EE512A – Advanced Inference in Graphical Models

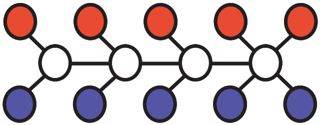
— Fall Quarter, Lecture 5 —

http://j.ee.washington.edu/~bilmes/classes/ee512a_fall_2014/

Prof. Jeff Bilmes

University of Washington, Seattle
Department of Electrical Engineering
http://melodi.ee.washington.edu/~bilmes

Oct 13th, 2014



Announcements

- Reading assignments, posted to our canvas announcements page (https://canvas.uw.edu/courses/914697/announcements): intro.pdf, ugms.pdf on undirected graphical models, and tree_inference.pdf on trees.
- Slides from previous time this course was offered are at our previous web page (http://j.ee.washington.edu/~bilmes/classes/ee512a_fall_2011/) and even earlier at http://melodi.ee.washington.edu/~bilmes/ee512fa09/.

Class Road Map - EE512a

- L1 (9/29): Introduction, Families, Semantics
- L2 (10/1): MRFs, elimination, Inference on Trees
- L3 (10/6): Tree inference, message passing, more general queries, non-tree)
- L4 (10/8): Non-trees, perfect elimination, triangulated graphs
- L5 (10/13): Triangulated Graphs, Triangulation, Multiple queries, Junction Trees
- L6 (10/15):
- L7 (10/20):
- L8 (10/22):
- L9 (10/27):
- L10 (10/29):
- L11 (11/3):
- L12 (11/5):
- L13 (11/10):
- L14 (11/12):
- L15 (11/17):
- L16 (11/19):
- L17 (11/24):
- L18 (11/26):
- L19 (12/1):
- L20 (12/3):
- Final Presentations: (12/10):

Finals Week: Dec 8th-12th, 2014.

Today

To tree, or not to tree, that is the question:

Today

To tree, or not to tree, that is the question:
Whether 'tis nobler in the mind to suffer

Today

To tree, or not to tree, that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of nontriangulated models,

Today

To tree, or not to tree, that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of nontriangulated models,
Or to take arms against a sea of cycles,

Today

To tree, or not to tree, that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of nontriangulated models,
Or to take arms against a sea of cycles,
And by opposing

Today

To tree, or not to tree, that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of nontriangulated models,
Or to take arms against a sea of cycles,
And by opposing chord them?

Neighbors of v same in original and reconstituted graph

Lemma 5.2.2

When elimination is run for a second time on the reconstituted graph with the same order, the set of neighbors v at the time v is eliminated is the same in both the original and in the reconstituted graph.

Proof.

Any neighbor of v in the reconstituted graph must be either an original-graph edge, or it must be due to a fill-in edge between v and some other node that is not an original graph neighbor. All of the fill-in neighbors must be due to elimination of nodes before v since after v is eliminated no new neighbors can be added to v. But the point at which v is eliminated in the original graph and the point at which it v is eliminated in the reconstituted graph, the same previous set of nodes have been eliminated, so any neighbors of v in the reconstituted graph will have been already added to the original graph when v is eliminated in the original graph.

Complexity of elimination process

Lemma 5.2.2

Given an elimination order, the computational complexity of the elimination process is $O(r^{k+1})$ where k is the largest set of neighbors encountered during elimination. This is the size of the largest clique in the reconstituted graph.

Proof.

First, when we eliminate σ_i in G_{i-1}, eliminating variable v when it is in the context of its current neighbors will cost $O(r^\ell)$ where $\ell = |\delta_{G_{i-1}}(v) + 1|$ — thus, the overall cost will be $O(r^{k+1})$.
Next, we show that largest clique in the reconstituted graph is equal to the complexity. Consider the reconstituted graph, and assume its largest clique is of size $k + 1$. When we re-run elimination on this graph, there will be no fill in. . . .

Perfect elimination graphs

- Since such graphs are inevitable, lets define them and give them a name

Definition 5.2.2 (perfect elimination graph)

A graph $G = (V, E)$ is a *perfect elimination graph* if there exists an ordering σ of the nodes such that eliminating nodes in G based on σ produces no fill-in edges.

- any perfect elimination ordering on a perfect elimination graph will have complexity exponential in the size of the largest clique in that graph

Maxcliques of perfect elimination graphs

Lemma 5.2.2

When running the elimination algorithm, all maxcliques in the resulting reconstituted graph are encountered as elimination cliques during elimination.

Proof.

Each elimination step produces a clique, but not necessarily a maxclique. Set of maxcliques in the resulting reconstituted perfect elimination graph is a subset of the set of cliques encountered during elimination. This is because of the neighbor property proven above in Lemma **??** — if there was a maxclique in the reconstituted graph that was not one of the elimination cliques, that maxclique would be encountered on a run of elimination with the same order on the reconstituted graph, but for the first variable to encounter this maxclique, it would have the same set of neighbors in original graph, contradicting the fact that it was not one of the elimination cliques. □

Finding the maxcliques of G'

Lemma 5.2.2

Given a graph G, an order σ, and a reconstituted graph G', the elimination algorithm can produce the set of maxcliques in G'.

Proof.

Consider node v's elimination clique c_v (i.e., v along with its neighbors $\delta(v)$ at the time of elimination of v). Since c_v is complete, either c_v is a maxclique or a subset of some maxclique. c_v can not be a subset of any subsequently encountered maxcliques since all such future maxcliques would not involve v. Therefore c_v must be a maxclique or a subset of some previously encountered maxclique. If c_v is not a subset of some previously encountered maxclique, it must be a maxclique (we add c_v to a list of maxcliques). Since all maxcliques are encountered as elimination cliques, all maxcliques are discovered in this way. $\square$

Embedding

Definition 5.2.3 (embedding)

Any graph $G = (V, E)$ can be embedded into a graph $G' = (V, E')$ if G is a spanning subgraph of G', meaning that $E \subseteq E'$.

- Embedding never shrinks family of distributions
- Any G may be embedded into G_σ.
- We wish to embed G into the class of perfect elimination graphs (this is a subset of all undirected graphs).
- Does this restrict us in any way? (e.g., remove family members?)
- Does it change values of resulting queries we wish to compute?
- No, only potential issue is computation.
- Graphical model structure learning would be: start with $p \in \mathcal{F}(G, \mathcal{M}^{(\mathrm{f})})$, find some spanning subgraph $G' = (V, E')$ where $E' \subset E$, and solve inference there for a $p' \in \mathcal{F}(G', \mathcal{M}^{(\mathrm{f})})$ that is as close as possible to p. We defer this topic until later in the course.

Triangulated Graphs

Definition 5.2.3 (Triangulated graph)

A graph G is triangulated (equivalently chordal) if all cycles have a chord.

- in triangulated graph: any cycles of length > 3 must have a chord.
- Cycles of length 3 have no non-adjacent vertices
- Triangulated graphs include
  1. a clique is a triangulated graph (all cycles have chord).
  2. a tree is a triangulated graph, since there are no cycles that could disobey the chordal requirement.
  3. a chain is a triangulated graph, since it is a tree.
  4. a set of disconnected vertices is triangulated (since there are no cycles).

Triangulated Graphs

Theorem 5.2.5

Given graph G, elimination order σ, and perfect elimination graph $G' = G_\sigma$ obtained by elimination on G. We may reconstruct a perfect elimination order (w.r.t. G_σ) from G_σ by repeatedly choosing any simplicial node and eliminating it. Call this new order σ'. Now σ' might not be the same order as σ, but both are perfect elimination orders for G'.

Proof.

If there is more than one possible order, we must reach a point at which there are two possible simplicial nodes $u, v \in G'$. Eliminating u does not render v non-simplicial since no edges are added and thus v has if anything only a reduced set of neighbors. Each time we eliminate a simplicial node, any other node that was simplicial in the original elimination order stays simplicial when it comes time to eliminate it. $\square$

Triangulated graphs and minimal separators

Lemma 5.2.6

A graph $G = (V, E)$ is triangulated iff all minimal separators are complete.

Proof.

First, suppose all minimal separators in $G = (V, E)$ are complete. Consider any cycle $u, v, w, x_1, x_2, \ldots, x_k, u$ starting and ending at node u, where $k \geq 1$. Then the pair v, x_i for some $i \in \{1, \ldots, k\}$ must be part of a minimal (u, w)-separator, which is complete, so v and that x_i are connected thereby creating a chord in the cycle. The cycle is arbitrary, so all cycles are chorded. ...

Triangulated graphs have at least two simplicial nodes.

We also have the following important theorem.

Lemma 5.3.1

A triangulated graph on $n \geq 2$ nodes is either a clique, or there are two non-adjacent nodes that are simplicial.

Note that this appears to be very much like the property of a tree where a simplicial node takes the role of a leaf-node. Is this a coincidence?

Triangulated graphs have at least two simplicial nodes.

Proof of Theorem 5.3.1.

Any clique is triangulated and all nodes are simplicial, so assume the graph is not a clique. Induction on $n = |V(G)|$: any graph with $1 < n \leq 3$ is triangulated and has two simplicial nodes. Assume true for $n-1$ nodes, and show for n nodes. Let a and b be two non-adjacent vertices, let S be a minimal (a,b)-separator which must be complete. Let G_A and G_B be the connected components of $G[V \setminus S]$ containing respectively a and b. Let $A = V(G_A)$ and $B = V(G_B)$. By induction, $G[A \cup S]$ and $G[B \cup S]$ are either cliques, or contain two non-adjacent simplicial vertices. First case, all nodes are simplicial, second case both simplicial non-adjacent vertices cannot be in S since S is complete. In all cases, we may choose two non-adjacent simplicial vertices, one each in A and B, and these vertices are adjacent to no nodes other than $A \cup S$ and $B \cup S$ respectively. These nodes remain simplicial and non-adjacent in G. □

Recap so far

- Non-tree graphs: effectively doing inference on perfect elimination graph.

Recap so far

- Non-tree graphs: effectively doing inference on perfect elimination graph.
- After elimination, we've got a perfect (fill-in free) elimination graph.

Recap so far

- Non-tree graphs: effectively doing inference on perfect elimination graph.
- After elimination, we've got a perfect (fill-in free) elimination graph.
- We encounter the maxcliques when we run elimination.

Recap so far

- Non-tree graphs: effectively doing inference on perfect elimination graph.
- After elimination, we've got a perfect (fill-in free) elimination graph.
- We encounter the maxcliques when we run elimination.
- Elimination cliques are supserset of set of maxcliques.

Recap so far

- Non-tree graphs: effectively doing inference on perfect elimination graph.
- After elimination, we've got a perfect (fill-in free) elimination graph.
- We encounter the maxcliques when we run elimination.
- Elimination cliques are supserset of set of maxcliques.
- We may embed any $G = (V, E)$ into any $G = (V, E \cup F)$.

Recap so far

- Non-tree graphs: effectively doing inference on perfect elimination graph.
- After elimination, we've got a perfect (fill-in free) elimination graph.
- We encounter the maxcliques when we run elimination.
- Elimination cliques are supserset of set of maxcliques.
- We may embed any $G = (V, E)$ into any $G = (V, E \cup F)$.
- Given perfect elimination graph, easy to find perfect elimination order.

Recap so far

- Non-tree graphs: effectively doing inference on perfect elimination graph.
- After elimination, we've got a perfect (fill-in free) elimination graph.
- We encounter the maxcliques when we run elimination.
- Elimination cliques are supserset of set of maxcliques.
- We may embed any $G = (V, E)$ into any $G = (V, E \cup F)$.
- Given perfect elimination graph, easy to find perfect elimination order.
- Triangulated graphs (chordal), all cycles are chorded.

Recap so far

- Non-tree graphs: effectively doing inference on perfect elimination graph.
- After elimination, we've got a perfect (fill-in free) elimination graph.
- We encounter the maxcliques when we run elimination.
- Elimination cliques are supserset of set of maxcliques.
- We may embed any $G = (V, E)$ into any $G = (V, E \cup F)$.
- Given perfect elimination graph, easy to find perfect elimination order.
- Triangulated graphs (chordal), all cycles are chorded.
- Various definitions of separators.

Recap so far

- Non-tree graphs: effectively doing inference on perfect elimination graph.
- After elimination, we've got a perfect (fill-in free) elimination graph.
- We encounter the maxcliques when we run elimination.
- Elimination cliques are supserset of set of maxcliques.
- We may embed any $G = (V, E)$ into any $G = (V, E \cup F)$.
- Given perfect elimination graph, easy to find perfect elimination order.
- Triangulated graphs (chordal), all cycles are chorded.
- Various definitions of separators.
- Triangulated iff all min separators are complete.

