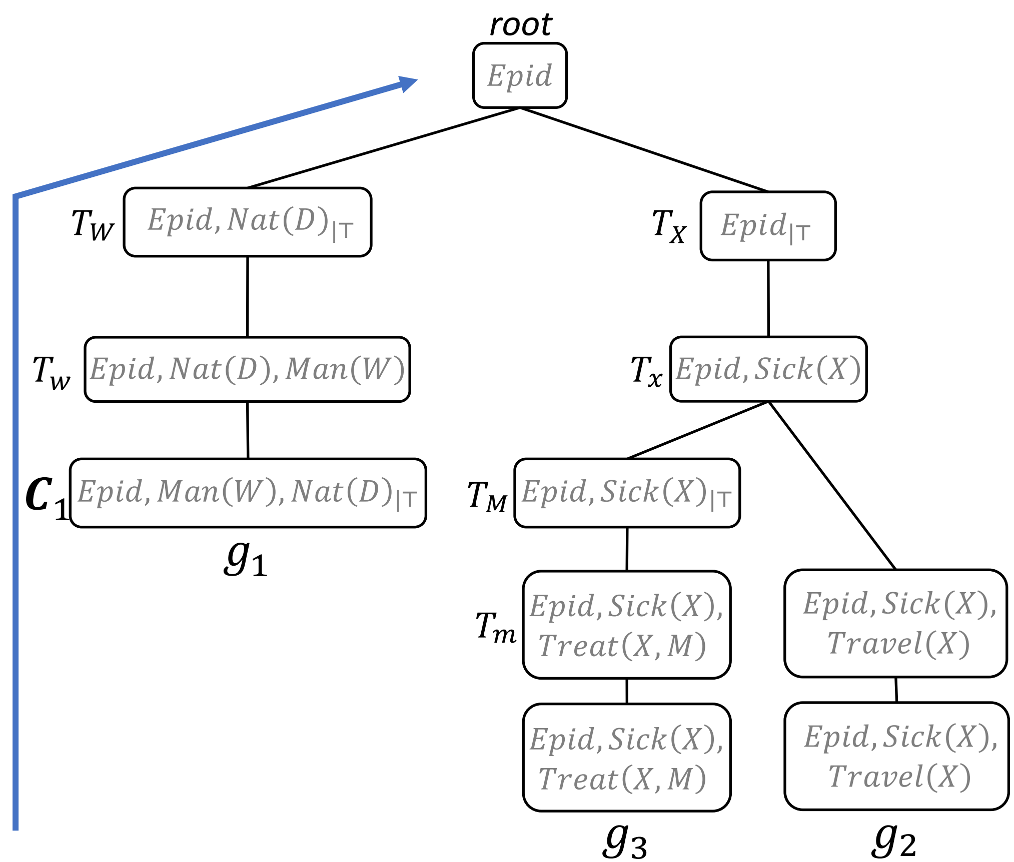
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_1\}$
  - Let us call it $\boldsymbol{C}_1$
  - Merge inbound
    - $\boldsymbol{C}_1$ and T_d parcluster identical → merge (call result $\boldsymbol{C}_1$ again)
    - $\boldsymbol{C}_1$ and T_D parcluster identical → merge



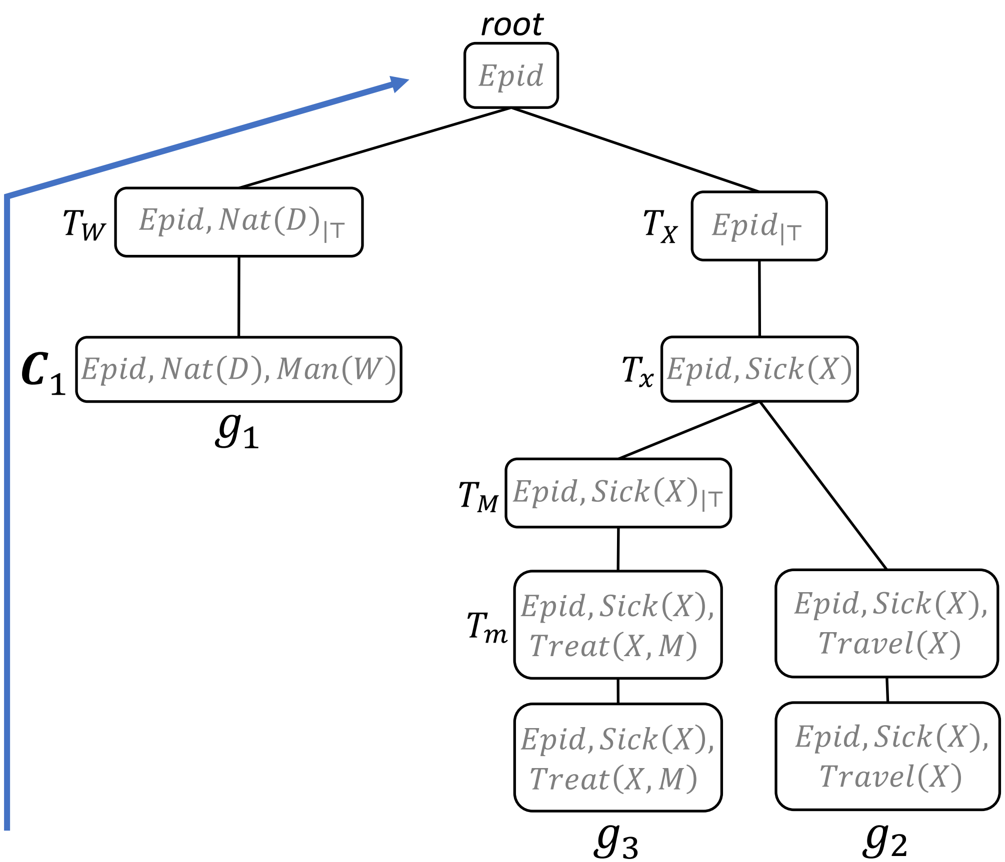
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_1\}$
  - Let us call it $\boldsymbol{C}_1$
  - Merge inbound
    - $\boldsymbol{C}_1$ and T_d parcluster identical → merge (call result $\boldsymbol{C}_1$ again)
    - $\boldsymbol{C}_1$ and T_D parcluster identical → merge
    - $\boldsymbol{C}_1$ and T_W parcluster identical → merge



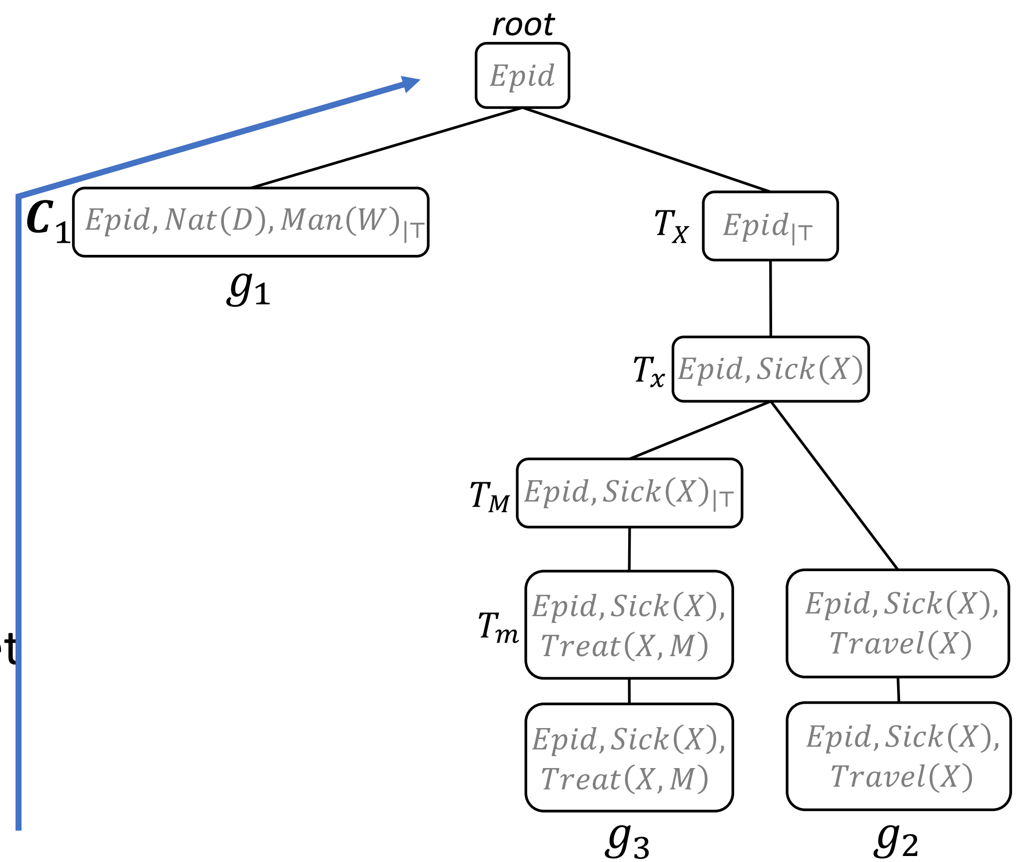
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_1\}$
  - Let us call it $\boldsymbol{C}_1$
  - Merge inbound
    - $\boldsymbol{C}_1$ and T_d parcluster identical → merge (call result $\boldsymbol{C}_1$ again)
    - $\boldsymbol{C}_1$ and T_D parcluster identical → merge
    - $\boldsymbol{C}_1$ and T_w parcluster identical → merge
    - T_W parcluster subset of $\boldsymbol{C}_1$ → merge



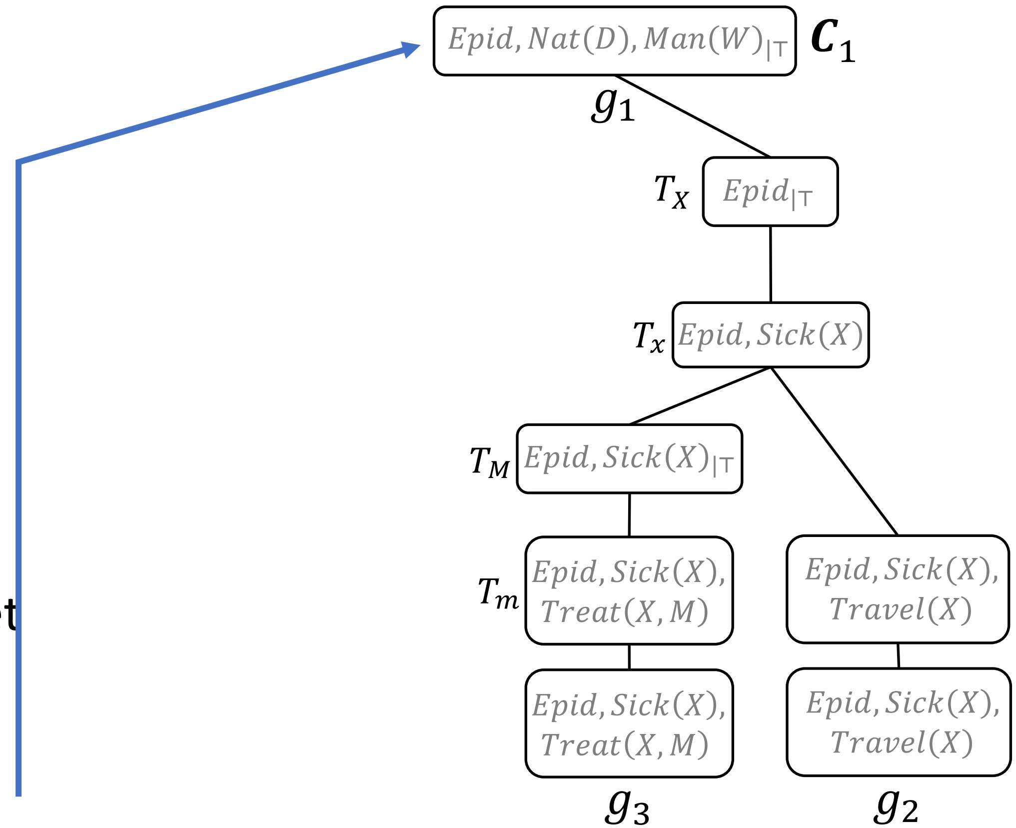
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_1\}$
  - Let us call it $\boldsymbol{C}_1$
  - Merge inbound
    - $\boldsymbol{C}_1$ and T_d parcluster identical → merge (call result $\boldsymbol{C}_1$ again)
    - $\boldsymbol{C}_1$ and T_D parcluster identical → merge
    - $\boldsymbol{C}_1$ and T_w parcluster identical → merge
    - T_W parcluster subset of $\boldsymbol{C}_1$ → merge
    - Root parcluster subset of $\boldsymbol{C}_1$ → merge



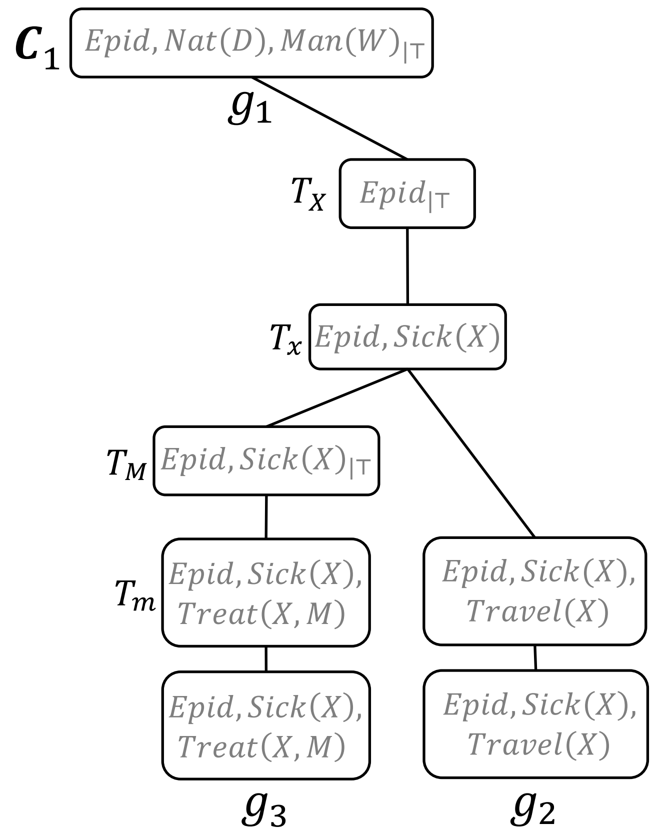
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_1\}$
  - Let us call it $\boldsymbol{C}_1$
  - Merge inbound
    - $\boldsymbol{C}_1$ and T_d parcluster identical → merge (call result $\boldsymbol{C}_1$ again)
    - $\boldsymbol{C}_1$ and T_D parcluster identical → merge
    - $\boldsymbol{C}_1$ and T_w parcluster identical → merge
    - T_W parcluster subset of $\boldsymbol{C}_1$ → merge
    - Root parcluster subset of $\boldsymbol{C}_1$ → merge