Recap so far

- Non-tree graphs: effectively doing inference on perfect elimination graph.
- After elimination, we've got a perfect (fill-in free) elimination graph.
- We encounter the maxcliques when we run elimination.
- Elimination cliques are supserset of set of maxcliques.
- We may embed any $G = (V, E)$ into any $G = (V, E \cup F)$.
- Given perfect elimination graph, easy to find perfect elimination order.
- Triangulated graphs (chordal), all cycles are chorded.
- Various definitions of separators.
- Triangulated iff all min separators are complete.
- Any triangulated graph on ≥ 2 nodes has two simplicial nodes.

Triangulated/Elimination

- In a triangulated graphs, all nodes simplicial?

Triangulated/Elimination

- In a triangulated graphs, all nodes simplicial?
- If G is triangulated and v simplicial, if we eliminate v, is $G[V \setminus v]$ still triangulated?

Triangulated/Elimination

- In a triangulated graphs, all nodes simplicial?
- If G is triangulated and v simplicial, if we eliminate v, is $G[V \setminus v]$ still triangulated?
- Therefore:

Corollary 5.3.2

For any triangulated graph, there exists an elimination order that does not produce any fill in.

So if we know the graph is triangulated, we can easily find a perfect elimination order. Why?

Triangulated/Elimination

- In a triangulated graphs, all nodes simplicial?
- If G is triangulated and v simplicial, if we eliminate v, is $G[V \setminus v]$ still triangulated?
- Therefore:

Corollary 5.3.2

For any triangulated graph, there exists an elimination order that does not produce any fill in.

So if we know the graph is triangulated, we can easily find a perfect elimination order. Why? We can strengthen the above in fact:

Triangulated vs. Perfect elimination graphs

Lemma 5.3.3

If G is a graph and there exists a perfect elimination order, then G is triangulated.

Triangulated vs. Perfect elimination graphs

Lemma 5.3.3

If G is a graph and there exists a perfect elimination order, then G is triangulated.

Proof.

By induction. It is obviously true for 1 and 2 nodes. Assume true for n nodes, and we are given an $n+1$ node graph. Since there exists an elimination order without fill-in, there exists a simplicial node, where chordless cycles can not exist through that node since all of its neighbors are connected. Once we eliminate that node, no fill-in is created, and induction step applies. □

Triangulated vs. Perfect elimination graphs

Lemma 5.3.3

If G is a graph and there exists a perfect elimination order, then G is triangulated.

Proof.

By induction. It is obviously true for 1 and 2 nodes. Assume true for n nodes, and we are given an $n + 1$ node graph. Since there exists an elimination order without fill-in, there exists a simplicial node, where chordless cycles can not exist through that node since all of its neighbors are connected. Once we eliminate that node, no fill-in is created, and induction step applies. □

We summarize the bijection as follows:

Triangulated vs. Perfect elimination graphs

Lemma 5.3.3

If G is a graph and there exists a perfect elimination order, then G is triangulated.

Proof.

By induction. It is obviously true for 1 and 2 nodes. Assume true for n nodes, and we are given an $n+1$ node graph. Since there exists an elimination order without fill-in, there exists a simplicial node, where chordless cycles can not exist through that node since all of its neighbors are connected. Once we eliminate that node, no fill-in is created, and induction step applies. □

We summarize the bijection as follows:

Theorem 5.3.4

A graph G is triangulated iff there exists a perfect elimination order over the nodes in G.

Triangulated vs. Perfect elimination graphs

Corollary 5.3.5

Take any graph G and an elimination order σ, then the reconstituted graph $G' = (V, E \cup F_\sigma)$ is triangulated.

Triangulated vs. Perfect elimination graphs

Generating triangulated graphs

- Therefore, we can generate a reconstituted elimination graph (or any triangulated graph) using a reverse elimination order.

Algorithm 1: Regenerate triangulated graph.

Input: A triangulated graph $G = (V, E)$ and a perfect elimination order σ
Result: A new graph G' identical to G.

```
Recall that δ_{G_{i-1}}(σ_i) are neighbors of σ_i in G at the point σ_i is eliminated. ;
Start out with V(G') empty ;
Add σ_N to V(G') ;
for i = N − 1 ... 1 do
    Add σ_i to V(G') ;
    Add δ_{G_{i-1}}(σ_i) to E(G') ;          /* at this δ_{G_{i-1}}(σ_i) is complete */
```

Recall that $\delta_{G_{i-1}}(\sigma_i)$ are neighbors of σ_i in G at the point σ_i is eliminated. ;
Start out with $V(G')$ empty ;
Add σ_N to $V(G')$;
for $i = N - 1 \ldots 1$ **do**
  Add σ_i to $V(G')$;
  Add $\delta_{G_{i-1}}(\sigma_i)$ to $E(G')$; /* at this $\delta_{G_{i-1}}(\sigma_i)$ is complete */

Triangulated vs. Perfect elimination graphs

- Trees can be generated this way (recall one of the definitions)
- Does elimination span the space of all possible triangulations of a graph? (i.e., can any triangulation of G be obtained by some elimination order?)

Triangulated vs. Perfect elimination graphs

- Trees can be generated this way (recall one of the definitions)
- Does elimination span the space of all possible triangulations of a graph? (i.e., can any triangulation of G be obtained by some elimination order?)

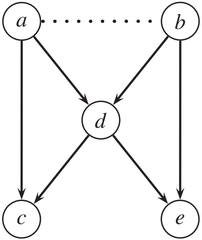
Theorem 5.3.6

Let $G = (V, E)$ be a graph and let $G' = (V, E \cup F)$ be a triangulation of G with F the required edge fill-in. If the triangulated graph is minimal *in the sense that for any $F' \subset F$, the graph $G'' = (V, E \cup F')$ is no longer triangulated, then F can be obtained by the result of an elimination order. That is, the elimination algorithm and the various variable orderings may produce all minimal triangulations of a graph G.*

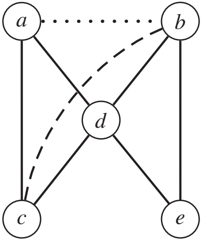
- Minimal triangulations are state-space optimal for positive distributions only!

Triangulated vs. Perfect elimination graphs

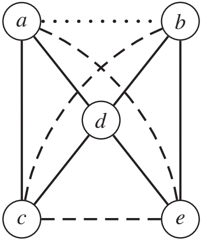
Minimal triangulations are state-space optimal for positive distributions only. Let d be a deterministic function of a and b. All variables have r values but d has $r^2 - 1$ values.



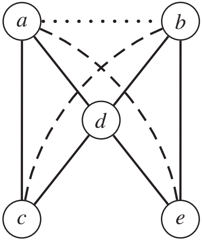
Moralized already chordal, perfect elim. order (c, a, b, d). One clique at $O(r^2)$, two at $O(r^4)$.



Elimination order (a, c, b, d), cost is still $O(r^4)$



Start by eliminating d, cost is still $O(r^4)$



Triangulation unobtainable with elimination, cost $O(r^3)$.

re-cap

- To compute marginals, we must run elimination of nodes.

re-cap

- To compute marginals, we must run elimination of nodes.
- Doing so necessarily produces a triangulated graph.

re-cap

- To compute marginals, we must run elimination of nodes.
- Doing so necessarily produces a triangulated graph.
- Complexity of this process is exponential in largest clique in result.

re-cap

- To compute marginals, we must run elimination of nodes.
- Doing so necessarily produces a triangulated graph.
- Complexity of this process is exponential in largest clique in result.
- We encounter the cliques (and the largest) during elimination so we get the complexity while we are doing elimination

re-cap

- To compute marginals, we must run elimination of nodes.
- Doing so necessarily produces a triangulated graph.
- Complexity of this process is exponential in largest clique in result.
- We encounter the cliques (and the largest) during elimination so we get the complexity while we are doing elimination
- Elimination adds edges, we can embed original graph into resulting triangulated graph (triangulated graph “covers” original graph)

re-cap

- To compute marginals, we must run elimination of nodes.
- Doing so necessarily produces a triangulated graph.
- Complexity of this process is exponential in largest clique in result.
- We encounter the cliques (and the largest) during elimination so we get the complexity while we are doing elimination
- Elimination adds edges, we can embed original graph into resulting triangulated graph (triangulated graph "covers" original graph)
- can't avoid a triangulated graph — always dealing with triangulated graphs implicitly or explicitly when doing elimination.

re-cap

- To compute marginals, we must run elimination of nodes.
- Doing so necessarily produces a triangulated graph.
- Complexity of this process is exponential in largest clique in result.
- We encounter the cliques (and the largest) during elimination so we get the complexity while we are doing elimination
- Elimination adds edges, we can embed original graph into resulting triangulated graph (triangulated graph "covers" original graph)
- can't avoid a triangulated graph — always dealing with triangulated graphs implicitly or explicitly when doing elimination.
- want to find minimally triangulated covering graph, one with smallest largest maxclique

re-cap

- To compute marginals, we must run elimination of nodes.
- Doing so necessarily produces a triangulated graph.
- Complexity of this process is exponential in largest clique in result.
- We encounter the cliques (and the largest) during elimination so we get the complexity while we are doing elimination
- Elimination adds edges, we can embed original graph into resulting triangulated graph (triangulated graph "covers" original graph)
- can't avoid a triangulated graph — always dealing with triangulated graphs implicitly or explicitly when doing elimination.
- want to find minimally triangulated covering graph, one with smallest largest maxclique
- i.e., find optimal elimination order

re-cap

- To compute marginals, we must run elimination of nodes.
- Doing so necessarily produces a triangulated graph.
- Complexity of this process is exponential in largest clique in result.
- We encounter the cliques (and the largest) during elimination so we get the complexity while we are doing elimination
- Elimination adds edges, we can embed original graph into resulting triangulated graph (triangulated graph “covers” original graph)
- can’t avoid a triangulated graph — always dealing with triangulated graphs implicitly or explicitly when doing elimination.
- want to find minimally triangulated covering graph, one with smallest largest maxclique
- i.e., find optimal elimination order
- there are $n!$ elimination orders

re-cap

- To compute marginals, we must run elimination of nodes.
- Doing so necessarily produces a triangulated graph.
- Complexity of this process is exponential in largest clique in result.
- We encounter the cliques (and the largest) during elimination so we get the complexity while we are doing elimination
- Elimination adds edges, we can embed original graph into resulting triangulated graph (triangulated graph "covers" original graph)
- can't avoid a triangulated graph — always dealing with triangulated graphs implicitly or explicitly when doing elimination.
- want to find minimally triangulated covering graph, one with smallest largest maxclique
- i.e., find optimal elimination order
- there are $n!$ elimination orders
- is this easy or hard?

re-cap

- To compute marginals, we must run elimination of nodes.
- Doing so necessarily produces a triangulated graph.
- Complexity of this process is exponential in largest clique in result.
- We encounter the cliques (and the largest) during elimination so we get the complexity while we are doing elimination
- Elimination adds edges, we can embed original graph into resulting triangulated graph (triangulated graph "covers" original graph)
- can't avoid a triangulated graph — always dealing with triangulated graphs implicitly or explicitly when doing elimination.
- want to find minimally triangulated covering graph, one with smallest largest maxclique
- i.e., find optimal elimination order
- there are $n!$ elimination orders
- is this easy or hard? We shall see . . .

k-trees

Generalizations of a tree as defined as follows:

Definition 5.3.7 (k-tree)

A complete graph with $k+1$ nodes is a k-tree. To construct a k tree with $n+1$ nodes starting from a k-tree with n nodes, choose some size k complete sub-graph of the n-node k-tree and connect the $n+1$'st node to all nodes in the k-node complete sub-graph.

k-trees

Generalizations of a tree as defined as follows:

Definition 5.3.7 (k-tree)

A complete graph with $k+1$ nodes is a k-tree. To construct a k tree with $n+1$ nodes starting from a k-tree with n nodes, choose some size k complete sub-graph of the n-node k-tree and connect the $n+1$'st node to all nodes in the k-node complete sub-graph.

- Any complete n-graph is an $n-1$-tree

k-trees

Generalizations of a tree as defined as follows:

Definition 5.3.7 (k-tree)

A complete graph with $k+1$ nodes is a k-tree. To construct a k tree with $n+1$ nodes starting from a k-tree with n nodes, choose some size k complete sub-graph of the n-node k-tree and connect the $n+1$'st node to all nodes in the k-node complete sub-graph.

- Any complete n-graph is an $n-1$-tree
- a regular tree is a 1-tree.

k-trees

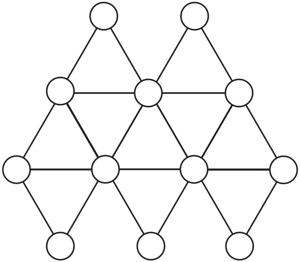
Generalizations of a tree as defined as follows:

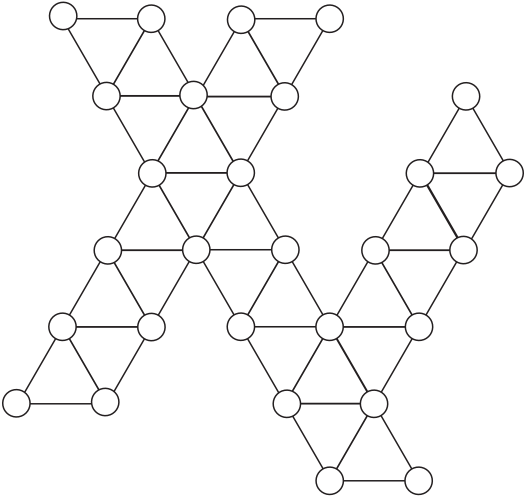
Definition 5.3.7 (k-tree)

A complete graph with $k+1$ nodes is a k-tree. To construct a k tree with $n+1$ nodes starting from a k-tree with n nodes, choose some size k complete sub-graph of the n-node k-tree and connect the $n+1$'st node to all nodes in the k-node complete sub-graph.

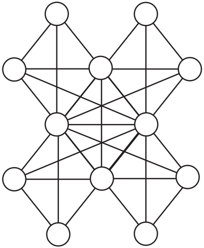
- Any complete n-graph is an $n-1$-tree
- a regular tree is a 1-tree.
- all k-trees are triangulated

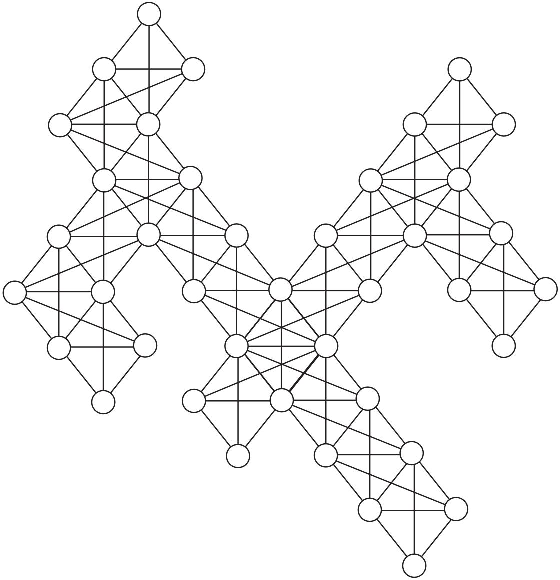
Example of 2-trees





Example of 3-trees





k-trees

- In a tree, all minimal separators are size 1

k-trees

- In a tree, all minimal separators are size 1
- In a k-tree, all minimal separators are size k (and thus a k-clique).

k-trees

- In a tree, all minimal separators are size 1
- In a k-tree, all minimal separators are size k (and thus a k-clique).
- In a k-tree, all maxcliques are size $k+1$, so maximum clique size is $k+1$.

k-trees

- In a tree, all minimal separators are size 1
- In a k-tree, all minimal separators are size k (and thus a k-clique).
- In a k-tree, all maxcliques are size $k+1$, so maximum clique size is $k+1$.
- In a k-tree, complexity of inference will be $O(r^{k+1})$.

k-trees

- In a tree, all minimal separators are size 1
- In a k-tree, all minimal separators are size k (and thus a k-clique).
- In a k-tree, all maxcliques are size $k + 1$, so maximum clique size is $k + 1$.
- In a k-tree, complexity of inference will be $O(r^{k+1})$.
- even stronger:

k-trees

- In a tree, all minimal separators are size 1
- In a k-tree, all minimal separators are size k (and thus a k-clique).
- In a k-tree, all maxcliques are size $k+1$, so maximum clique size is $k+1$.
- In a k-tree, complexity of inference will be $O(r^{k+1})$.
- even stronger:

Lemma 5.3.8

A graph $G = (V, E)$ is a k-tree iff

- *G is connected*
- *G's maximum clique is of size $k+1$*
- *Every minimal separator of G is a k-clique.*

k-trees

Definition 5.3.9 (partial k-tree)

Any spanning sub-graph of a k-tree is a partial k-tree.

- Any partial k-tree is embeddable into a k-tree.

k-trees

Definition 5.3.9 (partial k-tree)

Any spanning sub-graph of a k-tree is a partial k-tree.

- Any partial k-tree is embeddable into a k-tree.
- Inference in a partial k-tree is at most $O(r^{k+1})$.

k-trees

Definition 5.3.9 (partial k-tree)

Any spanning sub-graph of a k-tree is a partial k-tree.

- Any partial k-tree is embeddable into a k-tree.
- Inference in a partial k-tree is at most $O(r^{k+1})$.
- k-trees are triangulated, but arbitrary triangulated graph not necessarily a k-tree

k-trees

Definition 5.3.9 (partial k-tree)

Any spanning sub-graph of a k-tree is a partial k-tree.

- Any partial k-tree is embeddable into a k-tree.
- Inference in a partial k-tree is at most $O(r^{k+1})$.
- k-trees are triangulated, but arbitrary triangulated graph not necessarily a k-tree
- any triangulated graph can be embedded into a k tree for large enough k — silly example, set $k = (n - 1)$.

k-trees

Definition 5.3.9 (partial k-tree)

Any spanning sub-graph of a k-tree is a partial k-tree.

- Any partial k-tree is embeddable into a k-tree.
- Inference in a partial k-tree is at most $O(r^{k+1})$.
- k-trees are triangulated, but arbitrary triangulated graph not necessarily a k-tree
- any triangulated graph can be embedded into a k tree for large enough k — silly example, set $k = (n - 1)$.
- But is it possible for smaller k?

k-trees and embeddings

Lemma 5.3.10

If G is a triangulated graph with at least $k+1$ vertices and has a maximum clique of size at most $k+1$, then G can be embedded into a k-tree.

Proof.

Let $\sigma = (\sigma_1, \ldots, \sigma_n)$ be perfect elim. order for G. We embed G into k-tree by adding edges to G so that same ordering is perfect in the k-tree. Induction.
Base case, any set of $k+1$ vertices can be embedded into a k tree by making those vertices a clique. Thus, add edges to last $k+1$ eliminated vertices, i.e., make $\{\sigma_{n-k}, \sigma_{n-k+1}, \ldots, \sigma_n\}$ a $k+1$-clique. ...

cont.

... proof continued.

Induction: assume the subgraph with vertices $\{\sigma_{i+1}, \ldots, \sigma_n\}$ has been embedded into a k-tree T_{i+1}. Since the maximum clique size of G is $k+1$, in G vertex σ_i is adjacent to a clique c with no more than k vertices in $\{\sigma_{i+1}, \ldots, \sigma_n\}$. In the k-tree T_{i+1}, c is contained in a k-clique c'. When we make σ_i adjacent to all of the vertices of c', we obtain a k-tree T_i since σ_i is still simplicial in T_i. Repeating to σ_1 and result is supergraph of G with same order being perfect. □

k-trees and embeddings

- Therefore, reconstituted elimination graph can be embedded into a k-tree for large enough k.

k-trees and embeddings

- Therefore, reconstituted elimination graph can be embedded into a k-tree for large enough k.
- Note: in k-tree all cliques are size $k+1$. In our reconstituted perfect elimination graph, might only have one clique that is of size $k+1$, so k-tree embedding migth add a lot of $k+1$ cliques (but this doen't change exponential cost dependence on largest clique size).

k-trees and embeddings

- Therefore, reconstituted elimination graph can be embedded into a k-tree for large enough k.
- Note: in k-tree all cliques are size $k+1$. In our reconstituted perfect elimination graph, might only have one clique that is of size $k+1$, so k-tree embedding migth add a lot of $k+1$ cliques (but this doen't change exponential cost dependence on largest clique size).
- Goal: We want to find the elimination order σ that results in the smallest k such that $G' = (V, E \cup F_\sigma)$ can be embedded into a k-tree.

k-trees and embeddings

- Therefore, reconstituted elimination graph can be embedded into a k-tree for large enough k.
- Note: in k-tree all cliques are size $k+1$. In our reconstituted perfect elimination graph, might only have one clique that is of size $k+1$, so k-tree embedding migth add a lot of $k+1$ cliques (but this doen't change exponential cost dependence on largest clique size).
- Goal: We want to find the elimination order σ that results in the smallest k such that $G' = (V, E \cup F_\sigma)$ can be embedded into a k-tree.
- i.e.,find best "Chordal cover"

k-trees and embeddings

- Goal: We want to find the elimination order σ that results in the smallest k such that $G' = (V, E \cup F_\sigma)$ can be embedded into a k-tree, i.e.,find best "Chordal cover"

k-trees and embeddings

- Goal: We want to find the elimination order σ that results in the smallest k such that $G' = (V, E \cup F_\sigma)$ can be embedded into a k-tree, i.e.,find best "Chordal cover"

Theorem 5.3.11

For an arbitrary graph $G = (V, E)$, finding the smallest k such that G can be embedded into a k-tree is an NP-complete optimization problem (i.e., the decision version of the problem, asking if G can be embedded into a k-tree of size k, is NP-complete).

k-trees and embeddings

- Goal: We want to find the elimination order σ that results in the smallest k such that $G' = (V, E \cup F_\sigma)$ can be embedded into a k-tree, i.e.,find best "Chordal cover"

Theorem 5.3.11

For an arbitrary graph $G = (V, E)$, finding the smallest k such that G can be embedded into a k-tree is an NP-complete optimization problem (i.e., the decision version of the problem, asking if G can be embedded into a k-tree of size k, is NP-complete).

- consider again elimination as summing out variables - not possible to guarantee optimal summation in poly-time order unless P=NP.

k-trees and embeddings

- Goal: We want to find the elimination order σ that results in the smallest k such that $G' = (V, E \cup F_\sigma)$ can be embedded into a k-tree, i.e.,find best "Chordal cover"

Theorem 5.3.11

For an arbitrary graph $G = (V, E)$, finding the smallest k such that G can be embedded into a k-tree is an NP-complete optimization problem (i.e., the decision version of the problem, asking if G can be embedded into a k-tree of size k, is NP-complete).

- consider again elimination as summing out variables - not possible to guarantee optimal summation in poly-time order unless P=NP.
- We resort to heuristics (min fill, min size, random chose from top ℓ with random restarts, etc. work well).

Heuristics for elimination

Since we can't expect to find a perfect elimination order, we have heuristics:

1. **min fill-in heuristic:**. Eliminate next the node n that would result in the smallest number of fill-in edges at that step. Break ties arbitrarily.

Heuristics for elimination

Since we can't expect to find a perfect elimination order, we have heuristics:

1. **min fill-in heuristic:**. Eliminate next the node n that would result in the smallest number of fill-in edges at that step. Break ties arbitrarily.
2. **min size heuristic:** Eliminate next the node that would result in the smallest clique when eliminated (i.e., choose the node as one with the smallest edge degree). Break ties arbitrarily.

Heuristics for elimination

Since we can't expect to find a perfect elimination order, we have heuristics:

1. **min fill-in heuristic:**. Eliminate next the node n that would result in the smallest number of fill-in edges at that step. Break ties arbitrarily.
2. **min size heuristic:** Eliminate next the node that would result in the smallest clique when eliminated (i.e., choose the node as one with the smallest edge degree). Break ties arbitrarily.
3. **min weight heuristic:** If the nodes have non-uniform domain sizes, then we choose next the node that would result in the clique with the smallest state space, which is defined as the product of the domain sizes. Break ties arbitrarily.

Better Heuristics for elimination

Variants and improvements to the above heuristics.

1. **tie-breaking:** When one heuristic has tie, choose one of the other heuristics to break tie.

Better Heuristics for elimination

Variants and improvements to the above heuristics.

1. **tie-breaking:** When one heuristic has tie, choose one of the other heuristics to break tie.
2. **non-greedy:** Rather than greedily choosing best vertex, take the m-best vertices (e.g., the $m < n$ nodes that would result in, say, the smallest fill-in) and eliminate one of them.

Better Heuristics for elimination

Variants and improvements to the above heuristics.

1. **tie-breaking:** When one heuristic has tie, choose one of the other heuristics to break tie.
2. **non-greedy:** Rather than greedily choosing best vertex, take the m-best vertices (e.g., the $m < n$ nodes that would result in, say, the smallest fill-in) and eliminate one of them.
3. **random next step:** Create a distribution over those m-best vertices, where the probability formed by either: 1) uniform, or 2) inversely proportional to the greedy score (e.g., inverse fill-in). Draw from this distribution to choose node to eliminate.