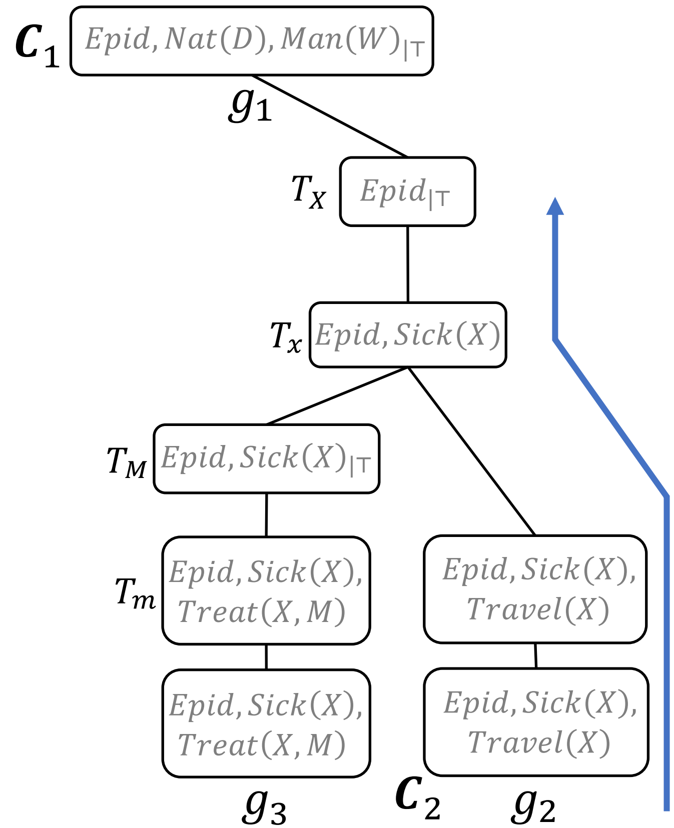
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_1\}$
  - Let us call it $\boldsymbol{C}_1$
  - At this point, we have reached the former root and cannot merge further inbound
    - Also: the T_X parcluster contains logvar X, which is not a subset or superset of the logvars of $\boldsymbol{C}_1$ (D, W)
  - Merging stops



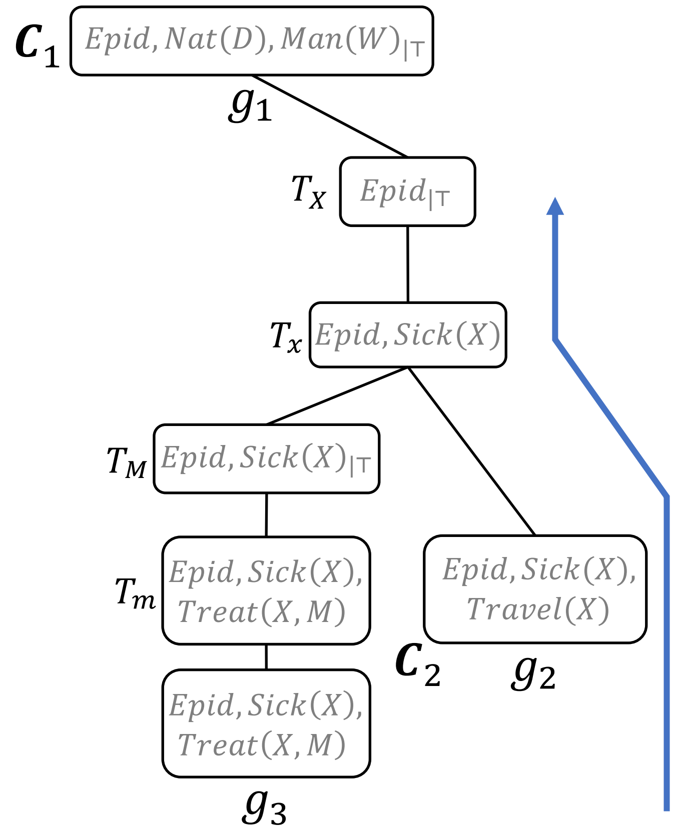
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_2\}$
  - Let us call it $\boldsymbol{C}_2$
  - Merge inbound
    - $\boldsymbol{C}_2$ and neighbouring parcluster identical → merge



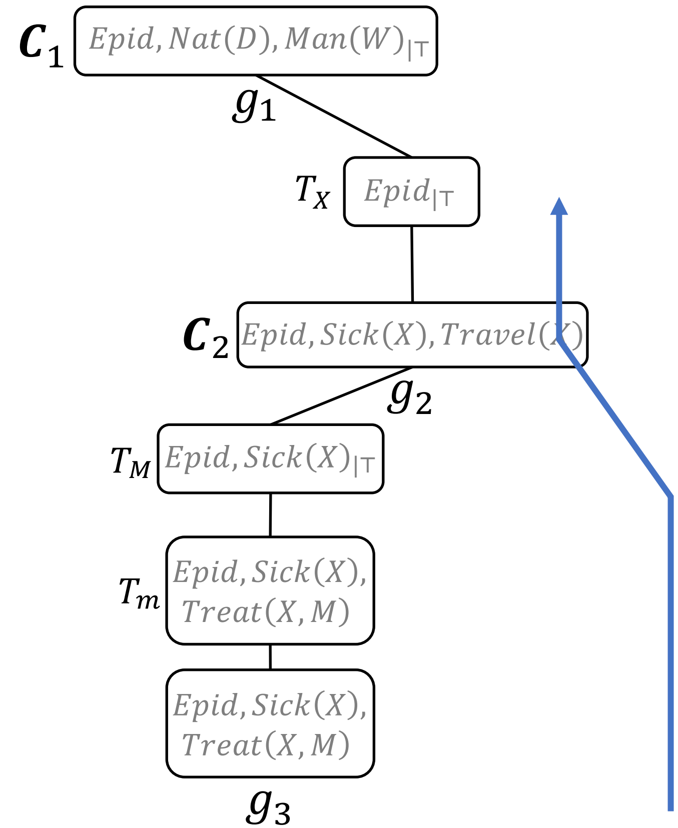
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_2\}$
  - Let us call it $\boldsymbol{C}_2$
  - Merge inbound
    - $\boldsymbol{C}_2$ and neighbouring parcluster identical → merge
    - T_x parcluster is a subset of $\boldsymbol{C}_2$ → merge



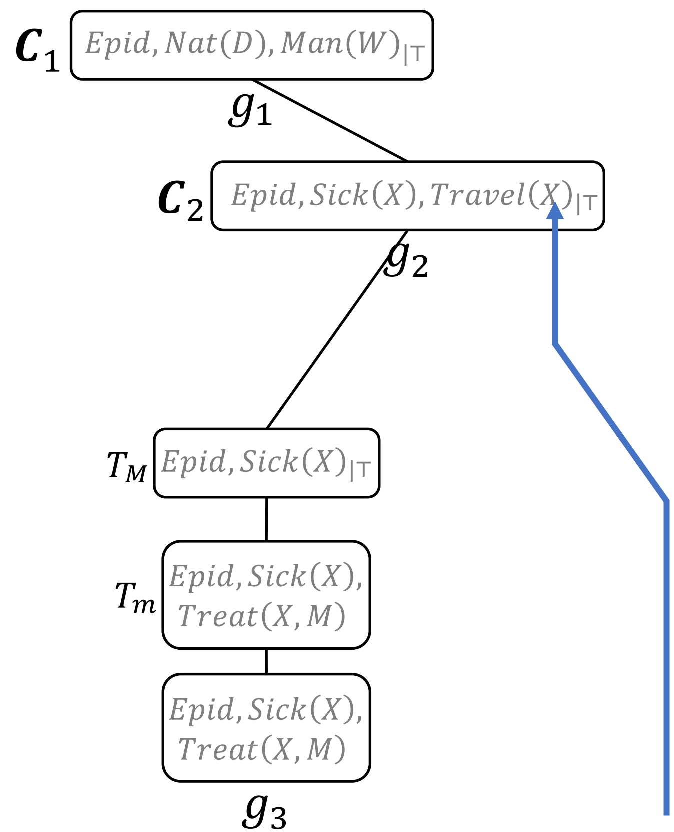
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_2\}$
  - Let us call it $\boldsymbol{C}_2$
  - Merge inbound
    - $\boldsymbol{C}_2$ and neighbouring parcluster identical → merge
    - T_x parcluster is a subset of $\boldsymbol{C}_2$ → merge
    - T_X parcluster is a subset of $\boldsymbol{C}_2$ → merge



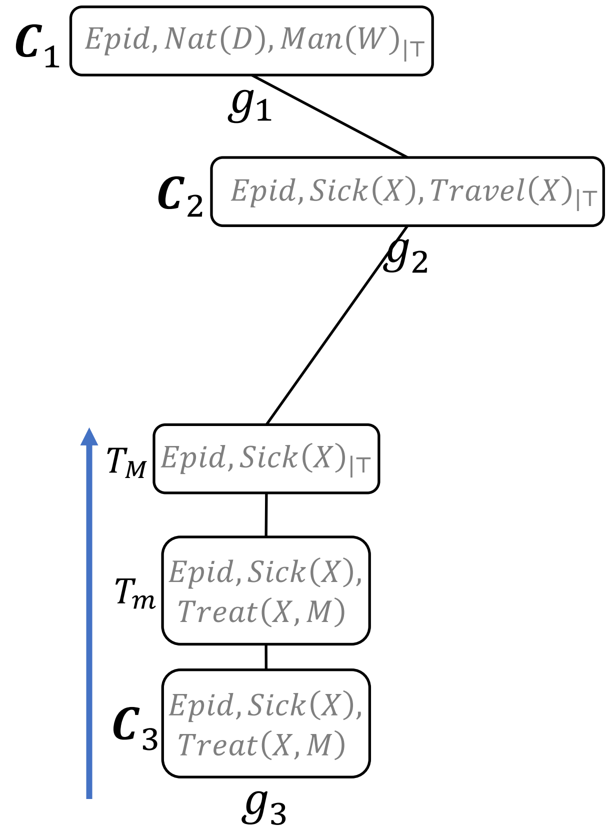
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_2\}$
  - Let us call it $\boldsymbol{C}_2$
  - Merge inbound
    - $\boldsymbol{C}_2$ and neighbouring parcluster identical → merge
    - T_x parcluster is a subset of $\boldsymbol{C}_2$ → merge
    - T_X parcluster is a subset of $\boldsymbol{C}_2$ → merge
  - Merging cannot move further inbound
    - $\boldsymbol{C}_1$ is neither a subset nor a superset of $\boldsymbol{C}_2$
    - Merging stops



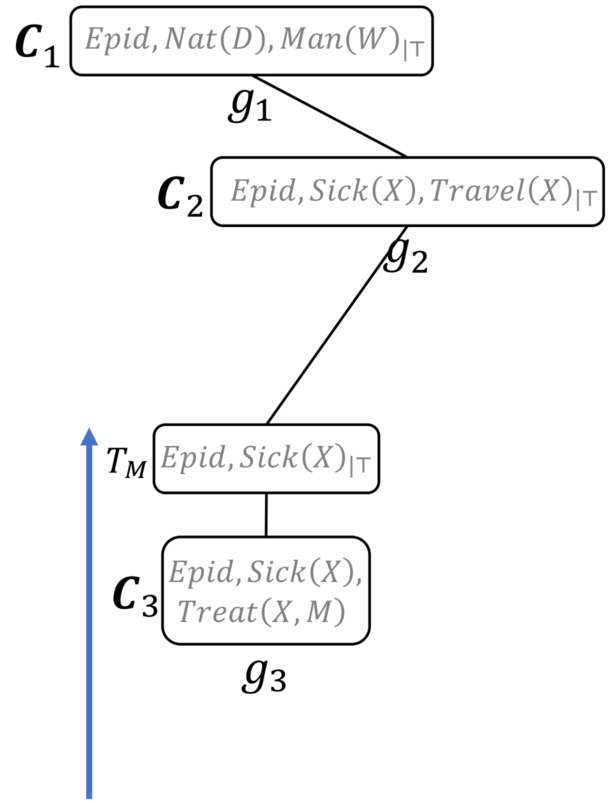
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_3\}$
  - Let us call it $\boldsymbol{C}_3$
  - Merge inbound
    - $\boldsymbol{C}_3$ and T_m parcluster identical → merge



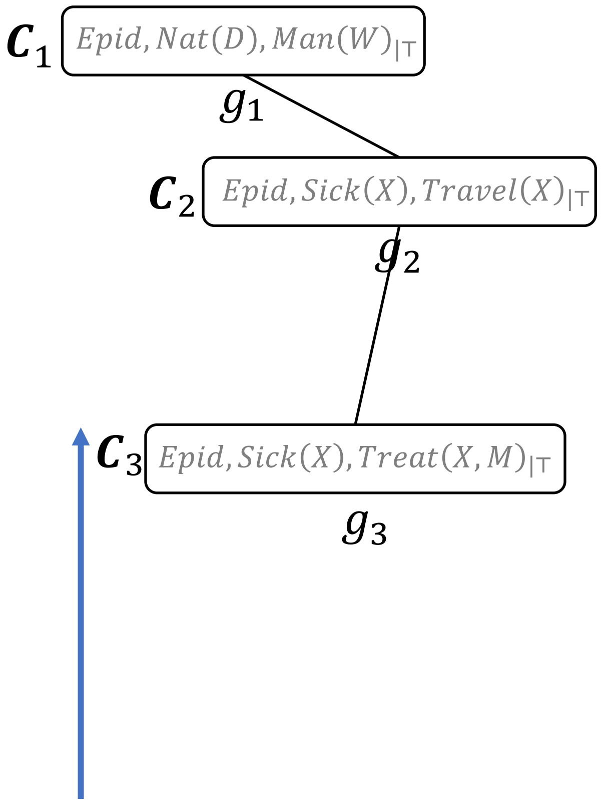
Minimisation: Example Continued

- Consider leaf parcluster with local model $\{g_3\}$
  - Let us call it $\boldsymbol{C}_3$
  - Merge inbound
    - $\boldsymbol{C}_3$ and T_m parcluster identical → merge
    - T_m parcluster is a subset of $\boldsymbol{C}_3$ → merge