Better Heuristics for elimination

Variants and improvements to the above heuristics.

1. **tie-breaking:** When one heuristic has tie, choose one of the other heuristics to break tie.
2. **non-greedy:** Rather than greedily choosing best vertex, take the m-best vertices (e.g., the $m < n$ nodes that would result in, say, the smallest fill-in) and eliminate one of them.
3. **random next step:** Create a distribution over those m-best vertices, where the probability formed by either: 1) uniform, or 2) inversely proportional to the greedy score (e.g., inverse fill-in). Draw from this distribution to choose node to eliminate.
4. **random repeats:** Run above heuristics multiple times, producing different elimination orders. Choose one that results in the smallest maximum clique size.

k-trees and embeddings

- Goal: We want to find the elimination order σ that results in the smallest k such that $G' = (V, E \cup F_\sigma)$ can be embedded into a k-tree, i.e.,find best "Chordal cover"

Theorem 5.3.11

For an arbitrary graph $G = (V, E)$, finding the smallest k such that G can be embedded into a k-tree is an NP-complete optimization problem (i.e., the decision version of the problem, asking if G can be embedded into a k-tree of size k, is NP-complete).

- consider again elimination as summing out variables - not possible to guarantee optimal summation in poly-time order unless P=NP.
- We resort to heuristics (min fill, min size, random chose from top ℓ with random restarts, etc. work well).
- Inapproximability result: (see below)

Other views of the difficulty

Class of related problems that indicate the difficulty were are in.

Theorem 5.4.1 (Maximum Clique)

Given an arbitrary graph $G = (V, E)$, find the largest clique $C \subseteq V(G)$ (where large is measured in terms of $|C|$) is an NP-complete optimization problem.

- We have an $f(n)$ approximation algorithm if a solution of an algorithm provides a value that is always at least the size of the largest clique divided by $f(n)$, i.e., SOL $\geq$ OPT$/f(n)$.

Other views of the difficulty

Class of related problems that indicate the difficulty were are in.

Theorem 5.4.1 (Maximum Clique)

Given an arbitrary graph $G = (V, E)$, find the largest clique $C \subseteq V(G)$ (where large is measured in terms of $|C|$) is an NP-complete optimization problem.

- We have an $f(n)$ approximation algorithm if a solution of an algorithm provides a value that is always at least the size of the largest clique divided by $f(n)$, i.e., SOL $\geq$ OPT$/f(n)$.
- Good news: possible to do no worse than $O(|V|/(\log|V|)^2)$ times size of true maximum size clique (Boppana & Halldórsson).

Other views of the difficulty

Class of related problems that indicate the difficulty were are in.

Theorem 5.4.1 (Maximum Clique)

Given an arbitrary graph $G = (V, E)$, find the largest clique $C \subseteq V(G)$ (where large is measured in terms of $|C|$) is an NP-complete optimization problem.

- We have an $f(n)$ approximation algorithm if a solution of an algorithm provides a value that is always at least the size of the largest clique divided by $f(n)$, i.e., SOL $\geq$ OPT$/f(n)$.
- Good news: possible to do no worse than $O(|V|/(\log|V|)^2)$ times size of true maximum size clique (Boppana & Halldórsson).
- Bad news: inapproximability, not possible to do better than $O(|V|^{1-\epsilon})$ for any $\epsilon > 0$ (Håstad 1999).

Other views of the difficulty

Class of related problems that indicate the difficulty were are in.

Theorem 5.4.1 (Maximum Clique)

Given an arbitrary graph $G = (V, E)$, find the largest clique $C \subseteq V(G)$ (where large is measured in terms of $|C|$) is an NP-complete optimization problem.

- We have an $f(n)$ approximation algorithm if a solution of an algorithm provides a value that is always at least the size of the largest clique divided by $f(n)$, i.e., SOL $\geq$ OPT$/f(n)$.
- Good news: possible to do no worse than $O(|V|/(\log|V|)^2)$ times size of true maximum size clique (Boppana & Halldórsson).
- Bad news: inapproximability, not possible to do better than $O(|V|^{1-\epsilon})$ for any $\epsilon > 0$ (Håstad 1999).
- If we could find the smallest k such that it could be embedded it a k tree, we could identify the maximum clique in the graph. How?

Another view of the difficulty

While we're at it, even finding best chordal fill-in is hard

Theorem 5.4.2

Given an arbitrary graph $G = (V, E)$, *and* $G' = (V, E \cup F)$ *is a triangulation of* G, *finding the smallest such* F *is an NP-complete optimization problem.*

Another view of the difficulty

While we're at it, even finding best chordal fill-in is hard

Theorem 5.4.2

Given an arbitrary graph $G = (V, E)$, and $G' = (V, E \cup F)$ is a triangulation of G, finding the smallest such F is an NP-complete optimization problem.

Thus, to summarize, finding the optimal elimination order is likely computationally hard, as are other problems associated with graphs.

Some good news ☺- at least we can identify triangulated graphs

- We know that if there is a perfect elim order, the graph is triangulated.

Some good news ☺- at least we can identify triangulated graphs

- We know that if there is a perfect elim order, the graph is triangulated.
- keep eliminating simplicial nodes as long as you can, and output "not triangulated" if ever there is no simplicial node.

Some good news ☺- at least we can identify triangulated graphs

- We know that if there is a perfect elim order, the graph is triangulated.
- keep eliminating simplicial nodes as long as you can, and output "not triangulated" if ever there is no simplicial node.
- naïve implementation: find fill in of each node, eliminate the one with no fill-in $O(n^3)$.

Some good news ☺- at least we can identify triangulated graphs

- We know that if there is a perfect elim order, the graph is triangulated.
- keep eliminating simplicial nodes as long as you can, and output "not triangulated" if ever there is no simplicial node.
- naïve implementation: find fill in of each node, eliminate the one with no fill-in $O(n^3)$.
- There is a smart algorithm, maximum cardinality search (MCS), that can do this in $O(|V| + |E|)$

Some good news ☺- at least we can identify triangulated graphs

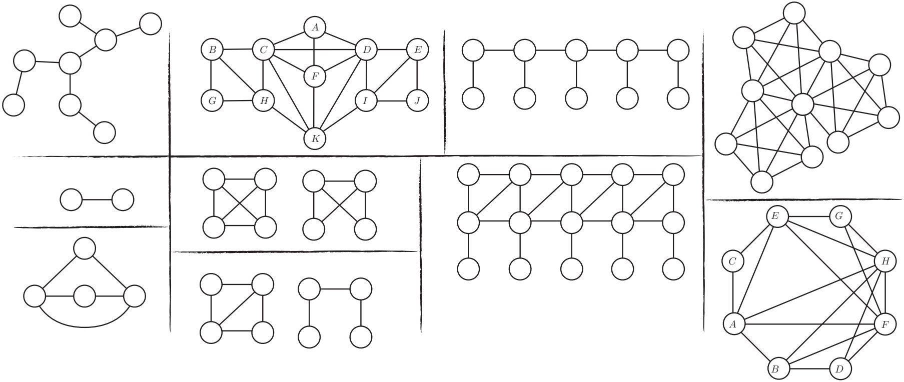
- We know that if there is a perfect elim order, the graph is triangulated.
- keep eliminating simplicial nodes as long as you can, and output "not triangulated" if ever there is no simplicial node.
- naïve implementation: find fill in of each node, eliminate the one with no fill-in $O(n^3)$.
- There is a smart algorithm, maximum cardinality search (MCS), that can do this in $O(|V| + |E|)$
- Basic idea of MCS: produce a perfect elimination order, if it exists, in reverse. Construct it by looking at previously labeled neighbors.

Maximum Cardinality Search (MCS)

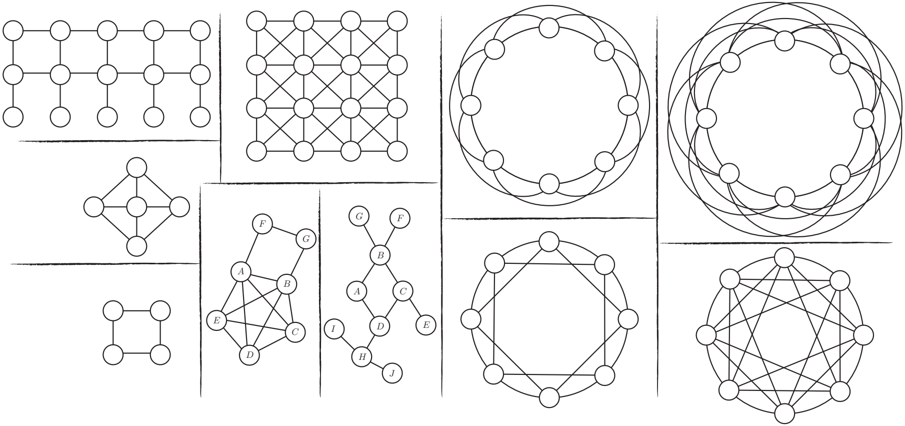
Input: An undirected graph $G = (V, E)$ with $n = |V|$.
Result: triangulated or not, MCS ordering $\sigma = (v_1, \ldots, v_n)$

$L \leftarrow \emptyset$; $i \leftarrow 1$;
while $|V \setminus L| > 0$ **do**
  Choose $v_i \in \mathrm{argmax}_{u \in V \setminus L} |\delta(u) \cap L|$; /* v_i's previously labeled neighbors has max cardinality. */
  $c_i \leftarrow \delta(v_i) \cap L$; /* c_i is v_i's neighbors in the reverse elimination order. */
  if $\{v_i\} \cup c_i$ *is not complete in* G **then**
    return *"not triangulated"* ;
  $L \leftarrow L \cup \{v_i\}$; $i \leftarrow i + 1$;
return *"triangulated", and the node ordering* σ

Ex: Run MCS on one of these graphs



Ex: Run MCS on one of these graphs



MCS

- Can also produce an elimination order and triangulate the graphs (but not particularly good)
- will produce a perfect elimination order on triangulated graphs
- why called maximum cardinality "search"

Theorem 5.4.3

A graphical G is triangulated iff in the MCS algorithm, at each point when a vertex is marked, that vertex's previously marked neighbors form a complete subgraph of G.

Corollary 5.4.4

Every maximum cardinality search of a triangulated graph G corresponds to a reverse perfect eliminating order of G.

Recap

- Triangulated graphs: if $|V| \geq 2$, always two simplicial nodes.

Recap

- Triangulated graphs: if $|V| \geq 2$, always two simplicial nodes.
- Triangulated graph iff perfect elimination graph.

Recap

- Triangulated graphs: if $|V| \geq 2$, always two simplicial nodes.
- Triangulated graph iff perfect elimination graph.
- All minimal triangulations of a graph can be created using elimination.

Recap

- Triangulated graphs: if $|V| \geq 2$, always two simplicial nodes.
- Triangulated graph iff perfect elimination graph.
- All minimal triangulations of a graph can be created using elimination.
- k-trees, generalization of trees. Sometimes called hyper-tree. All min-separators are k-cliques. partial k-trees. Embedding into k-trees.

Recap

- Triangulated graphs: if $|V| \geq 2$, always two simplicial nodes.
- Triangulated graph iff perfect elimination graph.
- All minimal triangulations of a graph can be created using elimination.
- k-trees, generalization of trees. Sometimes called hyper-tree. All min-separators are k-cliques. partial k-trees. Embedding into k-trees.
- Any triangulated graph G' can be embedded into k-tree where $k+1$ is the size of the largest clique of G'. Thus any graph can be embedded into a k-tree for large enough k.

Recap

- Triangulated graphs: if $|V| \geq 2$, always two simplicial nodes.
- Triangulated graph iff perfect elimination graph.
- All minimal triangulations of a graph can be created using elimination.
- k-trees, generalization of trees. Sometimes called hyper-tree. All min-separators are k-cliques. partial k-trees. Embedding into k-trees.
- Any triangulated graph G' can be embedded into k-tree where $k+1$ is the size of the largest clique of G'. Thus any graph can be embedded into a k-tree for large enough k.
- NP-complete: finding smallest k such that G is embeddable into k-tree.

Recap

- Triangulated graphs: if $|V| \geq 2$, always two simplicial nodes.
- Triangulated graph iff perfect elimination graph.
- All minimal triangulations of a graph can be created using elimination.
- k-trees, generalization of trees. Sometimes called hyper-tree. All min-separators are k-cliques. partial k-trees. Embedding into k-trees.
- Any triangulated graph G' can be embedded into k-tree where $k+1$ is the size of the largest clique of G'. Thus any graph can be embedded into a k-tree for large enough k.
- NP-complete: finding smallest k such that G is embeddable into k-tree.
- Triangulation heuristics: min-fill, etc.