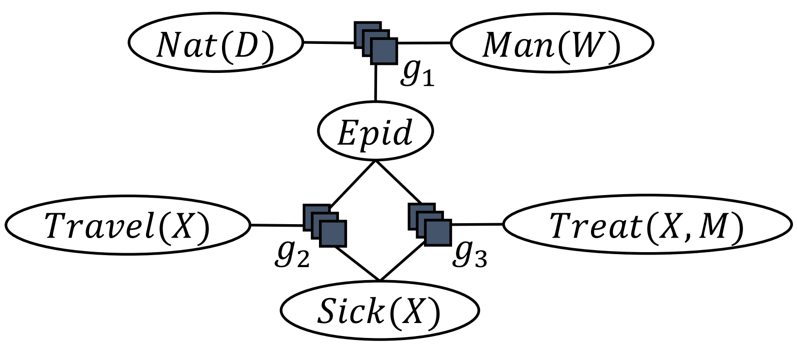
Minimisation: Example Continued

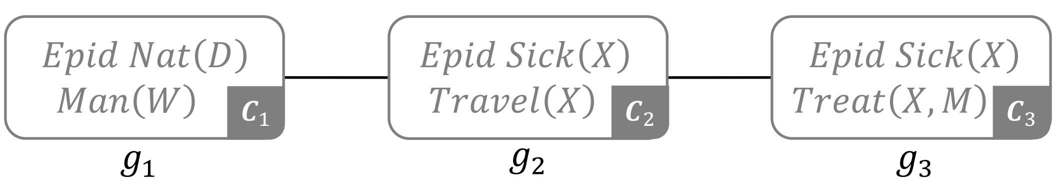
- Consider leaf parcluster with local model $\{g_3\}$
  - Let us call it $\boldsymbol{C}_3$
  - Merge inbound
    - $\boldsymbol{C}_3$ and T_m parcluster identical → merge
    - T_m parcluster is a subset of $\boldsymbol{C}_3$ → merge
  - Merging cannot move further inbound
    - $\boldsymbol{C}_3$ is neither a subset nor a superset of $\boldsymbol{C}_2$
    - Merging stops

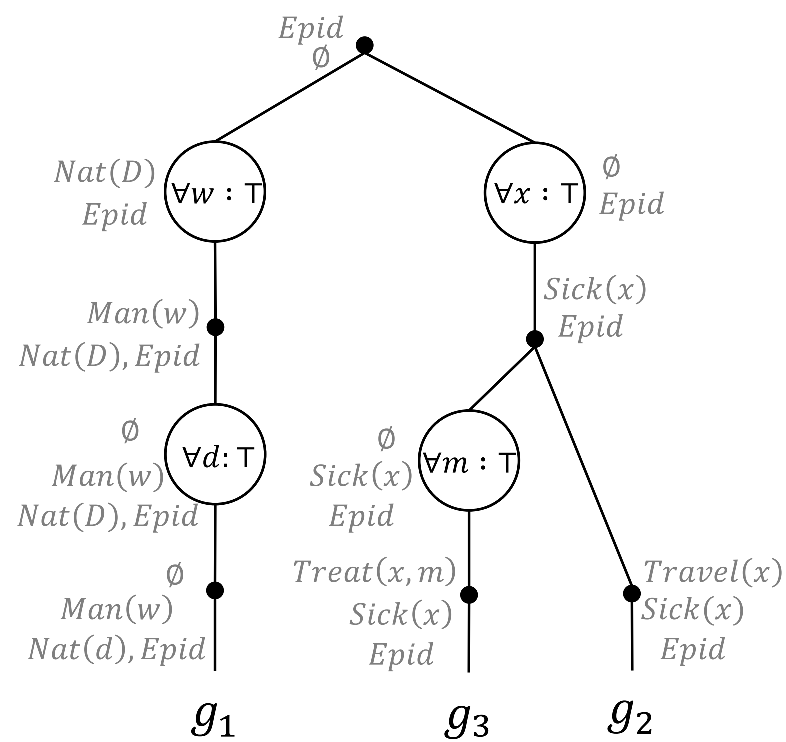


Minimisation: Example Continued

- Resulting FO jtree J from FO dtree T given model G
  - If we had started merging from leaf with g_3 inbound before merging from leaf with g_2, $\boldsymbol{C}_2$ and $\boldsymbol{C}_3$ would be switched







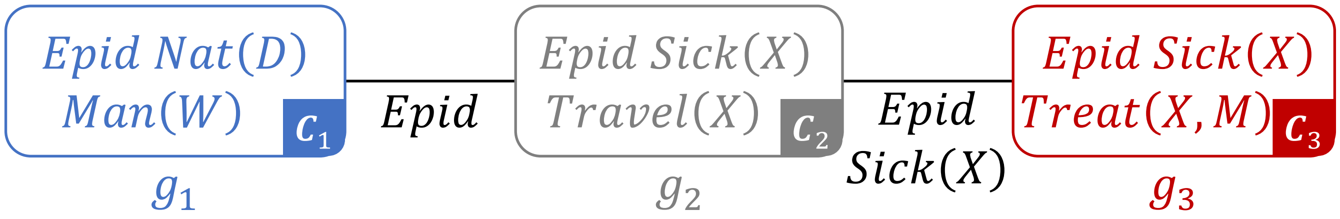
FO Jtree Construction

- Given a model G, the following steps are necessary
  1. Bring G into the required normal form for FO dtree construction
  2. Construct an FO dtree T for G
  3. Translate T into an FO jtree J
  4. Apply inverse substitutions to parclusters of descendants of DPG nodes in J
  5. Minimise J

  Construction
- Next?
- FO jtrees for query answering
  - Messages need to be passed to ensure independence
  - What about evidence?

Message Passing in FO Jtrees

- Ensure independence between parclusters
- Send messages based on two conditions
  - → If a node i has received all messages from neighbours but one, j, node i calculates and sends a message to j
  - → If a node i has received all messages, then it calculates and sends messages to all neighbours j that have not received a message yet

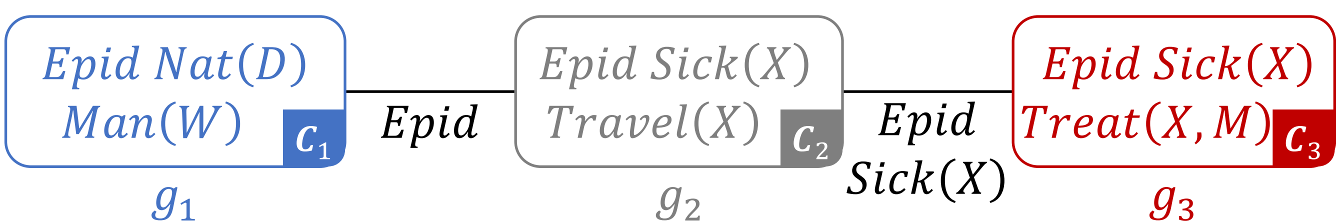


Message Passing in FO Jtrees

- Message m_{ij} from sender $\boldsymbol{C}_i$ to receiver $\boldsymbol{C}_j$
  - Set of parfactors $\{g_l\}_{l=1}^{n}$ with $rv(g_l) \subseteq \boldsymbol{S}_{ij}$
  - To calculate
    - Collect necessary information from local model and received messages:

$$G_{ij} = G_i \cup \bigcup_{k \in nbs(i), k \neq j} m_{ki}$$

      - Ignore the message that came from $\boldsymbol{C}_j$ (if it already exists)
    - Call slightly modified LVE with G_{ij} as input model, $\boldsymbol{S}_{ij}$ as query, and no evidence: $\mathrm{LVE}^*(G_{ij}, \boldsymbol{S}_{ij}, \emptyset)$
      - Specification of LVE^*: next slide



LVE for Message Passing

LVE$^*(G, Q, \{g_e\}_{e=1}^m)$

$G \leftarrow$ Shatter G on Q, $\{g_e\}_{e=1}^m$, and on itself

$G \leftarrow$ Absorb $\{g_e\}_{e=1}^m$ in G

while G contains non-query terms **do**

  if a PRV A fulfils the preconditions of $\mathrm{sum{-}out}$ **then**

    $G \leftarrow$ Apply $\mathrm{sum{-}out}$ to A in G

  else

    $G \leftarrow$ Apply an enabling operator (multiply, count−convert, expand, count−normalise, split, ground) on some parfactors in G

~~$g \leftarrow$ Multiply all parfactors in G into one parfactor~~

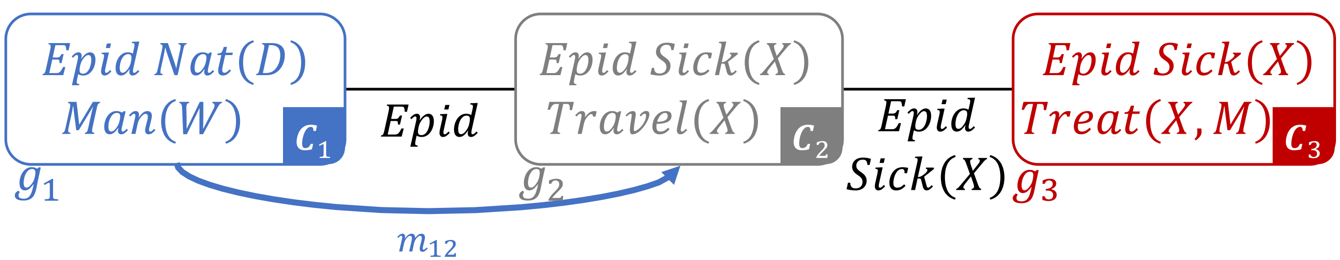
~~$g \leftarrow$ Normalise the potentials in g~~

~~**return** g~~

$\boxed{\textbf{return } G}$

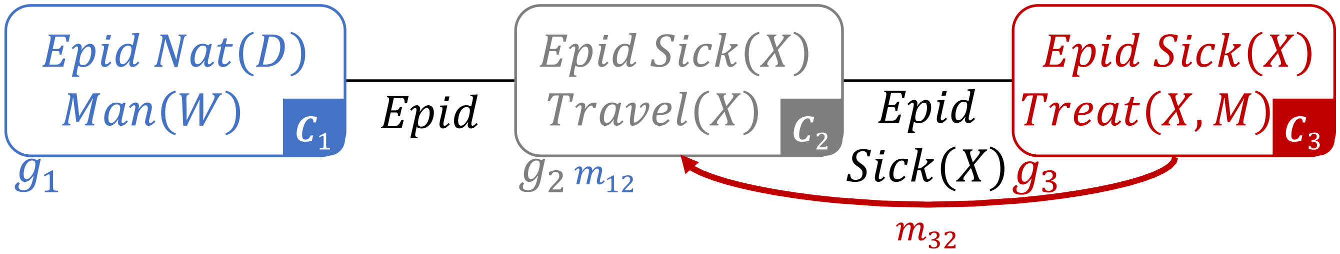
Message Passing in FO Jtrees

- E.g.,
  - Message m_{12} from $\boldsymbol{C}_1$ to $\boldsymbol{C}_2$
    - Collect $G_{12} = \{g_1\} \cup \emptyset$
      - No further neighbours except $\boldsymbol{C}_2$
    - Call $\mathrm{LVE}^*(\{g_1\}, \{Epid\}, \emptyset)$
      - LVE^* eliminates $Nat(D), Man(W)$ from $\{g_1\}$
        - Count-converting $Nat(D)$ into $\#_D[Nat(D)]$
        - Summing out $Man(W)$ and then $\#_D[Nat(D)]$
        - Returning $\{g_1'\}$
    - Send $\{g_1'\}$ as m_{12} to $\boldsymbol{C}_2$



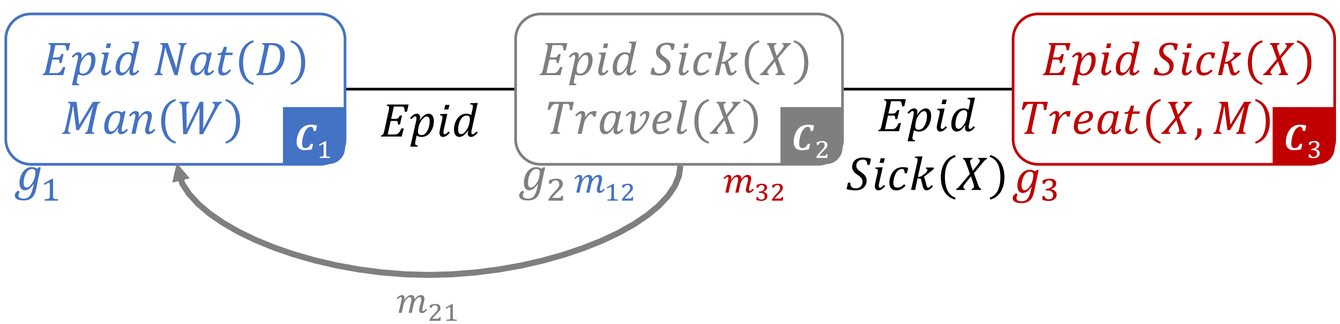
Message Passing in FO Jtrees

- E.g.,
  - Message m_{32} from $\boldsymbol{C}_3$ to $\boldsymbol{C}_2$
    - Collect $G_{32} = \{g_3\} \cup \emptyset$
      - No further neighbours except $\boldsymbol{C}_2$
    - Call $\text{LVE}^*(\{g_3\}, \{Epid, Sick(X)\}, \emptyset)$
      - LVE^* eliminates $Treat(X, M)$ from $\{g_3\}$
        - Summing out $Treat(X, M)$
        - Returning $\{g_3'\}$
    - Send $\{g_3'\}$ as m_{32} to $\boldsymbol{C}_2$



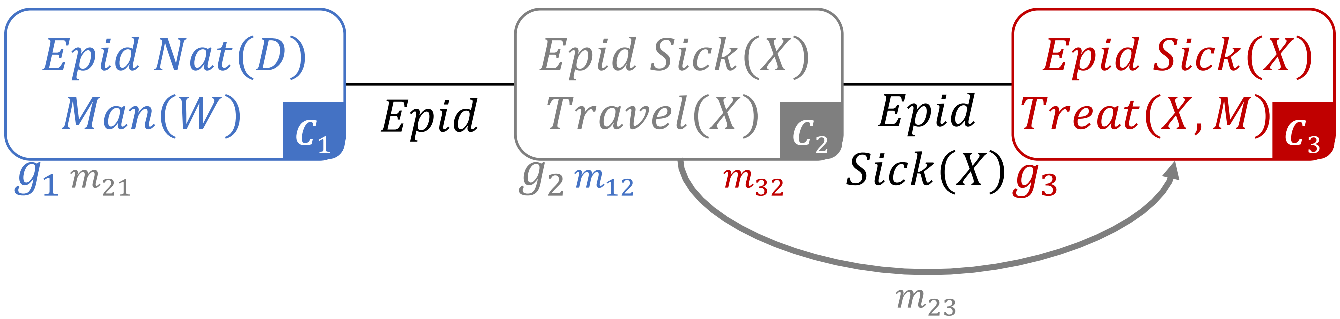
Message Passing in FO Jtrees

- E.g.,
  - Message m_{21} from $\boldsymbol{C}_2$ to $\boldsymbol{C}_1$
    - Collect $G_{21} = \{g_2\} \cup m_{32}$
      - Further neighbour: $\boldsymbol{C}_3$, sent message $m_{32} = \{g_3'\}$
    - Call $\mathrm{LVE}^*(\{g_2, g_3'\}, \{Epid\}, \emptyset)$
      - LVE^* eliminates $Travel(X), Sick(X)$ from $\{g_2, g_3'\}$
        - Summing out $Travel(X)$ from g_2, yielding g_2'
        - Summing out $Sick(X)$ from product of g_2' and g_3', yielding g_{23}'
        - Returning $\{g_{23}'\}$
    - Send $\{g_{23}'\}$ as m_{21} to $\boldsymbol{C}_1$



Message Passing in FO Jtrees

- E.g.,
  - Message m_{23} from $\boldsymbol{C}_2$ to $\boldsymbol{C}_3$
    - Collect $G_{23} = \{g_2\} \cup m_{12}$
      - Further neighbour: $\boldsymbol{C}_1$, sent message $m_{12} = \{g_1'\}$
    - Call $\text{LVE}^*(\{g_2, g_1'\}, \{Epid, Sick(X)\}, \emptyset)$
      - LVE^* eliminates $Travel(X)$ from $\{g_2, g_1'\}$
        - Summing out $Travel(X)$ from g_2, yielding g_2'
        - Returning $\{g_2', g_1'\}$
    - Send $\{g_2', g_1'\}$ as m_{23} to $\boldsymbol{C}_3$



Message Passing: Overview

- Given an FO jtree J, send messages if one of the two conditions is true
  - → If a node i has received all messages from neighbours but one, j, node i calculates and sends a message to j
  - → If a node i has received all messages, then it calculates and sends messages to all neighbours j that have not received a message yet
- To calculate a message:
  - Collect necessary information from local model and received messages:

$$G_{ij} = G_i \cup \bigcup_{k \in nbs(i), k \neq j} m_{ki}$$

  - Call $\mathrm{LVE}^*(G_{ij}, \boldsymbol{S}_{ij}, \emptyset)$

Message Passing

Query Answering in FO Jtrees

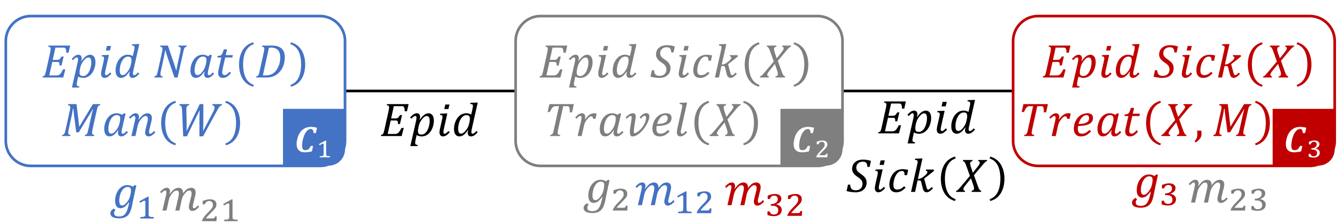
- After message passing, the parclusters are independent from each other given the messages
  - Prepared for query answering

Query Answering

- For each query with query term Q
  - Find parcluster $\boldsymbol{C}_i$ s.t. $Q \in \boldsymbol{C}_i$
  - Collect information from local model and messages, i.e,

$$G_Q = G_i \cup \bigcup_{j \in nbs(i)} m_{ji}$$

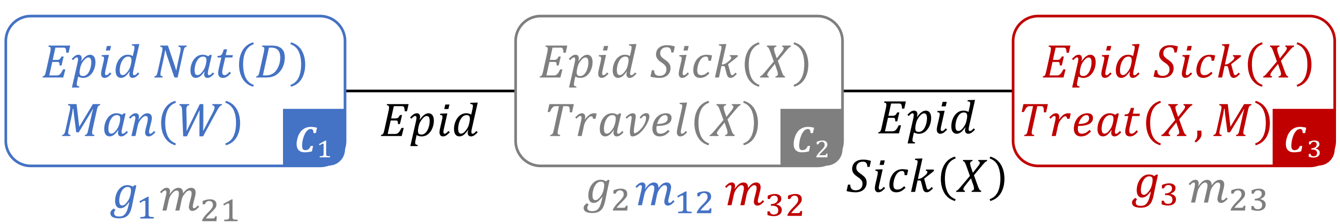
  - Call $\mathrm{LVE}\left(G_Q, Q, \emptyset\right)$ and return or store result of the call



Query Answering in FO Jtrees

- E.g., $P(Epid)$
  - All parclusters contain $Epid$, choose one at random, e.g., $\boldsymbol{C}_2$
  - Collect $G_{Epid} = \{g_2\} \cup m_{12} \cup m_{32} = \{g_2, g_1', g_3'\}$
  - Call $\mathrm{LVE}(\{g_2, g_1', g_3'\}, Epid, \emptyset)$, yielding a parfactor g containing the probability distribution over $Epid$

- What about evidence?

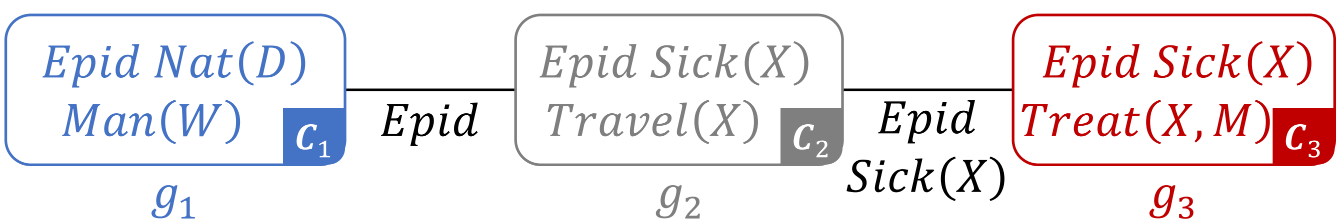


Evidence in FO Jtrees

- Evidence applies to PRVs in some parclusters
  - Changes the distributions in local models
  - Information sent in messages might change
    - Even if summed out and therefore hidden from the other parclusters
- Therefore, handle evidence before sending messages
- Given a set of evidence parfactors $\left\{\phi_e\big(R(\boldsymbol{X})\big)_{|C_e}\right\}_{e=1}^{m}$
  - For each $\phi_e\big(R(\boldsymbol{X})\big)$
    - For *each* parcluster $\boldsymbol{C}_i$ where $R(\boldsymbol{X}) \in \boldsymbol{C}_i$
      - Shatter G_i on $R(\boldsymbol{X})_{|C_e}$
      - Absorb $\phi_e\big(R(\boldsymbol{X})\big)_{|C_e}$ in G_i

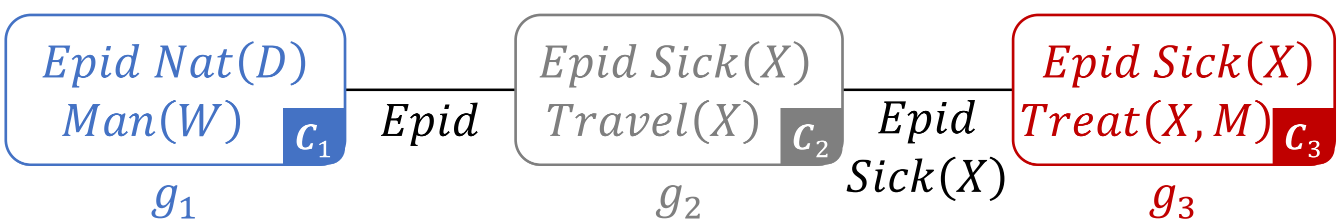
Evidence Entering

- Only, then send messages



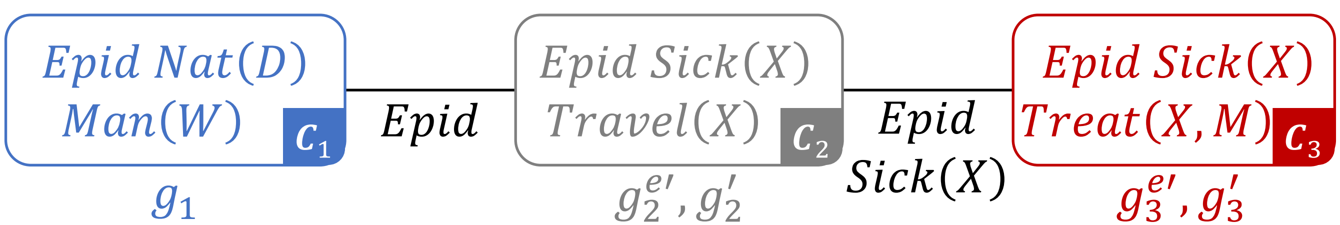
Evidence in FO Jtrees

- E.g., given $Sick(eve) = true$ as evidence in g_e
  - In $\boldsymbol{C}_2$
    - Shatter $G_2 = \{g_2\}$ on $Sick(eve)$, yielding $\{g_2^e, g_2'\}$
    - Absorb g_e in g_2^e, yielding $g_2^{e\prime}$
    - Result: $G_2 = \{g_2^{e\prime}, g_2'\}$
  - In $\boldsymbol{C}_3$
    - Shatter $G_3 = \{g_3\}$ on $Sick(eve)$, yielding $\{g_3^e, g_3'\}$
    - Absorb g_e in g_3^e, yielding $g_3^{e\prime}$
    - Result: $G_3 = \{g_3^{e\prime}, g_3'\}$
  - Then, send messages based on the local models that have absorbed the evidence



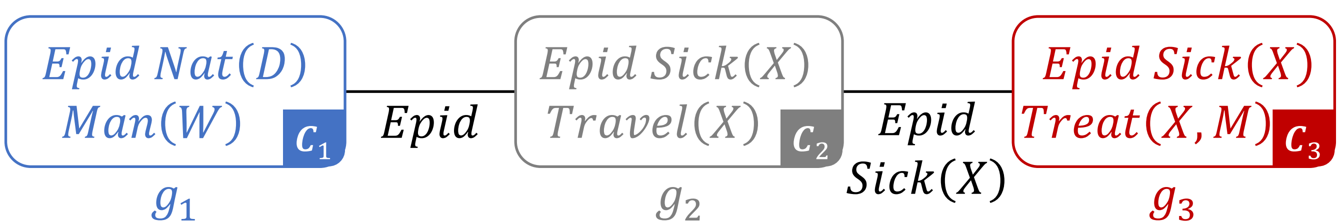
Evidence in FO Jtrees

- E.g., given $Sick(eve) = true$ as evidence in g_e
  - Message m_{12} does not change compared to previous example
  - Message m_{32} calculated based on $\{g_3^{e\prime}, g_3^{\prime}\}$
    - Call LVE*($\{g_3^{e\prime}, g_3^{\prime}\}, \{Epid, Sick(X)\}, \emptyset$), yielding $\{g_3^{e\prime\prime}, g_3^{\prime\prime}\}$
  - Message m_{23} calculated based on $\{g_2^{e\prime}, g_2^{\prime}\} \cup m_{12}$
    - Call LVE*($\{g_2^{e\prime}, g_2^{\prime}, g_1^{\prime}\}, \{Epid, Sick(X)\}, \emptyset$), yielding $\{g_2^{e\prime\prime}, g_2^{\prime\prime}, g_1^{\prime}\}$
  - Message m_{21} calculated based on $\{g_2^{e\prime}, g_2^{\prime}\} \cup m_{32}$
    - Call LVE*($\{g_2^{e\prime}, g_2^{\prime}, g_3^{e\prime\prime}, g_3^{\prime\prime}\}, \{Epid\}, \emptyset$), yielding $\{g_2^{e\prime\prime}, g_2^{\prime\prime}, g_3^{e\prime\prime}, g_3^{\prime\prime\prime}\}$



Evidence and Queries in FO Jtrees

- After evidence handling
  - All queries are answered in an FO jtree with handled evidence $\{g_e\}_{e=1}^m$ yield results conditional on $\{g_e\}_{e=1}^m$
  - So, given evidence $\{g_e\}_{e=1}^m$ and query terms $\{Q_i\}_{i=1}^n$ for a model G
    - The posed queries are $P(Q_i \mid \{g_e\}_{e=1}^m), 1 \leq i \leq n$, w.r.t. P_G
- FO jtree constructed without specific evidence
  - Reuse for different evidence sets
    - As long as model stays the same
  - Reset the local models before entering new evidence



LJT: Algorithm

$\mathbf{LJT}(G, \{Q_i\}_{i=1}^{n}, \{g_e\}_{e=1}^{m})$

Construct an FO jtree J for G

Enter evidence $\{g_e\}_{e=1}^{m}$ into J

Pass message in J

Answer queries with query terms $\{Q_i\}_{i=1}^{n}$ in J

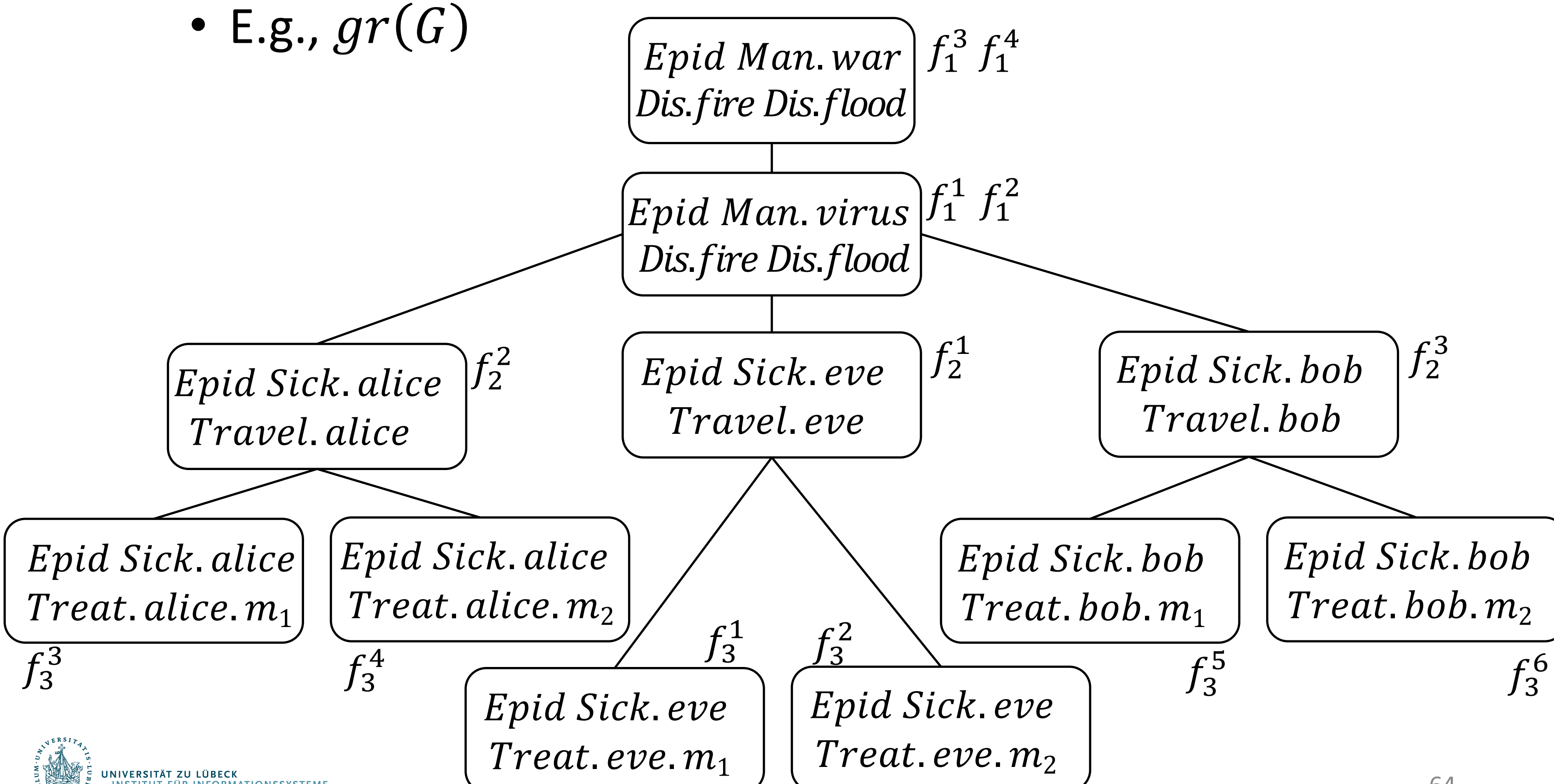
- Look for blue boxes on the previous slides to find the descriptions of each step