Recap

- Triangulated graphs: if $|V| \geq 2$, always two simplicial nodes.
- Triangulated graph iff perfect elimination graph.
- All minimal triangulations of a graph can be created using elimination.
- k-trees, generalization of trees. Sometimes called hyper-tree. All min-separators are k-cliques. partial k-trees. Embedding into k-trees.
- Any triangulated graph G' can be embedded into k-tree where $k+1$ is the size of the largest clique of G'. Thus any graph can be embedded into a k-tree for large enough k.
- NP-complete: finding smallest k such that G is embeddable into k-tree.
- Triangulation heuristics: min-fill, etc.
- MCS can identify a triangulated graph efficiently.

Multiple queries

- Let $\mathcal{C}$ be the set of all cliques in original graph. Often, we want to compute $p(x_C)$ for all $C \in \mathcal{C}$.

Multiple queries

- Let $\mathcal{C}$ be the set of all cliques in original graph. Often, we want to compute $p(x_C)$ for all $C \in \mathcal{C}$.
- Do not want to run separate elimination $|\mathcal{C}|$ many times.

Multiple queries

- Let $\mathcal{C}$ be the set of all cliques in original graph. Often, we want to compute $p(x_C)$ for all $C \in \mathcal{C}$.
- Do not want to run separate elimination $|\mathcal{C}|$ many times.
- Recall tree (i.e., 1-tree) case - messages for one query used for other queries. Message re-use/efficiency only grows with num. queries.

Multiple queries

- Let $\mathcal{C}$ be the set of all cliques in original graph. Often, we want to compute $p(x_C)$ for all $C \in \mathcal{C}$.
- Do not want to run separate elimination $|\mathcal{C}|$ many times.
- Recall tree (i.e., 1-tree) case - messages for one query used for other queries. Message re-use/efficiency only grows with num. queries. Can we do the same thing for arbitrary graphs?

Multiple queries

- Let $\mathcal{C}$ be the set of all cliques in original graph. Often, we want to compute $p(x_C)$ for all $C \in \mathcal{C}$.
- Do not want to run separate elimination $|\mathcal{C}|$ many times.
- Recall tree (i.e., 1-tree) case - messages for one query used for other queries. Message re-use/efficiency only grows with num. queries. Can we do the same thing for arbitrary graphs?
- Consider only the class of triangulated models since to do otherwise (for exact inference) is not necessary.

Multiple queries

- Let $\mathcal{C}$ be the set of all cliques in original graph. Often, we want to compute $p(x_C)$ for all $C \in \mathcal{C}$.
- Do not want to run separate elimination $|\mathcal{C}|$ many times.
- Recall tree (i.e., 1-tree) case - messages for one query used for other queries. Message re-use/efficiency only grows with num. queries. Can we do the same thing for arbitrary graphs?
- Consider only the class of triangulated models since to do otherwise (for exact inference) is not necessary.
- But is one triangulated model optimal for all queries?

Multiple queries

- A triangulated graph is a cover of G

Multiple queries

- A triangulated graph is a cover of G
- Any clique in G will still be a clique in a triangulation G': that is, given clique $c \in \mathcal{C}(G)$, there exists $c' \in \mathcal{C}(G')$ with $c \subseteq c'$.

Multiple queries

- A triangulated graph is a cover of G
- Any clique in G will still be a clique in a triangulation G': that is, given clique $c \in \mathcal{C}(G)$, there exists $c' \in \mathcal{C}(G')$ with $c \subseteq c'$.
- Given $p(x_{c'})$, can compute $p(x_c) = \sum_{x_{c' \setminus c}} p(x_{c'})$ at $O(r^{|c'|})$, same cost triangulated graph.

Multiple queries

- A triangulated graph is a cover of G
- Any clique in G will still be a clique in a triangulation G': that is, given clique $c \in \mathcal{C}(G)$, there exists $c' \in \mathcal{C}(G')$ with $c \subseteq c'$.
- Given $p(x_{c'})$, can compute $p(x_c) = \sum_{x_{c' \setminus c}} p(x_{c'})$ at $O(r^{|c'|})$, same cost triangulated graph.
- optimal k-tree embedding for G is one that minimizes the maximum clique for any triangulation of G, so if we have found this embedding, this will be optimal for any original-graph clique marginal.

Multiple queries

- A triangulated graph is a cover of G
- Any clique in G will still be a clique in a triangulation G': that is, given clique $c \in \mathcal{C}(G)$, there exists $c' \in \mathcal{C}(G')$ with $c \subseteq c'$.
- Given $p(x_{c'})$, can compute $p(x_c) = \sum_{x_{c' \setminus c}} p(x_{c'})$ at $O(r^{|c'|})$, same cost triangulated graph.
- optimal k-tree embedding for G is one that minimizes the maximum clique for any triangulation of G, so if we have found this embedding, this will be optimal for any original-graph clique marginal.
- Even if we found a "good" elimination order (one that produces a maxclique of reasonable size), this order can be shared for other clique queries.

Non-clique queries

- Recall: 1-tree case, if we want a marginal over a non-sub-tree, we might be in trouble.

Non-clique queries

- Recall: 1-tree case, if we want a marginal over a non-sub-tree, we might be in trouble.
- Similarly, if we desire non-clique queries for general graph, then computation can get worse. Computing $p(x_L)$ for arbitrary L could turn x_L into a clique in the worst case (Rose's theorem).

Non-clique queries

- Recall: 1-tree case, if we want a marginal over a non-sub-tree, we might be in trouble.
- Similarly, if we desire non-clique queries for general graph, then computation can get worse. Computing $p(x_L)$ for arbitrary L could turn x_L into a clique in the worst case (Rose's theorem).
- If x_L is not clique in G', then we can view G' as not being "valid" for the query $p(x_L)$.

Non-clique queries

- Recall: 1-tree case, if we want a marginal over a non-sub-tree, we might be in trouble.
- Similarly, if we desire non-clique queries for general graph, then computation can get worse. Computing $p(x_L)$ for arbitrary L could turn x_L into a clique in the worst case (Rose's theorem).
- If x_L is not clique in G', then we can view G' as not being "valid" for the query $p(x_L)$.
- In such case, need to re-triangulate, starting with a graph where x_L is made complete.

Computing all clique queries efficiently via elimination

- Remarkably, in the case of clique queries, we can actually re-use the elimination order.

Computing all clique queries efficiently via elimination

- Remarkably, in the case of clique queries, we can actually re-use the elimination order.
- We want to share more than just the elimination order.

Computing all clique queries efficiently via elimination

- Remarkably, in the case of clique queries, we can actually re-use the elimination order.
- We want to share more than just the elimination order.
- goal: in non-tree graphs, re-use work of computing marginals for the sake of getting multiple marginals.

Computing all clique queries efficiently via elimination

- Remarkably, in the case of clique queries, we can actually re-use the elimination order.
- We want to share more than just the elimination order.
- goal: in non-tree graphs, re-use work of computing marginals for the sake of getting multiple marginals.
- We'll see an amazing fact: if we find the optimal elimination order for 1 clique query, it is optimal for all clique queries!! ☺

Decomposition of G

Definition 5.5.1 (Decomposition of G)

A *decomposition* of a graph $G = (V, E)$ (if it exists) is a partition (A, B, C) of V such that:

- C separates A from B in G.
- C is a clique.

if A and B are both non-empty, then the decomposition is called *proper*.

If G has a decomposition, what dies this mean for the family $\mathcal{F}(G, \mathcal{M}^{(\mathrm{f})})$? Since C separates A from B, this means that $X_A \perp\!\!\!\perp X_B | X_C$ for any $p \in \mathcal{F}(G, \mathcal{M}^{(\mathrm{f})})$, which moreover means we can write the joint distribution in a particular form.

$$p(x) = p(x_A, x_B, x_C) = \frac{p(x_A, x_C)p(x_B, x_C)}{p(x_C)} \tag{5.1}$$

Decomposable models

Definition 5.5.2

A graph $G = (V, E)$ is decomposable if either: 1) G is a clique, or 2) G possesses a proper decomposition (A, B, C) s.t. both subgraphs $G[A \cup C]$ and $G[B \cup C]$ are decomposable.

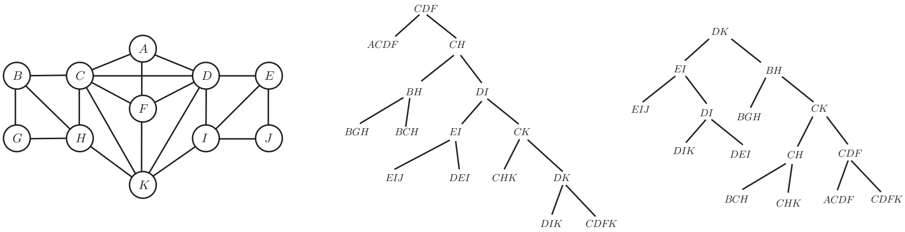
Decomposable models

Definition 5.5.2

A graph $G = (V, E)$ is decomposable if either: 1) G is a clique, or 2) G possesses a proper decomposition (A, B, C) s.t. both subgraphs $G[A \cup C]$ and $G[B \cup C]$ are decomposable.

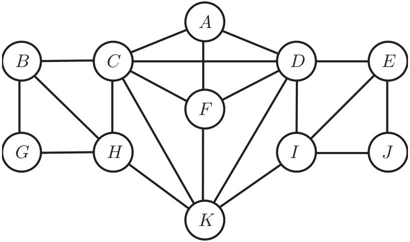
- Note that the separator is contained within the subgraphs: i.e., $G[A \cup C]$ rather than, say, $G[A]$.

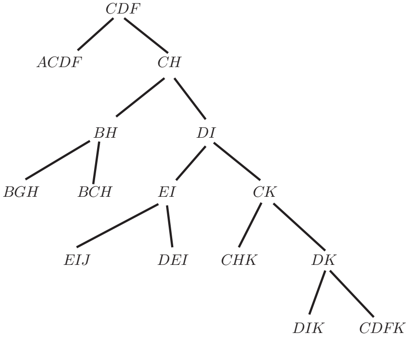
Decomposable models



- Graph and two decompositions of this graph.
- as we recurse down, if at any point decomposition is not found, graph is not decomposable.

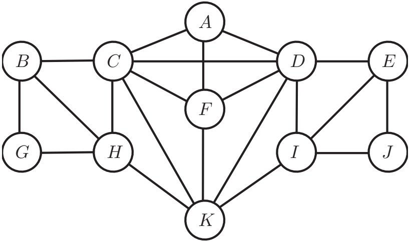
Decomposable models

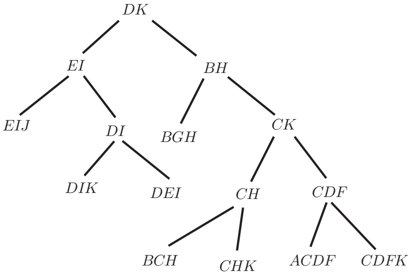




- Graph and two decompositions of this graph.
- as we recurse down, if at any point decomposition is not found, graph is not decomposable.

Decomposable models





- Graph and two decompositions of this graph.
- as we recurse down, if at any point decomposition is not found, graph is not decomposable.

Decomposition of G and Decomposable graphs

Summarizing both:

Definition 5.5.3 (Decomposition of G)

A *decomposition* of a graph $G = (V, E)$ (if it exists) is a partition (A, B, C) of V such that:

- C separates A from B in G.
- C is a clique.

if A and B are both non-empty, then the decomposition is called *proper*.

Definition 5.5.4

A graph $G = (V, E)$ is decomposable if either: 1) G is a clique, or 2) G possesses a **proper** decomposition (A, B, C) s.t. both subgraphs $G[A \cup C]$ and $G[B \cup C]$ are decomposable.

Note part 2. It says *possesses*. Bottom of tree might affect top.

Decomposable models

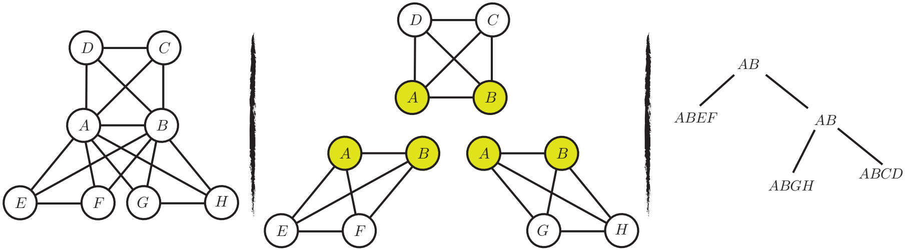
- Internal nodes in tree are complete graphs that are also separators.
- With G is decomposable, what are implications for a $p \in \mathcal{F}(G, \mathcal{M}^{(\mathrm{f})})$?

$$
\begin{aligned}
& p(A,B,C,D,E,F,G,H,I,J,K) \\
& = \frac{p(A,C,D,F)p(B,C,D,E,F,G,H,I,J,K)}{p(C,D,F)} \\
& = \frac{p(A,C,D,F)}{p(C,D,F)} \left(\frac{p(B,C,G,H)p(C,D,E,F,H,I,J,K)}{p(C,H)} \right) \\
& = \dots
\end{aligned}
$$

$$
= \frac{p(A,C,D,F)p(B,G,H)p(C,B,H)p(I,E,J)p(E,I,D)p(C,K,H)p(D,K,I)p(D,K,F,C)}{p(C,D,F)p(C,H)p(B,H)p(D,I)p(E,I)p(C,K)p(D,K)}
$$

Decomposable models

- S is a separator, so that $G[V \setminus S]$ consists of 2 or more **connected components**.
- We say that S *shatters* the graph G into those components, and let $d(S)$ be the number of connected components that S shatters G into. $d(S)$ is the shattering coefficient of G.
- Example: below, $d(\{A, B\}) = 3$



Decomposable models

- When $d(S) > 2$, separator marginal use more than once in the denominator
- The general form of the factorization becomes:

$$p(x) = \frac{\prod_{C \in \mathcal{C}(G)} p(x_C)}{\prod_{S \in \mathcal{S}(G)} p(x_S)^{d(S)-1}} \tag{5.2}$$

- Any decomposable model can be written this way
- 4-cycle is not decomposable. Two independence properties that can't be used simultaneously.

$$p(x_1, x_2, x_3, x_4) = \frac{p(x_1, x_2, x_4)p(x_1, x_3, x_4)}{p(x_1, x_4)} = \frac{p(x_1, x_2, x_3)p(x_2, x_3, x_4)}{p(x_2, x_3)} \tag{5.3}$$

Decomposable models

Proposition 5.5.5

All of the maxcliques in a graph lie on the leaf nodes of the binary decomposition tree

Proof.

For a decomposable model, the base case (leaf node) is a clique, otherwise it would not be decomposable. If a leaf was not a maxclique, then that means it is contained in a maxclique, and got split by a separator corresponding to that leaf's parent, but this is impossible since a maxcliques have no separator. □

Proposition 5.5.6

The (nec. unique) set of all minimal separators of graph are included in the non-leaf nodes of the binary decomposition tree, with $d(S) - 1$ being the number of times the minimal separator S appears as a given non-leaf node.

A bit of notation

- If C is separator, C shatters G into $d(C)$ connected components
- $G[V \setminus C]$ is the union of these components (not including C)
- Let $\{G_1, G_2, \dots, G_\ell\}$ be (disjoint) connected components of $G[V \setminus C]$, so $G_1 \cup G_2 \cup \dots \cup G_\ell = G[V \setminus C]$
- Given $a \in V(G_i)$ for some i, then $G[V \setminus C](a) = G_i$.

Triangulated vs. decomposable

Theorem 5.5.7

A given graph $G = (V, E)$ is triangulated iff it is decomposable.

Proof.

First, recall from Lemma 4.5.6 that a graph is triangulated iff it is decomposable. To prove the current theorem, we will first show (by induction) that decomposability implies that the graph is triangulated). Next, for the converse, we'll show that every minimal separator complete in G implies decomposable.

Triangulated vs. decomposable

Proof of Theorem 5.5.7.

First, assume G is decomposable. If G is complete then it is triangulated. If it is not complete then there exists a proper decomposition (A, B, C) into decomposable subgraphs $G[A \cup C]$ and $G[B \cup C]$ both of which have fewer vertices, meaning $|A \cup C| < |V|$ and $|B \cup C| < |V|$. By the induction hypothesis, both $G[A \cup C]$ and $G[B \cup C]$ are chordal. Any potential chordless cycle, therefore, can't be contained in one of the sub-components, so if it exist in G must intersect both A and B. Since C separates A from B, the purported chordless cycle would intersect C twice, but C is complete the cycle has a chord. The first part of the theorem is proven.

Triangulated vs. decomposable

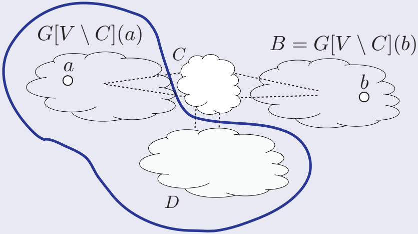
. . . proof of Theorem 5.5.7 cont.

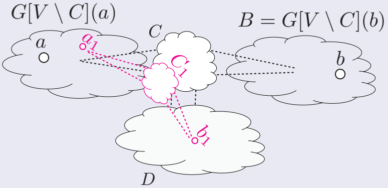
Next (to show the converse), assume that all minimum (a, b) separators are complete in G. If G is complete then it is decomposable. Otherwise, there exists two non-adjacent vertices $a, b \in V$ in G with a necessarily complete minimal separator C forming a partition $G[V \setminus C](a)$, $G[V \setminus C](b)$, and all of the remaining components of $G[V \setminus C]$. We merge the connected components together to form only two components as follows: let $A = G[V \setminus C](a) \cup D$ and $B = G[V \setminus C](b)$. Since C is complete, we see that (A, B, C) form a decomposition of G, but we still need that $G[A \cup C]$ and $G[B \cup C]$ to be decomposable (see figure).

Triangulated vs. decomposable

. . . proof of Theorem 5.5.7 cont.

$A = G[V \setminus C](a) \cup D$





Triangulated vs. decomposable

. . . proof of Theorem 5.5.7 cont.

Let C_1 be a minimal (a_1, b_1) separator in $G[A \cup C]$. But then C_1 is also a minimal (a_1, b_1) separator in G since, once we add B back to $G[A \cup C]$ to regenerate G, there still cannot be any new paths from a_1 to b_1 circumventing C_1. This is because any such path would involve nodes in B (the only new nodes) which, to reach B and return, requires going through C (which is complete) twice. Such a path cannot bypass C_1 since if it did, a shorter path not involving B would bypass C_1. Therefore, C_1 is complete in G, and an inductive argument says that $G[A \cup C]$ is decomposable. The same argument holds for $G[B \cup C]$. Therefore, G is decomposable. □

Tree decomposition

Definition 5.5.8 (tree decomposition)

Given a graph $G = (V, E)$, a tree-decomposition of a graph is a pair $(\{C_i : i \in I\}, T)$ where $T = (I, F)$ is a tree with node index set I, edge set F, and $\{C_i\}_i$ (one for each $i \in I$) is a collection of subsets of $V(G)$ such that:

1. $\cup_{i \in I} C_i = V$
2. for any $(u, v) \in E(G)$, there exists $i \in I$ with $u, v \in C_i$
3. for any $v \in V$, the set $\{i \in I : v \in C_i\}$ forms a connected subtree of T

Tree decomposition is also hard

- The tree-width of the tree-decomposition is the size of the largest C_i minus one (i.e., $\max_{i \in I} |C_i| - 1$.

Tree decomposition is also hard

- The tree-width of the tree-decomposition is the size of the largest C_i minus one (i.e., $\max_{i \in I} |C_i| - 1$.

Theorem 5.5.9

Given graph $G = (V, E)$, finding the tree decomposition $T = (I, F)$ of G that minimizes the tree width ($\max_{i \in I} |C_i| - 1$) is an NP-complete optimization problem.

Tree decomposition is also hard

- The tree-width of the tree-decomposition is the size of the largest C_i minus one (i.e., $\max_{i \in I} |C_i| - 1$.

Theorem 5.5.9

Given graph $G = (V, E)$, finding the tree decomposition $T = (I, F)$ of G that minimizes the tree width ($\max_{i \in I} |C_i| - 1$) is an NP-complete optimization problem.

- Multiplicatively approximable within $O(\log |V|)$, but not possible to additively do better than $|V|^{1-\epsilon}$ for any $\epsilon > 0$.

Tree decomposition is also hard

- The tree-width of the tree-decomposition is the size of the largest C_i minus one (i.e., $\max_{i \in I} |C_i| - 1$.

Theorem 5.5.9

Given graph $G = (V, E)$, finding the tree decomposition $T = (I, F)$ of G that minimizes the tree width $(\max_{i \in I} |C_i| - 1)$ is an NP-complete optimization problem.

- Multiplicatively approximable within $O(\log |V|)$, but not possible to additively do better than $|V|^{1-\epsilon}$ for any $\epsilon > 0$.
- How does this relate to our problem though?

$\longrightarrow$ trees

- All roads lead to trees, namely junction trees.
- Next set of slides will make the transformation mathematically precise.

Cluster graphs

Definition 5.6.1 (Cluster graph)

Consider forming a new graph based on G where the new graph has nodes that correspond to clusters in the original G, and has edges existing between two (cluster) nodes only when the corresponding clusters have a non-zero intersection. That is, let $\mathcal{C}(G) = \left\{C_1, C_2, \ldots, C_{|I|}\right\} =$ be a set of $|I|$ clusters of nodes $V(G)$, where $C_i \subseteq V(G), i \in I$. Consider a new graph $\mathcal{J} = (I, \mathcal{E})$ where each node in $\mathcal{J}$ corresponds to a set of nodes in G, and where edge $(i, j) \in \mathcal{E}$ if $C_i \cap C_j \neq \emptyset$. We will also use $S_{ij} = C_i \cap C_j$ as notation.

Cluster graphs

Definition 5.6.1 (Cluster graph)

Consider forming a new graph based on G where the new graph has nodes that correspond to clusters in the original G, and has edges existing between two (cluster) nodes only when the corresponding clusters have a non-zero intersection. That is, let $\mathcal{C}(G) = \left\{C_1, C_2, \dots, C_{|I|}\right\} =$ be a set of $|I|$ clusters of nodes $V(G)$, where $C_i \subseteq V(G), i \in I$. Consider a new graph $\mathcal{J} = (I, \mathcal{E})$ where each node in $\mathcal{J}$ corresponds to a set of nodes in G, and where edge $(i, j) \in \mathcal{E}$ if $C_i \cap C_j \neq \emptyset$. We will also use $S_{ij} = C_i \cap C_j$ as notation.

So two cluster nodes have an edge between them iff there is non-zero intersection between the nodes.

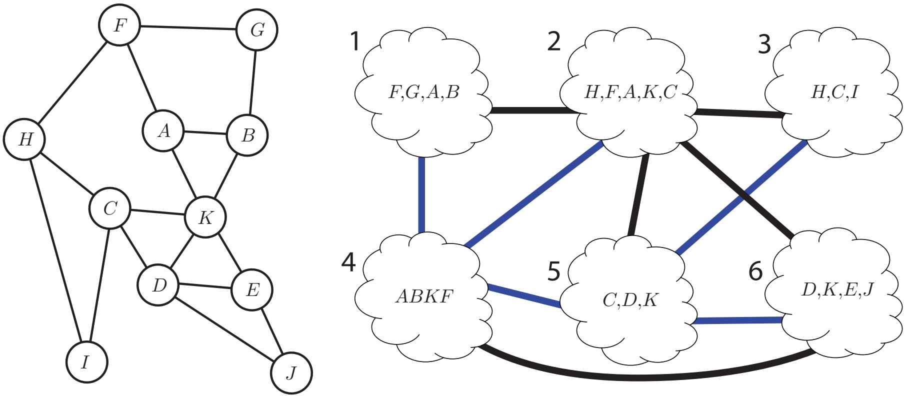
Cluster Trees

If the graph is a tree, then we have what is called a cluster tree.

Definition 5.6.2 (Cluster Tree)

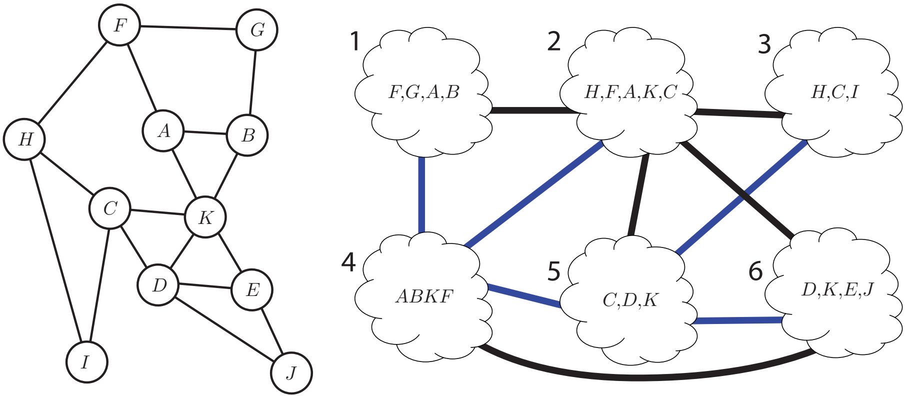
Let $\mathcal{C} = \{C_1, C_2, \ldots, C_{|I|}\}$ be a set of node clusters of graph $G = (V, E)$. A cluster tree is a tree $\mathcal{T} = (I, \mathcal{E}_T)$ with vertices corresponding to clusters in $\mathcal{C}$ and edges corresponding to pairs of clusters $C_1, C_2 \in \mathcal{C}$. We can label each vertex in $i \in I$ by the set of graph nodes in the corresponding cluster in G, and we label each edge $(i, j) \in \mathcal{E}_T$ by the cluster intersection, i.e., $S_{ij} = C_i \cap C_j$.