  Step Name

- *Constant* overhead for FO jtree construction
- Payoff if given multiple queries

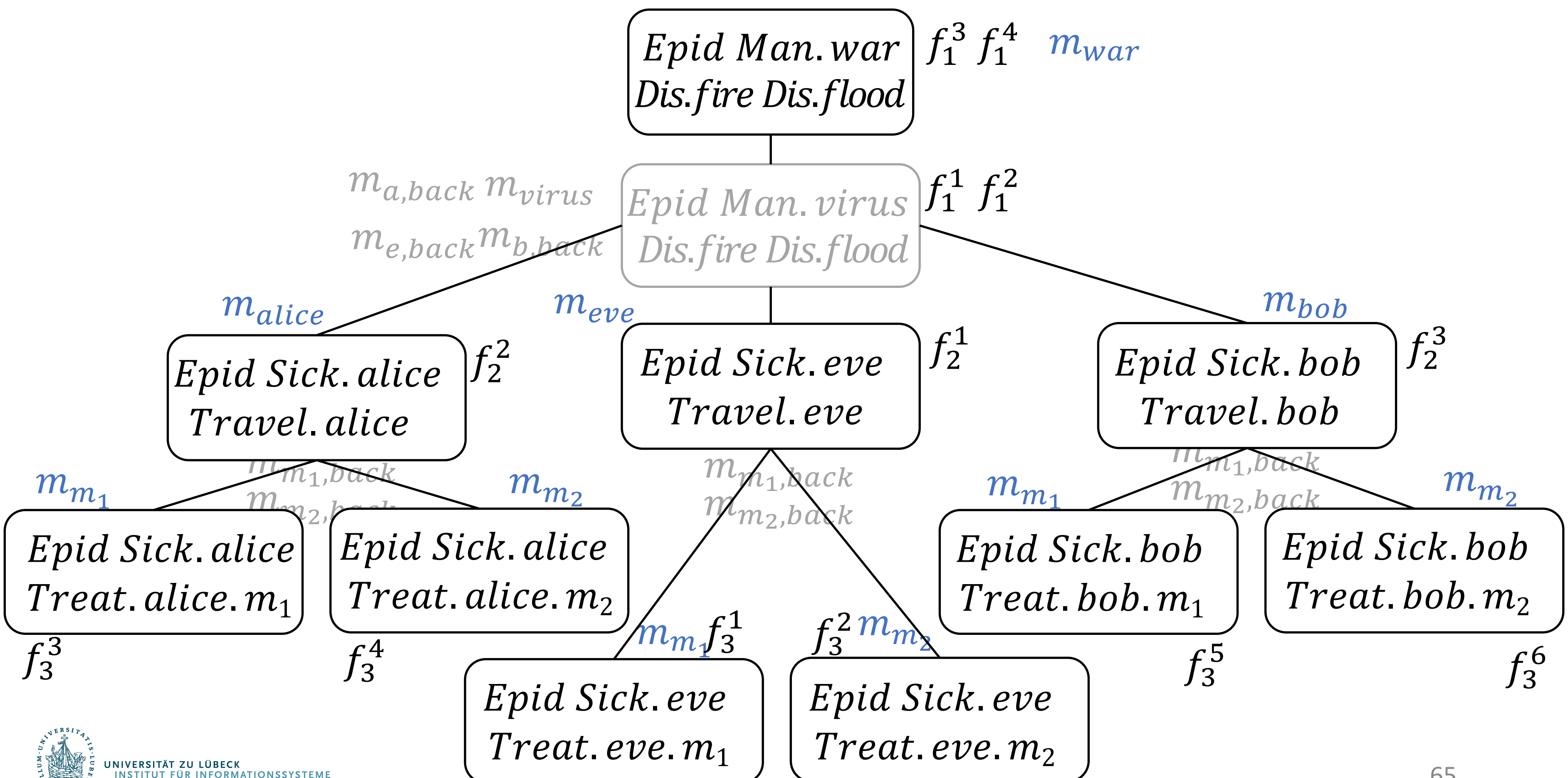
Comparison to Ground Inference

- Propositional Junction Tree Algorithm (JT)
  - Same algorithm, only with propositional model
  - E.g., $gr(G)$



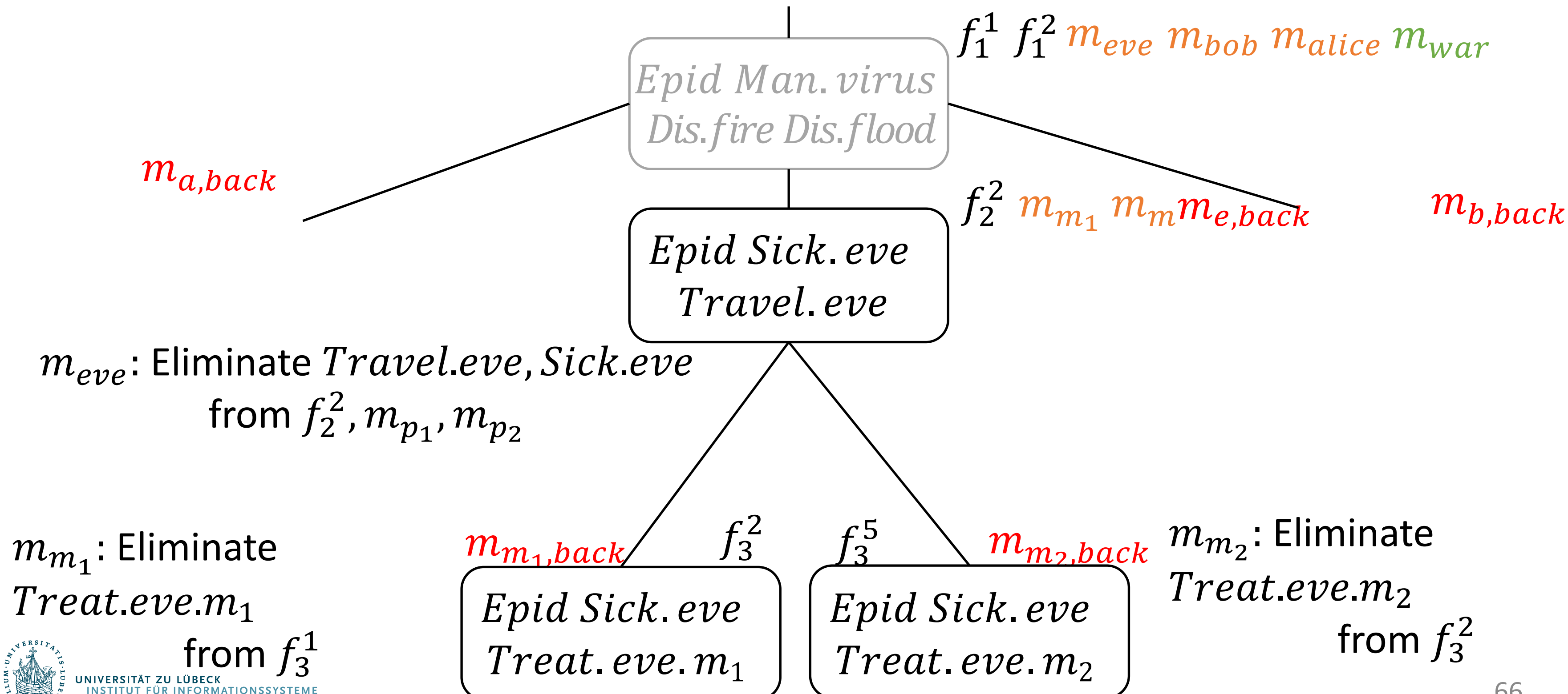
Junction Tree: Messages

- From periphery to centre and back

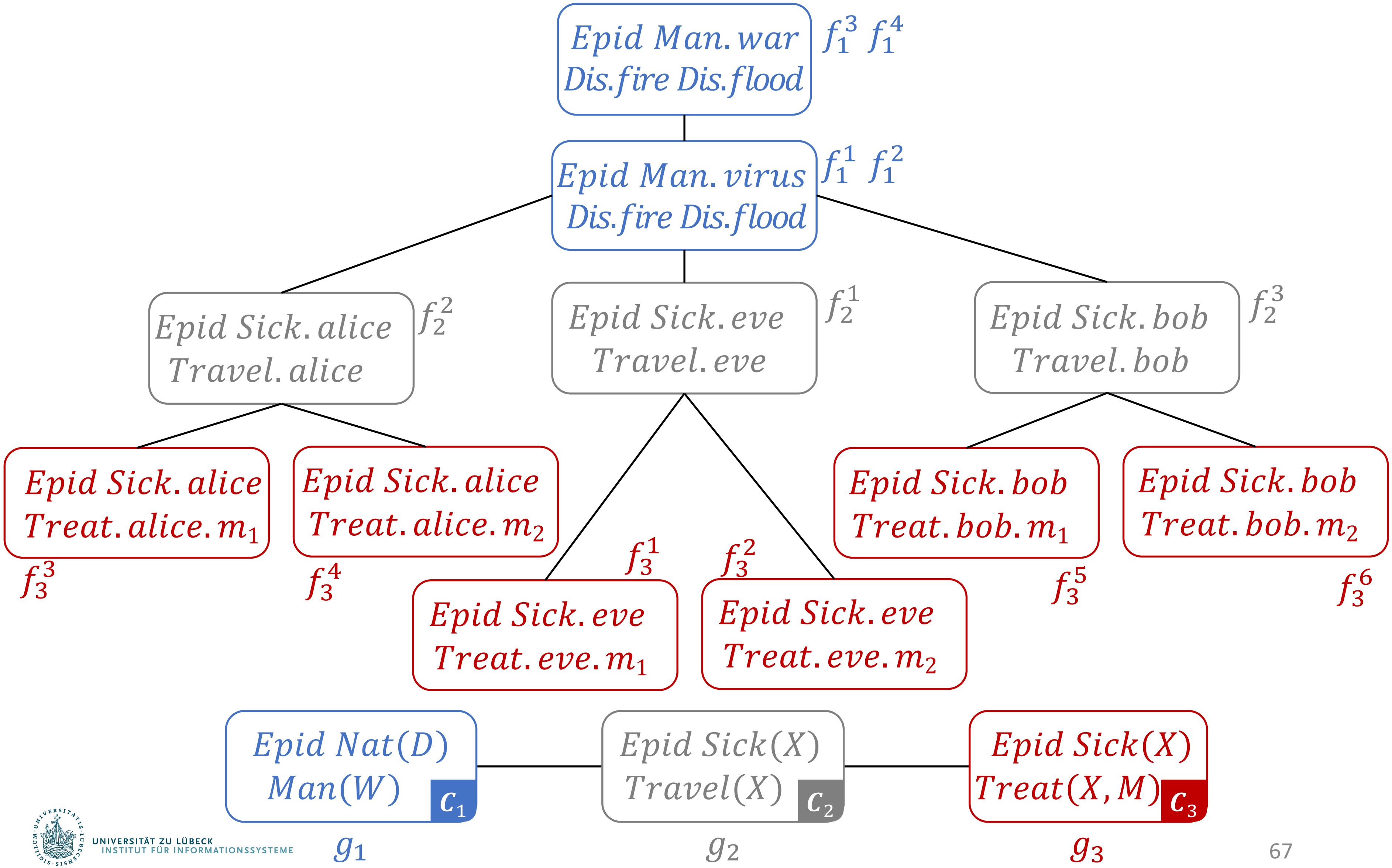


Junction Tree: Symmetry → Inefficiency

- Identical messages incoming
- Information already present
- Calculating identical messages + sending information partially present



Compact Encoding of Jtrees



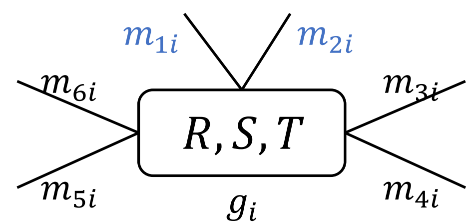
Message Calculation Strategies

- Message calculation strategy seen so far
  - Eliminate all non-separator PRVs from all messages but the one that came from receiver and the local model
  - Called *Shafer-Shenoy* architecture after the two researchers who first presented the scheme
- Another strategy for JT exists, called *Hugin* architecture
  - Multiply the factors of local model G_i into one factor g_i
  - Multiply each incoming message m_{ji} into g_i
    - Store m_{ji} as well
    - Each message consists of only one factor (no longer a set)
  - When sending message m_{ij}
    - Eliminate all non-separator randvars from $\frac{g_i}{m_{ji}}$
      - I.e., divide g_i by m_{ji} first
      - If m_{ji} does not exist, then divide by a symbolic 1 (or no division)

Message Calculation Strategies

What about a Lifted Hugin?

- *Hugin* architecture continued
  - May enlarge the factors at each node to the worst-case size of each node
    - E.g., $G_i = \{\phi(R,S), \phi(S,T)\} \rightarrow g_i = \phi(R,S,T)$
  - May lead to more involved multiplications
    - E.g., multiplying message $m_{ij} = \phi(R)$ into $g_i = \phi(R,S,T)$ more involved then multiplying $\phi(R)$ into an intermediate result $\phi'(R)$
  - Pays off if the nodes of the jtree have a high degree
    - Many duplicate multiplications during message calculation
    - E.g., $m_{3i}, m_{4i}, m_{5i}, m_{6i}$ and g_i have to be multiplied for both m_{1i}, m_{2i}



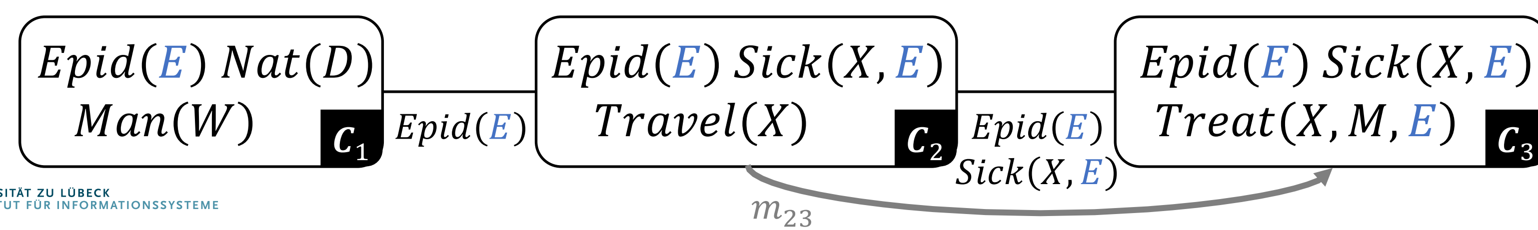
  - Requires a division operator for factors

Message Calculation Strategies

- Lifted Hugin?
  - Arguments pro and con also apply to lifted version
    - May enlarge the factors at each node to the worst-case size of each node
    - May lead to more involved multiplications
    - Pays off if the nodes of the jtree have a high degree
    - Requires a division operator for factors
  - Main obstacle: So far, no lifted division operator
    - We are working on it @Moritz
  - Also, CAUTION: In general, parfactors may be multiplied with different logvars such that previously unnecessary count conversions might become necessary

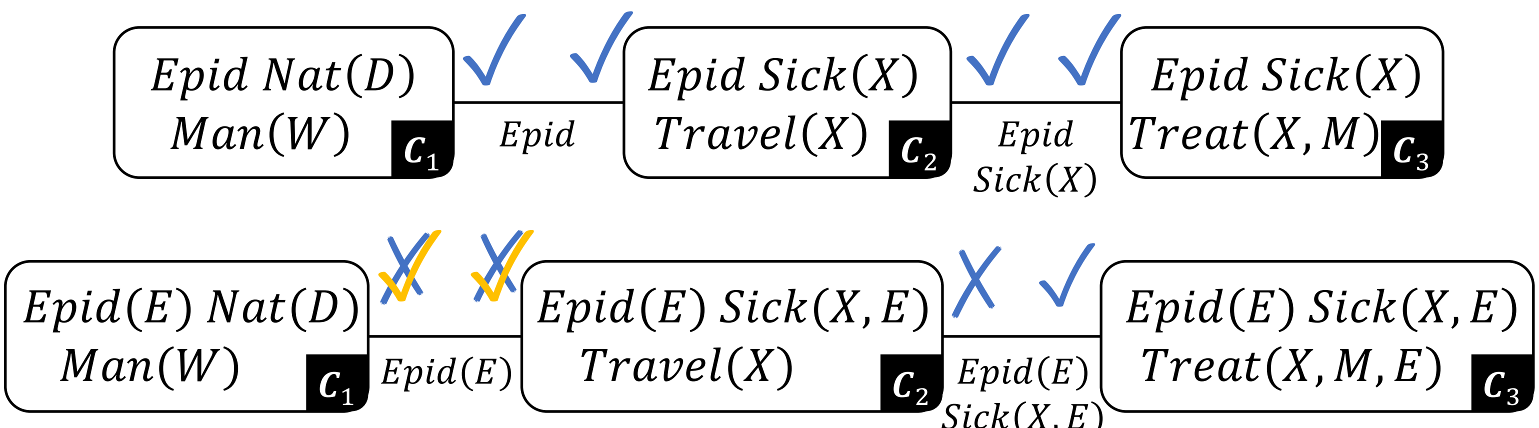
In terms of Lifting: Is it that simple?

- Algorithm-induced groundings due to message passing
  - For message calculation, non-separator PRVs are eliminated with separator PRVs as the query terms containing logvars
    - Non-separator PRVs have to fulfil $sum-out$ preconditions
      1. $\forall B \in rv(G \setminus \{g\}) : gr\left(B_{|C}\right) \cap gr\left(A_{|(\mathcal{X},C_{\mathcal{X}})}\right) = \emptyset$
      2. $\forall X \in \{X \mid |\pi_X(C_{\mathcal{X}})| > 1\} : X \in lv(A)$
      3. $\boldsymbol{X}^{excl} = lv(A) \setminus \left(\boldsymbol{X} \setminus lv(A)\right)$ count-normalised w.r.t. $\boldsymbol{X}^{com} = lv(A) \cap \boldsymbol{X}$ in C, with $\boldsymbol{X}$ the set of $\mathcal{X}$
    - Preconditions 1 + 3 fulfilled by construction
    - Precondition 2 may not be fulfilled → can cause groundings
    - E.g., logvar E added to PRVs $Epid, Sick(X), Treat(X, M)$
      - When calculating m_{23}, one has to eliminate $Travel(X)$
        - But: it does not contain both X and E and a count conversion does not apply as E occurs in two PRVs → E gets grounded



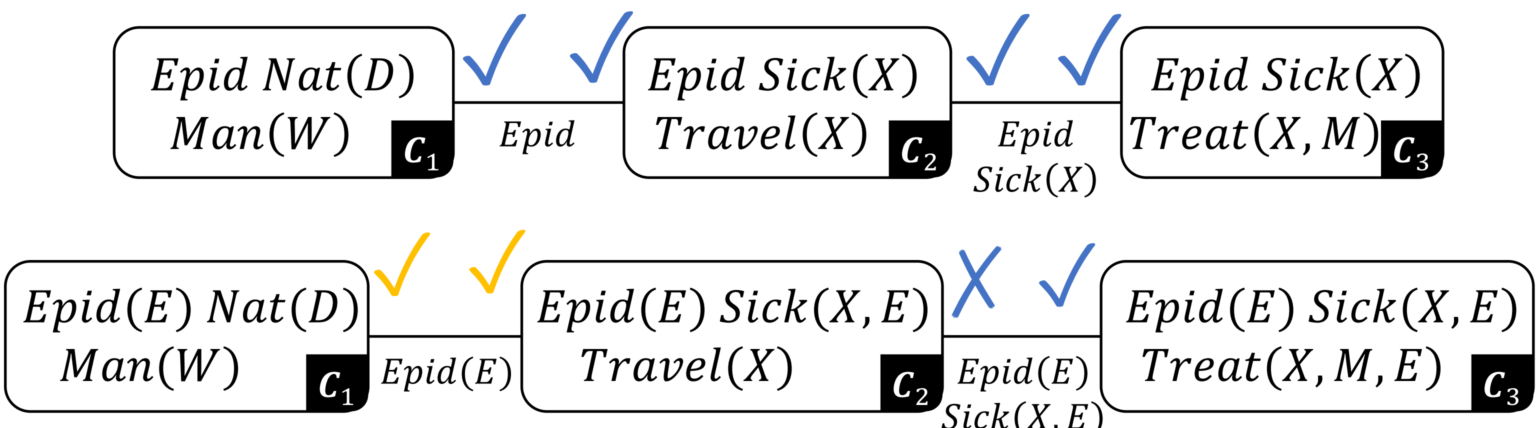
Conditions on Groundings

- For a lifted calculation of message m_{ij}, it necessarily has to hold that
  - for each PRV $A \in \left(\boldsymbol{C}_i \setminus \boldsymbol{S}_{ij}\right)$, i.e., A has to be eliminated:
    - for each separator PRV $S \in \boldsymbol{S}_{ij}$: $lv(S) \subseteq lv(A)$ (Cond. 1)
- If Cond. 1 does not hold, i.e., $lv(S) \nsubseteq lv(A)$, one may induce Cond. 1 by count conversion
  - If $lv(S) \setminus lv(A)$ are countable in G_{ij} (Cond. 2)



Conditions on Groundings

- Problem with induced Cond. 1 using count conversions on the logvars in $lv(S) \setminus lv(A)$:
  - Logvars that were previously not counted are now counted
  - All receiving parclusters need to be able to handle the counted versions, which needs to be checked
    - If a newly counted logvar arrives at a parcluster $\boldsymbol{C}_k$, it has to be countable in G_k as well (Cond. 3)
      - For further calculations, since they refer to the same set of randvars, they have to occur in the same form, i.e., at one point the logvar has to be counted in G_k as well

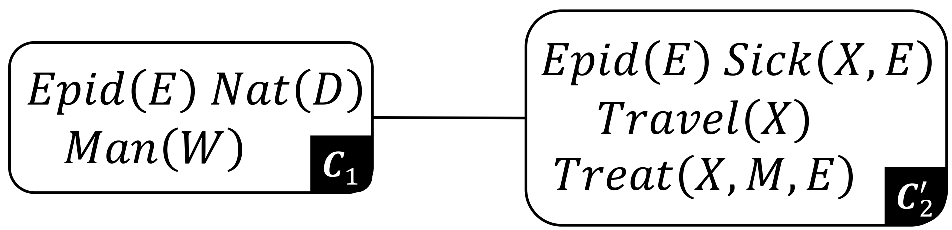


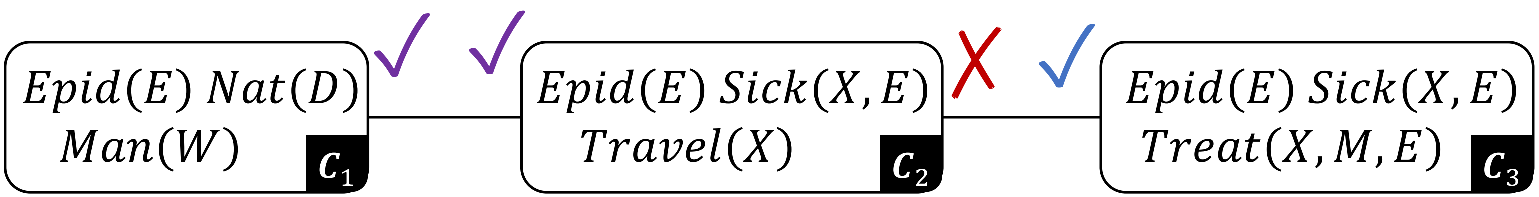
Fusion

- Extra step at end of construction called fusion
  - Test each possible message m_{ij} for each PRV A to eliminate and each separator PRV S based on the three conditions
    - If Cond. 1 holds: no groundings for A and S; continue
    - Otherwise:
      - If Cond. 2 holds: check Cond. 3
        - If Cond. 3 holds: no groundings for A and S; continue
        - Otherwise: groundings; mark m_{ij}; continue with next message
      - Otherwise: groundings; mark m_{ij}; continue with next message
  - For each message m_{ij} marked:
    - Merge parclusters $\boldsymbol{C}_i, \boldsymbol{C}_j$ (as in minimisation)

Fusion

  - E.g.,
    - Testing marks m_{23} → merge $\boldsymbol{C}_2, \boldsymbol{C}_3$





LJT: Complexity

- Uses also the notion of lifted width $w_T = (w_g, w_\#)$
  - w_g largest ground width
  - $w_\#$ largest counting width
  - As FO jtree constructed from FO dtree, w_T identical between LVE and LJT
    - Fusion may change w_T in terms of the FO jtree
      - But in terms of the LVE calculations in the merged parcluster, w_T is still the same with multiple nodes being combined into one
      - For simplicity, let us consider models that all fulfil Cond. 1 in fusion such that w_T is identical for both LJT and LVE

LJT: Complexity

- LJT complexity based on complexity of LVE:
  $$O\left(n_T \cdot \log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#}\right)$$
- Complexity of individual steps
  - Construction: linear in number of nodes, no calculations; negligible compared to later steps
  - Evidence entering: $O\left(n_J \cdot \log_2(n) \cdot r^{w_g - 1} \cdot n^{r_\# w_\#}\right)$
    - Absorbing evidence complexity: $O\left(\log_2(n) \cdot r^{w_g - 1} \cdot n^{r_\# w_\#}\right)$
      - Visits $\frac{1}{r} \cdot r^{w_g} \cdot n^{r_\# w_\#}$ lines, possibly exponentiates the potentials
    - At each node $\rightarrow n_J \cdot O\left(\log_2(n) \cdot r^{w_g - 1} \cdot n^{r_\# w_\#}\right)$
      - n_J number of nodes in FO jtree J
    - For each e evidence parfactors $\rightarrow e \cdot O\left(n_J \cdot \log_2(n) \cdot r^{w_g - 1} \cdot n^{r_\# w_\#}\right)$
      - Assuming $e \ll n_J \rightarrow O\left(n_J \cdot \log_2(n) \cdot r^{w_g - 1} \cdot n^{r_\# w_\#}\right)$
  - First two steps accumulated: $O\left(n_J \cdot \log_2(n) \cdot r^{w_g - 1} \cdot n^{r_\# w_\#}\right)$

LJT: Complexity

- Complexity of individual steps
  - First two steps accumulated: $O\left(n_J \cdot \log_2(n) \cdot r^{w_g - 1} \cdot n^{r_\# w_\#}\right)$
  - Message passing: $O\left(n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#}\right)$
    - Calculating one message = answering one query on a parcluster
      - Worst-case parfactor size at parcluster: $r^{w_g} \cdot n^{r_\# w_\#}$
      - Elimination of $\left|\boldsymbol{C}_i \setminus \boldsymbol{S}_{ij}\right|$ PRVs goes through each line, potentials may be exponentiated → $O(\log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#})$
    - Two messages per edge, $n_J - 1$ edges in J → $n_J \cdot O(\log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#})$
  - Query answering: $O(m \cdot \log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#})$
    - Each query answered in one parcluster → $O(\log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#})$
    - With m query terms → $m \cdot O(\log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#})$
- All four steps accumulated:

$$O\left(\left(n_J + m\right) \cdot \log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#}\right)$$

Comparison to LVE

- For both holds: w_g bounded from below by $\max_{\phi(A_1,\dots,A_k)\in G} k$
- LVE complexity of one query
  = LJT complexity of message passing
  - $O(n_T \cdot \log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#})$ vs.
  - $O(n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#})$
  - Actual number of calculations:
    - In LVE: c_{LVE}
    - For message pass: $2 \cdot c_{LVE}$
- For m queries
  - LVE: $O(m \cdot n_T \cdot \log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#})$
  - LJT: $O\left((n_J + m) \cdot \log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#}\right)$
  - Difference in $m \cdot n_T$ vs. $(n_J + m)$
    - LVE has complexity of $O(n_T \cdot \log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#})$ for one query
    - LJT only complexity of $O(\log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#})$ for one query

LJT only pays off if $m > 1$, most likely, starting with third query (two queries in LVE = one message pass)