Cluster Graphs/Trees



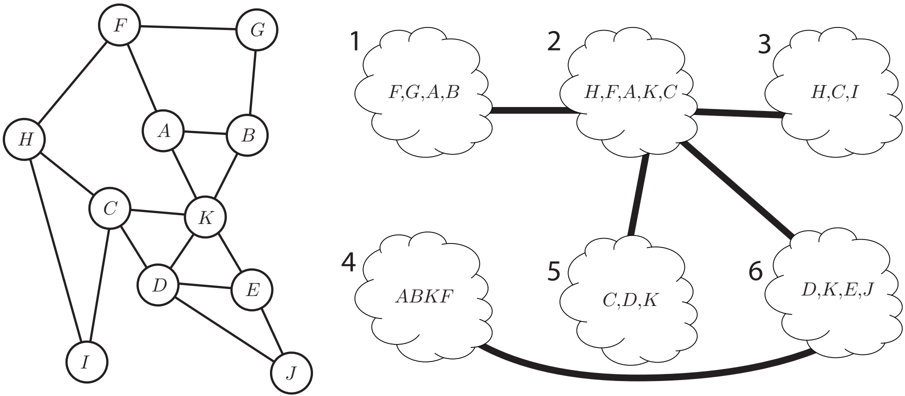
Left: a graph. Right: A cluster graph with $|I| = 6$ clusters, where $C_1 = \{F, G, A, B\}, C_2 = \{H, F, A, K, C\}, \ldots$. There is an edge $(1, 2)$ since $C_1 \cap C_2 = \{F, A\} \neq \emptyset$. If we remove all but the blue edges, then we get a cluster tree.

Cluster Graphs/Trees



Left: a graph. Right: A cluster graph with $|I| = 6$ clusters, where $C_1 = \{F, G, A, B\}, C_2 = \{H, F, A, K, C\}, \ldots$. There is an edge $(1, 2)$ since $C_1 \cap C_2 = \{F, A\} \neq \emptyset$. If we remove all but the blue edges, then we get a cluster tree.

Cluster Graphs/Trees



Left: a graph. Right: A cluster graph with $|I| = 6$ clusters, where $C_1 = \{F, G, A, B\}, C_2 = \{H, F, A, K, C\}, \ldots$. There is an edge $(1, 2)$ since $C_1 \cap C_2 = \{F, A\} \neq \emptyset$.

Cluster Intersection Property (c.i.p.)

- Important: Cluster graphs and cluster trees are based only on a set of clusters of nodes of $G = (V, E)$. We haven't, based on these definitions, yet used any of the o.g. edges of G.
- Edges in a cluster graph and cluster tree are not o.g. edges. Instead, they are based on if two clusters have non-empty intersection.
- We want to talk about cluster trees that have certain properties. A cluster graph might or might not have such properties.

Cluster Intersection Property (c.i.p.)

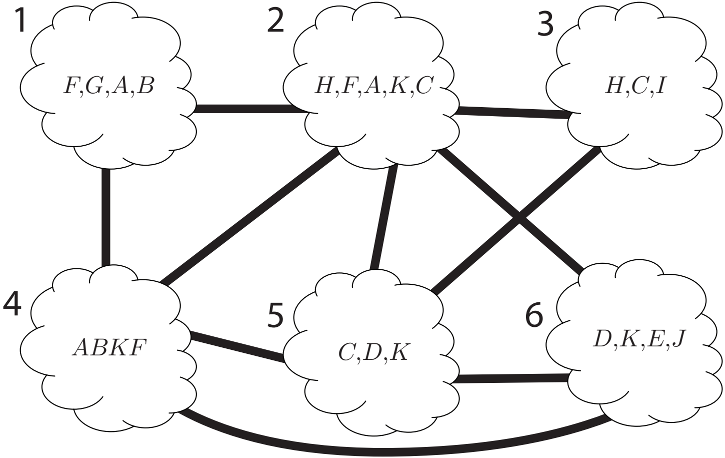
Definition 5.6.3 (Cluster Intersection Property)

We are given a cluster tree $\mathcal{T} = (I, \mathcal{E}_T)$, and let C_1, C_2 be any two clusters in the tree. Then the cluster intersection property states that $C_1 \cap C_2 \subseteq C_i$ for all C_i on the (by definition, necessarily) unique path between C_1 and C_2 in the tree $\mathcal{T}$.

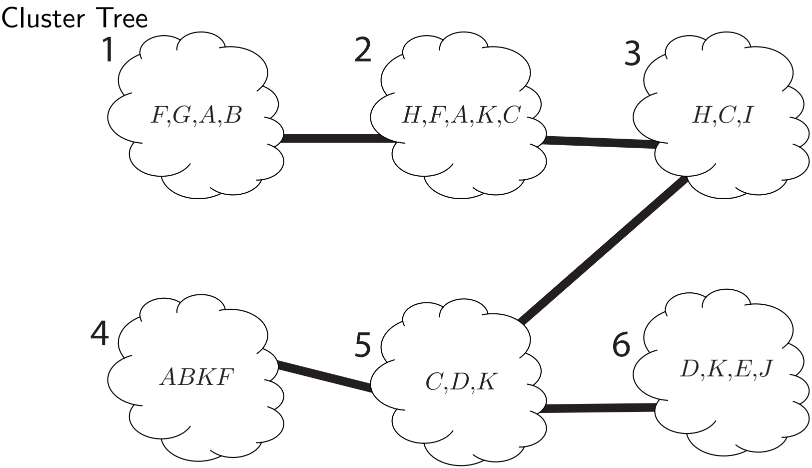
- A given cluster tree might or might not have that property.
- Example on the next few slides.

Examples

Cluster Graph

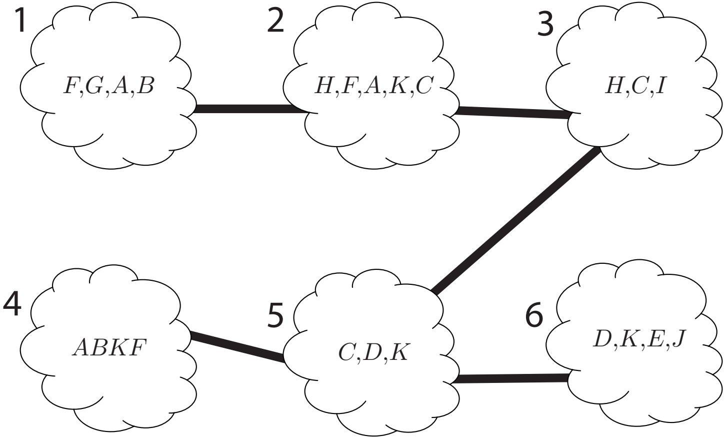


Examples



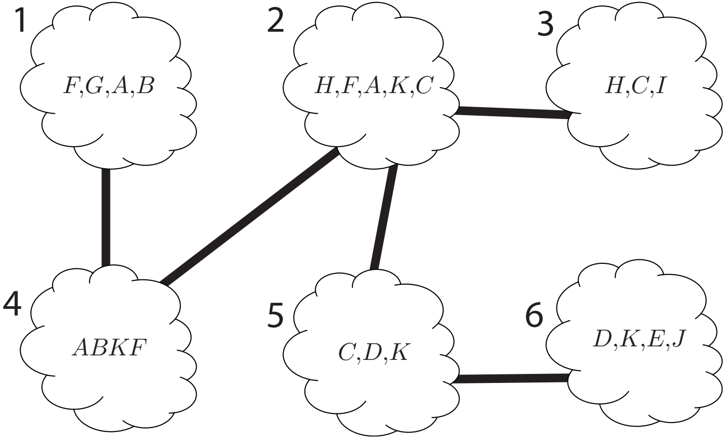
Examples

Cluster Tree that violates the cluster intersection property



Examples

Cluster Tree that obeys the cluster intersection property



Running Intersection Property (r.i.p.)

Definition 5.6.4 (Running Intersection Property (r.i.p.))

Let $C_1, C_2, \ldots, C_\ell$ be an ordered sequence of subsets of $V(G)$. Then the ordering obeys the running intersection property (r.i.p.) property if for all $i > 1$, there exists $j < i$ such that $C_i \cap (\cup_{k<i} C_k) = C_i \cap C_j$.

- r.i.p. is defined in terms of clusters of nodes in a graph. r.i.p. holds if such an ordering can be found.

Running Intersection Property (r.i.p.)

Definition 5.6.4 (Running Intersection Property (r.i.p.))

Let $C_1, C_2, \ldots, C_\ell$ be an ordered sequence of subsets of $V(G)$. Then the ordering obeys the running intersection property (r.i.p.) property if for all $i > 1$, there exists $j < i$ such that $C_i \cap (\cup_{k<i} C_k) = C_i \cap C_j$.

- r.i.p. is defined in terms of clusters of nodes in a graph. r.i.p. holds if such an ordering can be found.
- Cluster j acts as a representative for all of i's history.

Running Intersection Property (r.i.p.)

Given sequence of clusters $C_1, C_2, \ldots, C_\ell$. Define the history (accumulation) of sequence at position i:

Running Intersection Property (r.i.p.)

Given sequence of clusters $C_1, C_2, \ldots, C_\ell$. Define the history (accumulation) of sequence at position i:

$$H_i = C_1 \cup C_2 \cup \cdots \cup C_i. \tag{5.4}$$

Running Intersection Property (r.i.p.)

Given sequence of clusters $C_1, C_2, \ldots, C_\ell$. Define the history (accumulation) of sequence at position i:

$$H_i = C_1 \cup C_2 \cup \cdots \cup C_i. \tag{5.4}$$

Innovation (residual) or new nodes in C_i not encountered in the previous history, as:

$$R_i = C_i \setminus H_{i-1}. \tag{5.5}$$

Running Intersection Property (r.i.p.)

Given sequence of clusters $C_1, C_2, \ldots, C_\ell$. Define the history (accumulation) of sequence at position i:

$$H_i = C_1 \cup C_2 \cup \cdots \cup C_i. \tag{5.4}$$

Innovation (residual) or new nodes in C_i not encountered in the previous history, as:

$$R_i = C_i \setminus H_{i-1}. \tag{5.5}$$

Lastly, define the non-innovation, commonality, or separation elements between new and previous history:

$$S_i = C_i \cap H_{i-1} \tag{5.6}$$

Running Intersection Property (r.i.p.)

Given sequence of clusters $C_1, C_2, \ldots, C_\ell$. Define the history (accumulation) of sequence at position i:

$$H_i = C_1 \cup C_2 \cup \cdots \cup C_i. \tag{5.4}$$

Innovation (residual) or new nodes in C_i not encountered in the previous history, as:

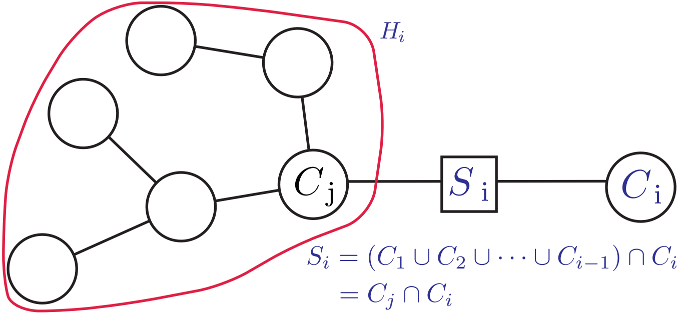
$$R_i = C_i \setminus H_{i-1}. \tag{5.5}$$

Lastly, define the non-innovation, commonality, or separation elements between new and previous history:

$$S_i = C_i \cap H_{i-1} \tag{5.6}$$

Note $C_i = R_i \cup S_i$, i^{th} clusters consists of the innovation R_i and the commonality S_i.

Running Intersection Property (r.i.p.)



Clusters are in r.i.p. order if the commonality S_i between new and history is fully contained in one element of history. I.e., there exists an $j < i$ such that $S_i \subseteq C_j$.