LJT: Completeness

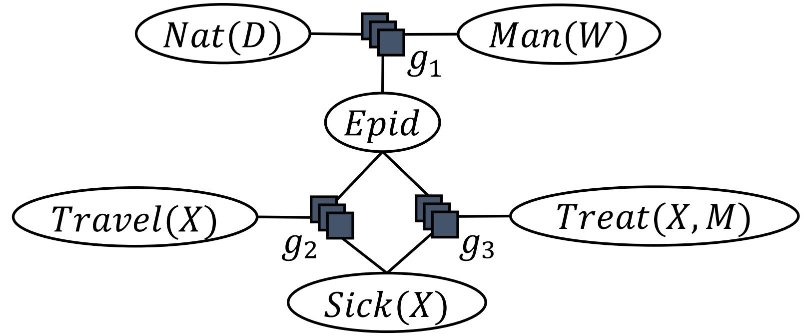
- Completeness results from LVE also hold for LJT
  - Proof using the FO jtree properties and the fusion conditions on a case basis regarding separators:
    - Separators containing only propositional randvars
      - Do not interfere with elimination order for $\mathrm{sum-out}$
    - Separators additionally containing one-logvar PRVs
      - Do not interfere with elimination order for $\mathrm{sum-out}$
        - Two-logvar PRVs within a parcluster eliminable
        - One logvar PRVs within a parcluster eliminable
          - Given all available counting versions
          - (Along the lines of completeness proof for $\mathcal{M}^{1prv}$)
    - Separators additionally containing two-logvar PRVs
      - All two-logvar PRVs eliminable
        - If inequality constraint between them → same parcluster, eliminable within one parcluster using group inversion
      - Because of fusion, PRVs with less logvars also part of separator or eliminable (may it be through extra count conversion)
        - Else parclusters would have been merged

LJT: Implementation

- Available at:
  - https://www.ifis.uni-luebeck.de/index.php?id=518&L=2
  - Based on the LVE implementation by Taghipour
    - Available at:
      - https://dtai.cs.kuleuven.be/software/gcfove
  - Includes an implementation of the propositional junction tree algorithm for comparison
- Input: BLOG files
  - Based on Bayesian Logic Programming Language
    - https://bayesianlogic.github.io

Runtimes: Increasing Domain Sizes

- Example model
  - All domain sizes $\in \{2,4,\dots,20,30,\dots,100, 200,\dots,1000\}$
  - No evidence
  - Queries:
    - $P(Travel(x_1))$
    - $P(Sick(x_1))$
    - $P(Treat(x_1,m_1))$
    - $P(Nat(d_1))$
    - $P(Man(w_1))$
    - $P(Epid)$
  - Test trade-off (overhead vs. faster inference)

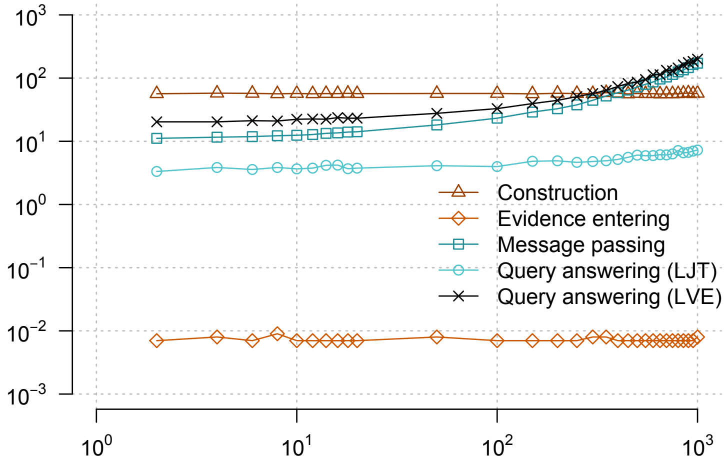


- Test increasing
  - Ground width w_g
    - Default: 3
  - Counting width $w_\#$
    - Default: 1
  - Number of nodes n_J
    - Default: 3
  - Domain size n
    - Default: 1000
  - Based on

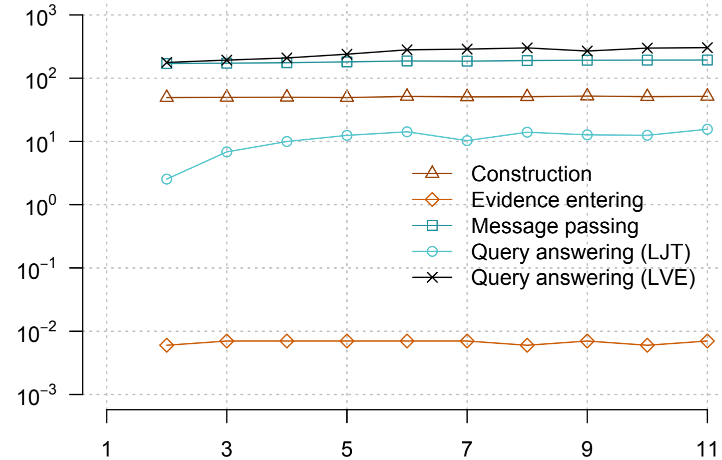
$$O\left(n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#}\right)$$

Step-wise

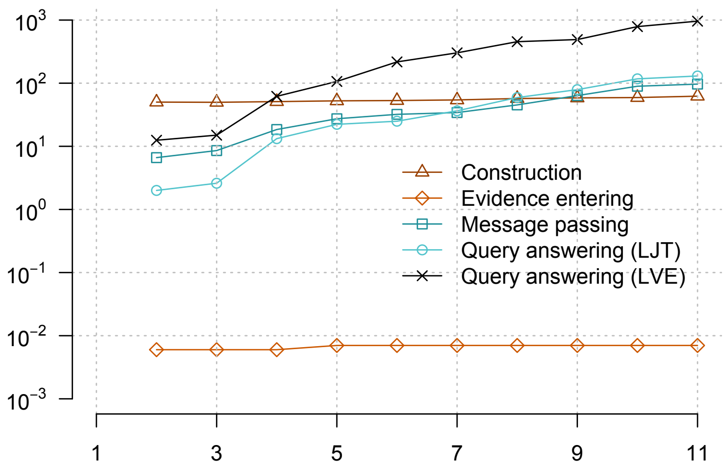
$O\left(n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#}\right)$



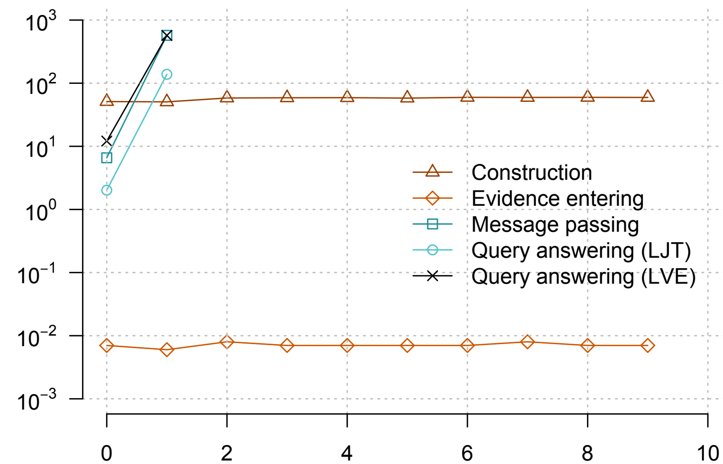
n ranging from 2 to 1000



n_J ranging from 2 to 11



w_g ranging from 2 to 11



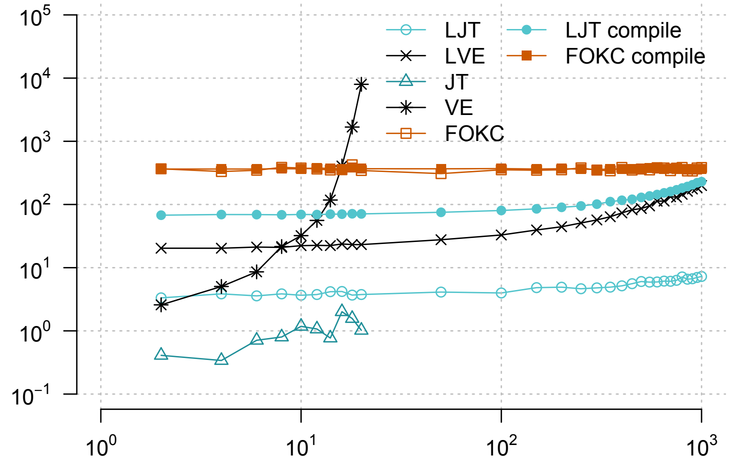
$w_\#$ ranging from 0 to 9

Runtimes in milliseconds

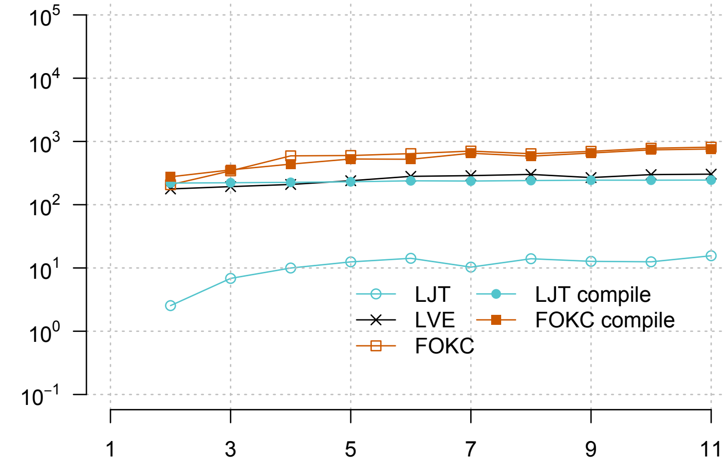
Default: $n = 1000, n_J = 3, w_g = 3, w_\# = 1$

Queries answering

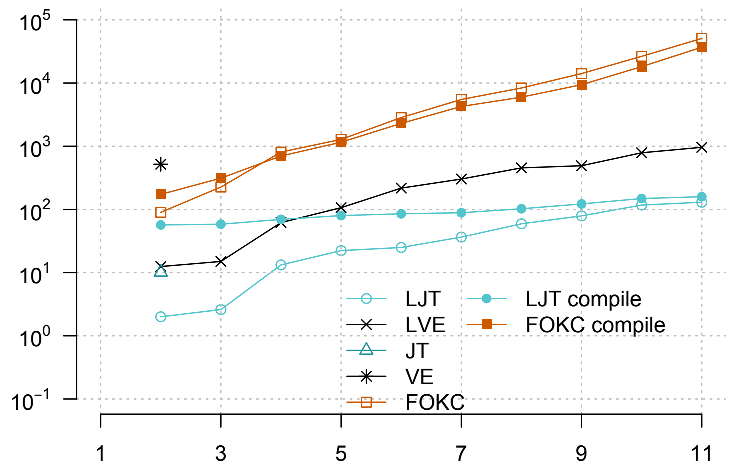
FOKC: see next lecture
compile: all overhead time



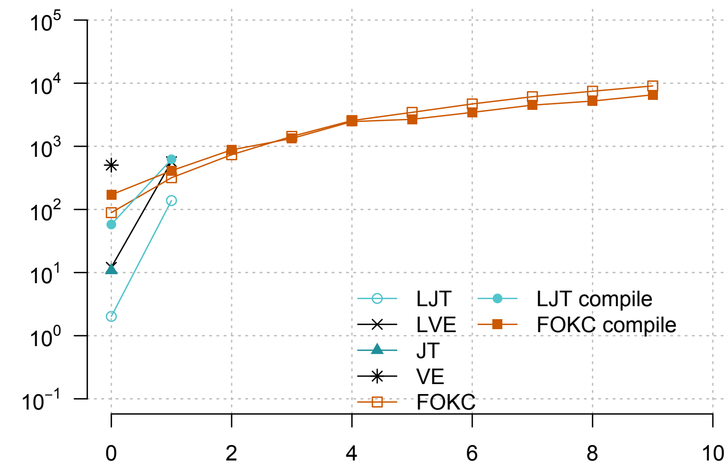
n ranging from 2 to 1000



n_J ranging from 2 to 11



w_g ranging from 2 to 11



$w_\#$ ranging from 0 to 9

Runtimes in milliseconds

Default: $n = 1000, n_J = 3, w_g = 3, w_\# = 1$

Trade-off Evaluation: Criteria

- For multi-query algorithms
  - Overhead to set off (model is *compiled* into a helper structure)

  vs.
  - Shorter individual query answering time
- With
  - $t_{q,cpl}$ runtime for answering single query with an algorithm that uses compilation
  - $t_{q,uncpl}$ runtime for answering single query with an algorithm without compilation
  - $t_{c,cpl}$ runtime for compilation with an algorithm that uses compilation
  - What is the ratio between individual query answering times?

$$\alpha = \frac{t_{q,cpl}}{t_{q,uncpl}}$$

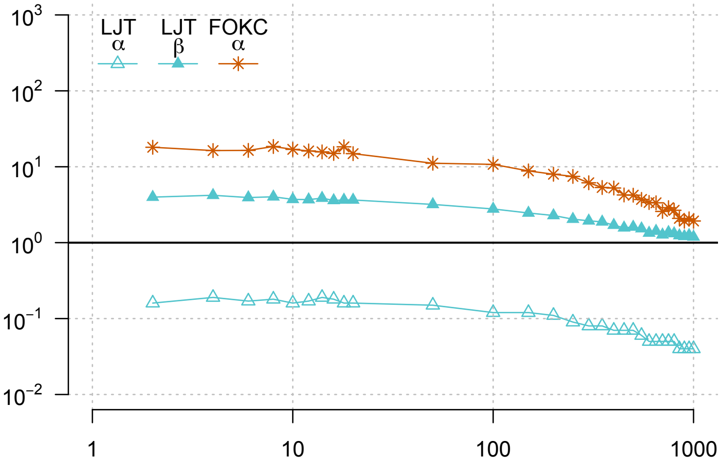
  - How many queries does it take to offset the overhead?

$$\beta = \frac{t_{c,cpl}}{t_{q,uncpl} - t_{q,cpl}}$$

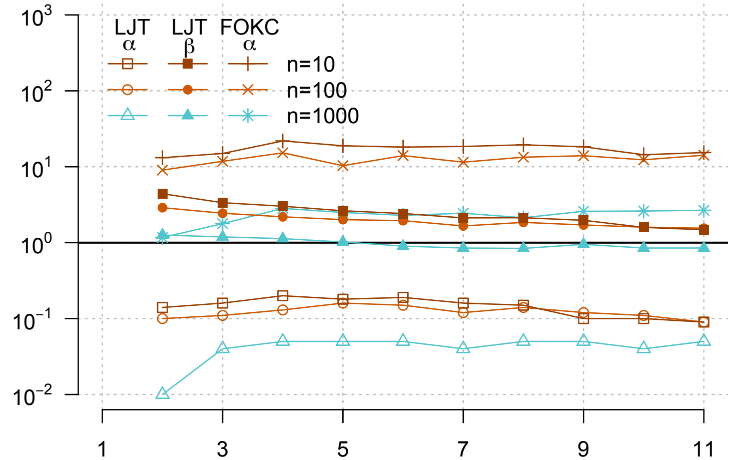
    - Makes only sense if $\alpha > 1$

Trade-off

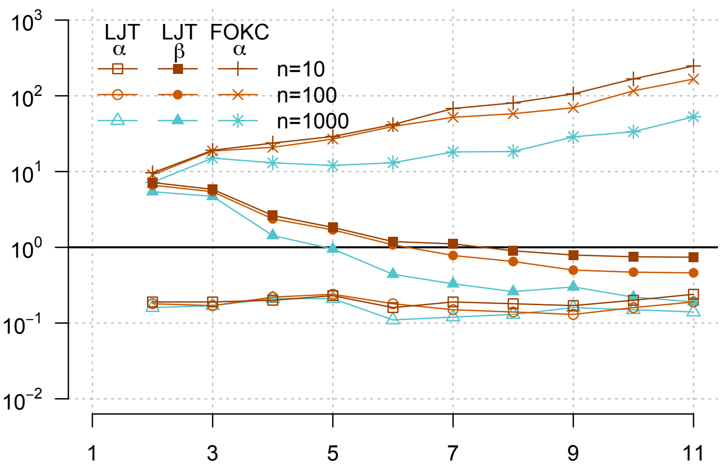
FOKC: see next lecture



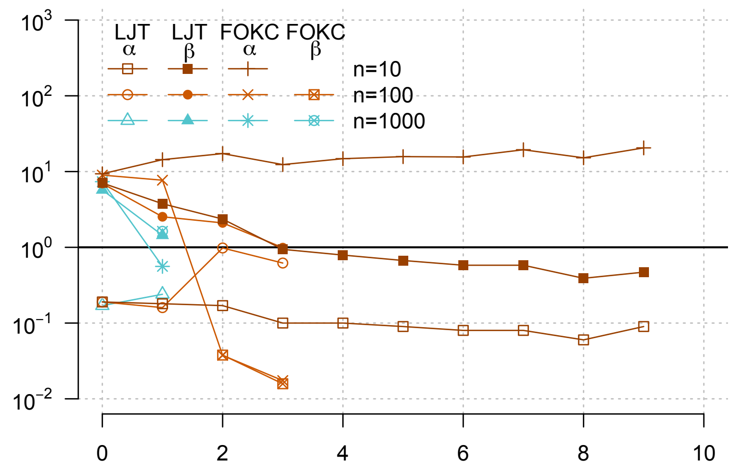
n ranging from 2 to 1000



n_J ranging from 2 to 11



w_g ranging from 2 to 11



$w_\#$ ranging from 0 to 9

Default: $n_J = 3, w_g = 3, w_\# = 1$

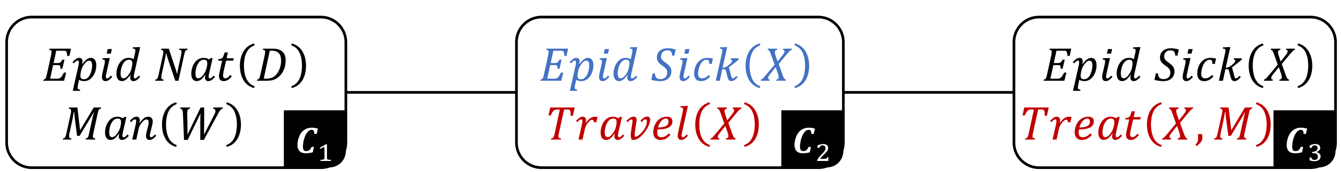
Beyond Standard LJT

- LJT is basically a framework for query answering that is independent of
  - Specific function encoding → calculating algorithm has to work with the encoding
    - Such as lists, tables, ADDs, etc.
  - Concrete query language → whatever the calculating algorithm can handle, LJT can (*within parclusters*)
    - E.g., with LVE, queries with
      - Uncertain evidence
      - Parameterised query terms
    - One exception: conjunctive queries!

⇒ Could use any other query answering algorithm for calculations as long as the query answering algorithm can handle message calculations

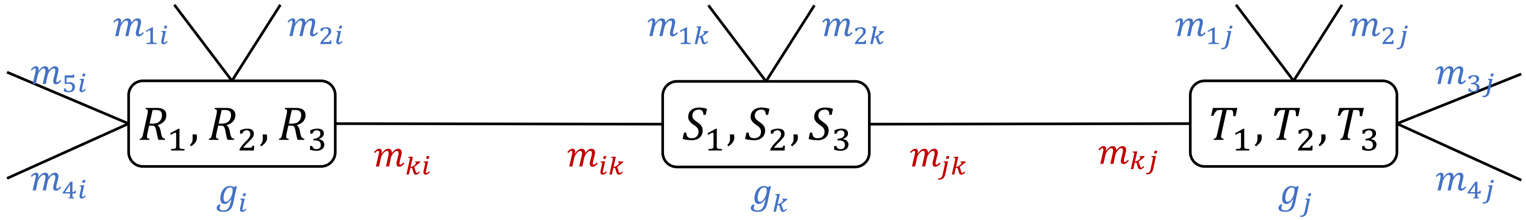
LJT for Conjunctive Queries

- Problem if query terms occur outside of one parcluster
  - E.g., with the FO jtree below
    - $P(Epid, Sick(eve))$ ✓
    - $P(Travel(eve), Treat(eve, m_1))$ ✗
- Solution:
  Temporarily merge parclusters such that the query terms occur in one parcluster



Parcluster Merging for Queries

- Find a subgraph J' of the FO jtree J for the query terms $\boldsymbol{Q}$ such that J' such that $\boldsymbol{Q} \subseteq rv(J')$
  - For query answering, use a set $G_{\boldsymbol{Q}}$ that consists of local models in J' and messages from outside J'
- Why subgraph?
  - Allows for ignoring messages within J' and including messages from outside J' into parclusters of J'
    - No duplicate information used
    - Messages reused as much as possible
  - E.g., consider subgraph of $\boldsymbol{C}_i, \boldsymbol{C}_k, \boldsymbol{C}_j$ for query on R_1, T_1
    - Take all outside messages and local models
    - Ignore inside messages $m_{ki}, m_{ik}, m_{jk}, m_{kj}$



Parcluster Merging for Queries

- Find a subgraph J' of the FO jtree J for the query terms $\boldsymbol{Q}$ such that J' such that $\boldsymbol{Q} \subseteq rv(J')$
- Subgraph should be minimal for optimal performance, i.e., a minimisation problem to solve:

$$\underset{J'}{\operatorname{argmin}} |rv(J')|$$
$$\text{s.t. } \boldsymbol{Q} \subseteq rv(J')$$

  - Trade-off between finding a subgraph fast and finding a minimal one
    - It is not about the number of parclusters but the number of PRVs in the parclusters!

Parcluster Merging for Queries

- Find a subgraph J' of the FO jtree J for the query terms $\boldsymbol{Q}$ such that J' such that $\boldsymbol{Q} \subseteq rv(J')$
- Subgraph should be minimal for optimal performance, i.e., a minimisation problem to solve:

$$\underset{J'}{\operatorname{argmin}} |rv(J')|$$
$$\text{s.t. } \boldsymbol{Q} \subseteq rv(J')$$

  - Simple heuristics (without guarantees on optimality):
    - Start with one parcluster that fulfils $\boldsymbol{Q} \cap \boldsymbol{C}_i \neq \emptyset$ as J'
      - $\boldsymbol{Q}' = \boldsymbol{Q} \setminus \boldsymbol{C}_i$ remains as not covered by J'
    - Perform a breadth-first search starting at $\boldsymbol{C}_i$
      - Whenever a newly visited parcluster $\boldsymbol{C}_j$ fulfils $\boldsymbol{Q}' \cap \boldsymbol{C}_j \neq \emptyset$, add all parclusters on the path between $\boldsymbol{C}_i$ and $\boldsymbol{C}_j$ to J' (if not already part of J') and set $\boldsymbol{Q}' = \boldsymbol{Q}' \setminus \boldsymbol{C}_i$
    - until $\boldsymbol{Q}' = \emptyset$

Query Answering in FO jtrees

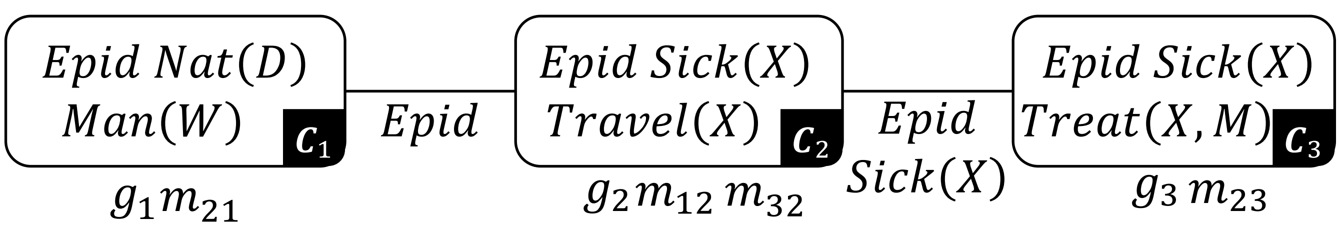
- Given an FO jtree J with messages passed
  - Prepared for query answering
- For each query with query terms $\boldsymbol{Q}$

Query Answering

  - Find subgraph J' s.t. $\boldsymbol{Q} \subseteq rv(J')$
  - Collect information from local models and outside messages, i.e,

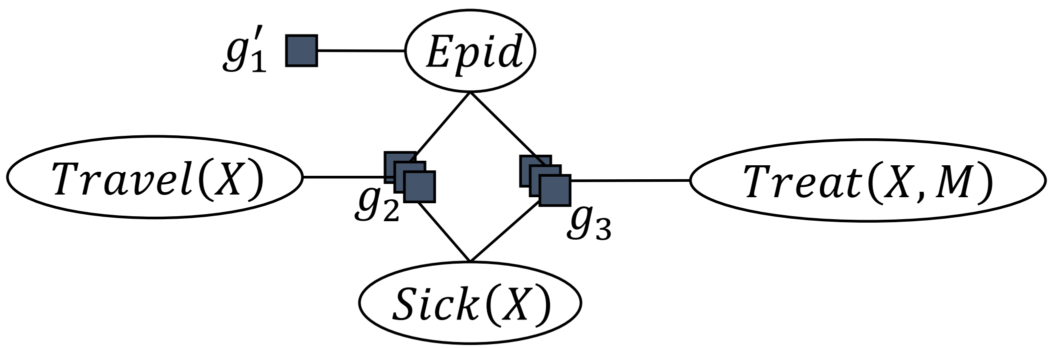
$$G_{\boldsymbol{Q}} = \bigcup_{i \in J'} G_i \cup \bigcup_{\substack{j \in nbs(i) \\ i \in J', j \notin J'}} m_{ji}$$

  - Call $\mathrm{LVE}\left(G_{\boldsymbol{Q}}, \boldsymbol{Q}, \emptyset\right)$ and return or store result of the call

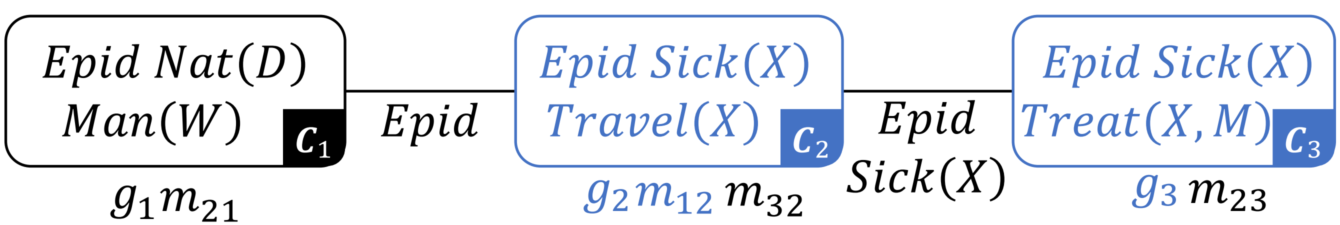


Example

- E.g., $P\big(Travel(eve), Treat(eve, m_1)\big)$
  - Subgraph: $\boldsymbol{C}_2, \boldsymbol{C}_3$
  - Submodel for query answering: $G_{\boldsymbol{Q}} = (g_2, g_3, m_{12})$

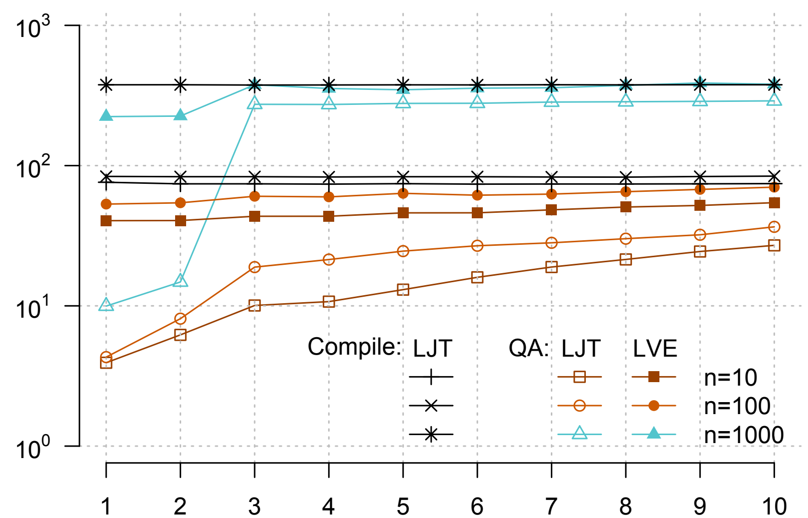


- Call LVE with $G_{\boldsymbol{Q}}$ and $\boldsymbol{Q} = \{Travel(eve), Treat(eve, m_1)\}$
  - Split off query terms
  - Eliminate all non-query terms
  - Normalise the result



Complexity & Runtimes

- With conjunctive queries, complexity for answering a single query depends on the size of the subtree $n_{J'}$
  - $O\left(n_{J'} \cdot \log_2(n) \cdot r^{w_g} \cdot n^{r_\# w_\#}\right)$
  - Assumption is that query terms occur close together and therefore $n_{J'}$ hopefully small
- Runtime behaviour observable in implementation
  - Increasing $n_{J'}$ on x-axis
  - Runtimes in milliseconds
  - More parclusters needed, runtimes increase
    - Closer to compile time
    - Closer to LVE time



Interim Summary

- Motivation
  - Find clusters that are enough for query answering
- FO jtree
  - From FO dtree clusters to FO jtree parclusters
- LJT algorithm
  - Propagation/message passing: Dynamic programming
- Complexity
  - Compared to LVE
    - Overhead for construction, message passing
    - Savings during query answering
- Completeness
  - Results for LVE hold as well
- Implementation
- Conjunctive queries
  - Find subgraph covering the query terms