First Two Properties

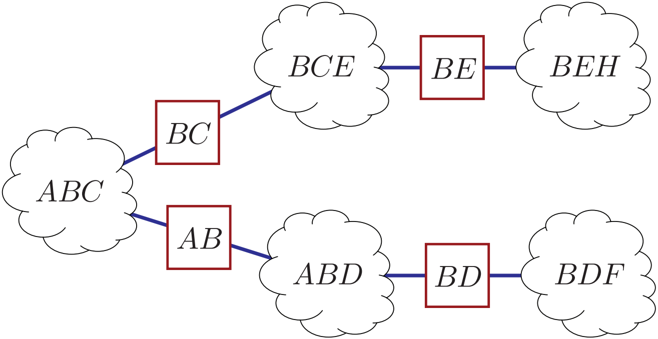
Lemma 5.6.5

The cluster intersection and running intersection properties are identical.

Proof.

Starting with clusters in r.i.p. order, construct cluster tree by connecting each i to its corresponding j node. This is a tree. Also, take any C_i, C_k with $i > k$. S_i summarizes everything between C_i and H_{i-1} so $C_i \cap C_k \subseteq S_i$. Apply recursively on unique path between C_i and C_j. Conversely, perform traversal (depth or breadth first search) on cluster tree. That order will satisfy r.i.p. since any possible intersection between C_i, C_j on unique path, it must be fully contained in neighbor. $\square$

First Two Properties



Example of a set of node clusters (within the cloud-like shapes) arranged in a tree that satisfies the r.i.p. and also the cluster intersection property. The intersections between neighboring node clusters are shown in the figure as square boxes. Consider the path or $\{B, E, H\} \cap \{B, D, F\} = \{B\}$.

Induced sub-tree property (i.s.p.)

Definition 5.6.6 (Induced Sub-tree Property)

Given a cluster tree $\mathcal{T}$ for graph G, the *induced sub-tree property* holds for $\mathcal{T}$ if for all $v \in V$, the set of clusters $C \in \mathcal{C}$ such that $v \in C$ induces a sub-tree $\mathcal{T}(v)$ of $\mathcal{T}$.

Note, by definition the sub-tree is necessarily connected.

Three properties

Lemma 5.6.7

Induced sub-tree property holds iff cluster intersection property holds

Proof.

Assume induced subtree holds. Take all $v \in C_i \cap C_j$, then each such v induces a sub-tree of $\mathcal{T}$, and all of these sub-trees overlap on the unique path between C_i and C_j in $\mathcal{T}$.
Conversely, when cluster intersection property holds, given $v \in V$, consider all clusters that contain v, $\mathcal{C}(v) = \{C \in \mathcal{C} : v \in C\}$. For any pair $C_1, C_2 \in \mathcal{C}(v)$, we have that $C_1 \cap C_2$ exists on the unique path between C_1 and C_2 in $\mathcal{T}$, and since $v \in C_1 \cap C_2$, v always exists on each of these paths. These paths, considered as a union together, cannot form a cycle (since they are paths on a tree). Moreover, these paths unioned together form a tree (they're connected). □

Therefore, cluster intersection property, running intersection property, and induced sub-tree property, are all identical. We'll henceforth refer them collectively as r.i.p.

Tree decomposition

Lets look again at tree decomposition, a cluster tree that satisfies (what we now know to be the) induced sub-tree property (e.g., r.i.p. and c.i.p. as well).

Definition 5.6.8 (tree decomposition)

Given a graph $G = (V, E)$, a tree-decomposition of a graph is a pair $(\{C_i : i \in I\}, T)$ where $\mathcal{T} = (I, \mathcal{E}_T)$ is a tree with node index set I, edge set $\mathcal{E}_T$, and $\{C_i\}_i$ (one for each $i \in I$) is a collection of clusters (subsets) of $V(G)$ such that:

1. $\cup_{i \in I} C_i = V$
2. for any edge $(u, v) \in E(G)$, there exists $i \in I$ with $u, v \in C_i$
3. (r.i.p.) for any $v \in V$, the set $\{i \in I : v \in C_i\}$ forms a (nec. connected) subtree of T

Recap

- We want all original graph (o.g.) clique marginals. Why?

Recap

- We want all original graph (o.g.) clique marginals. Why?
- Finding optimal elimination order is optimal for all o.g. clique marginals.

Recap

- We want all original graph (o.g.) clique marginals. Why?
- Finding optimal elimination order is optimal for all o.g. clique marginals.
- Def: decomposition of a graph, and factorization implication.

Recap

- We want all original graph (o.g.) clique marginals. Why?
- Finding optimal elimination order is optimal for all o.g. clique marginals.
- Def: decomposition of a graph, and factorization implication.
- Def: decomposable graph, and decomposition tree

Recap

- We want all original graph (o.g.) clique marginals. Why?
- Finding optimal elimination order is optimal for all o.g. clique marginals.
- Def: decomposition of a graph, and factorization implication.
- Def: decomposable graph, and decomposition tree
- Thm: triangulated graph $\equiv$ decomposable graph

Recap

- We want all original graph (o.g.) clique marginals. Why?
- Finding optimal elimination order is optimal for all o.g. clique marginals.
- Def: decomposition of a graph, and factorization implication.
- Def: decomposable graph, and decomposition tree
- Thm: triangulated graph $\equiv$ decomposable graph
- Def: tree decomposition (vertex and edge cover, and induced sub-tree).

Recap

- We want all original graph (o.g.) clique marginals. Why?
- Finding optimal elimination order is optimal for all o.g. clique marginals.
- Def: decomposition of a graph, and factorization implication.
- Def: decomposable graph, and decomposition tree
- Thm: triangulated graph $\equiv$ decomposable graph
- Def: tree decomposition (vertex and edge cover, and induced sub-tree).
- Def: cluster graph, cluster tree, based only on o.g. nodes, not o.g. edges. Edges in cluster graph cluster tree via cluster intersection.

Recap

- We want all original graph (o.g.) clique marginals. Why?
- Finding optimal elimination order is optimal for all o.g. clique marginals.
- Def: decomposition of a graph, and factorization implication.
- Def: decomposable graph, and decomposition tree
- Thm: triangulated graph $\equiv$ decomposable graph
- Def: tree decomposition (vertex and edge cover, and induced sub-tree).
- Def: cluster graph, cluster tree, based only on o.g. nodes, not o.g. edges. Edges in cluster graph cluster tree via cluster intersection.
- Def: cluster intersection property, running intersection property, induced sub-tree property, r.i.p.

Recap

- We want all original graph (o.g.) clique marginals. Why?
- Finding optimal elimination order is optimal for all o.g. clique marginals.
- Def: decomposition of a graph, and factorization implication.
- Def: decomposable graph, and decomposition tree
- Thm: triangulated graph $\equiv$ decomposable graph
- Def: tree decomposition (vertex and edge cover, and induced sub-tree).
- Def: cluster graph, cluster tree, based only on o.g. nodes, not o.g. edges. Edges in cluster graph cluster tree via cluster intersection.
- Def: cluster intersection property, running intersection property, induced sub-tree property, r.i.p.
- Next def: Junction tree, cluster tree with r.i.p. and edge cover.

Junction Tree

Definition 5.6.9

Given a graph $G = (V, E)$, a junction tree corresponding to G (if it exists) is a cluster tree $\mathcal{T} = (\mathcal{C}, E_T)$ having the r.i.p. over the clusters, and where the nodes u, v adjacent to every edge $(u, v) \in E(G)$ are together in at least one cluster.

Junction Tree

Definition 5.6.9

Given a graph $G = (V, E)$, a junction tree corresponding to G (if it exists) is a cluster tree $\mathcal{T} = (\mathcal{C}, E_T)$ having the r.i.p. over the clusters, and where the nodes u, v adjacent to every edge $(u, v) \in E(G)$ are together in at least one cluster.

- So, junction tree (JT), for a given graph G, is a cluster tree that: 1) satisfies r.i.p. over the clusters, and 2) includes all edges (edge cover). Not all r.i.p.-satisfying cluster trees need be an edge cover.

Junction Tree

Definition 5.6.9

Given a graph $G = (V, E)$, a junction tree corresponding to G (if it exists) is a cluster tree $\mathcal{T} = (\mathcal{C}, E_T)$ having the r.i.p. over the clusters, and where the nodes u, v adjacent to every edge $(u, v) \in E(G)$ are together in at least one cluster.

- So, junction tree (JT), for a given graph G, is a cluster tree that: 1) satisfies r.i.p. over the clusters, and 2) includes all edges (edge cover). Not all r.i.p.-satisfying cluster trees need be an edge cover.
- Clusters in JT need not be original graph cliques!!

Junction Tree

Definition 5.6.9

Given a graph $G = (V, E)$, a junction tree corresponding to G (if it exists) is a cluster tree $\mathcal{T} = (\mathcal{C}, E_T)$ having the r.i.p. over the clusters, and where the nodes u, v adjacent to every edge $(u, v) \in E(G)$ are together in at least one cluster.

- So, junction tree (JT), for a given graph G, is a cluster tree that: 1) satisfies r.i.p. over the clusters, and 2) includes all edges (edge cover). Not all r.i.p.-satisfying cluster trees need be an edge cover.
- Clusters in JT need not be original graph cliques!!
- JT could have clusters corresponding to cliques, maxcliques, or neither of the above.

Junction Tree

Definition 5.6.9

Given a graph $G = (V, E)$, a junction tree corresponding to G (if it exists) is a cluster tree $\mathcal{T} = (\mathcal{C}, E_T)$ having the r.i.p. over the clusters, and where the nodes u, v adjacent to every edge $(u, v) \in E(G)$ are together in at least one cluster.

- So, junction tree (JT), for a given graph G, is a cluster tree that: 1) satisfies r.i.p. over the clusters, and 2) includes all edges (edge cover). Not all r.i.p.-satisfying cluster trees need be an edge cover.
- Clusters in JT need not be original graph cliques!!
- JT could have clusters corresponding to cliques, maxcliques, or neither of the above.
- If clusters correspond to the original graph cliques (resp. maxcliques) in G, it called a junction tree of cliques (resp. maxcliques).

Junction Tree Preserving Operations

Lemma 5.6.10

Given a junction tree, form a new cluster tree as follows. For each cluster C in the JT, choose an order of nodes within C, say $c_1, c_2, \ldots, c_k$, and hang a chain of clusters off of C consisting of $C \setminus \{c_1\}$ hanging from C, $C \setminus \{c_1, c_2\}$ hanging from $C \setminus \{c_1\}$, $C \setminus \{c_1, c_2, c_3\}$ hanging from $C \setminus \{c_1, c_2\}$, and so on. Then the resulting cluster graph is a cluster tree, and moreover it is still junction tree.

Junction Tree Preserving Operations

Lemma 5.6.10

Given a junction tree, form a new cluster tree as follows. For each cluster C in the JT, choose an order of nodes within C, say $c_1, c_2, \ldots, c_k$, and hang a chain of clusters off of C consisting of $C \setminus \{c_1\}$ hanging from C, $C \setminus \{c_1, c_2\}$ hanging from $C \setminus \{c_1\}$, $C \setminus \{c_1, c_2, c_3\}$ hanging from $C \setminus \{c_1, c_2\}$, and so on. Then the resulting cluster graph is a cluster tree, and moreover it is still junction tree.

Lemma 5.6.11

Given a junction tree, where (C_i, C_j) are neighboring clusters in the tree, we can merge these two clusters forming a new cluster $C_{ij} = C_i \cup C_j$, and where the neighbors of C_{ij} are the set of neighbors of either C_i and C_j. Then the resulting structure is still junction tree.

Junction Tree Preserving Operations

Lemma 5.6.10

Given a junction tree, form a new cluster tree as follows. For each cluster C in the JT, choose an order of nodes within C, say $c_1, c_2, \ldots, c_k$, and hang a chain of clusters off of C consisting of $C \setminus \{c_1\}$ hanging from C, $C \setminus \{c_1, c_2\}$ hanging from $C \setminus \{c_1\}$, $C \setminus \{c_1, c_2, c_3\}$ hanging from $C \setminus \{c_1, c_2\}$, and so on. Then the resulting cluster graph is a cluster tree, and moreover it is still junction tree.

Lemma 5.6.11

Given a junction tree, where (C_i, C_j) are neighboring clusters in the tree, we can merge these two clusters forming a new cluster $C_{ij} = C_i \cup C_j$, and where the neighbors of C_{ij} are the set of neighbors of either C_i and C_j. Then the resulting structure is still junction tree.

If we keep doing the latter, we'll end up with one complete graph.

Sources for Today's Lecture

- Most of this material comes from the reading handout `tree_inference.pdf`