Part 4: Structural Decompositions and Algorithms

Friedrich Slivovsky

Dynamic Programming on Tree Decompositions

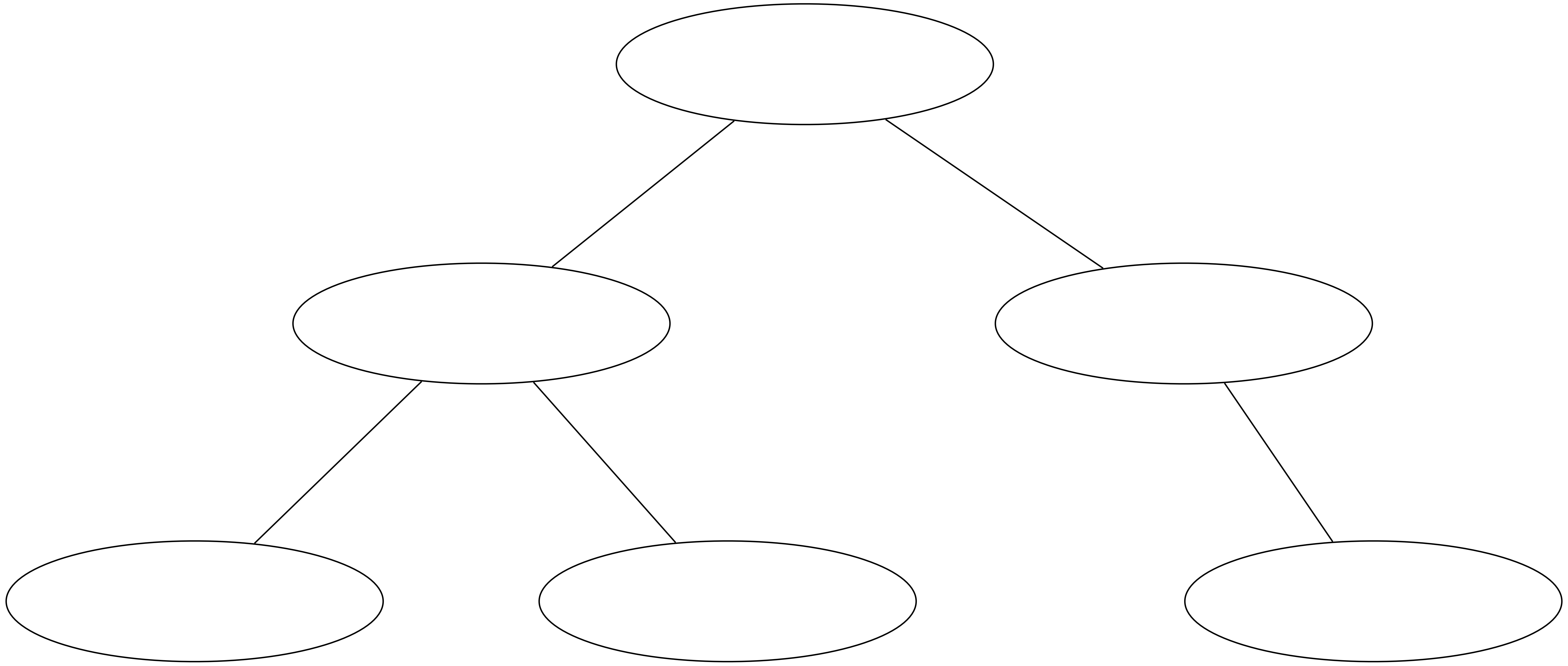
Dynamic Programming on Tree Decompositions

Dynamic Programming on Tree Decompositions

1. Compute a (nice) tree decomposition.

Dynamic Programming on Tree Decompositions

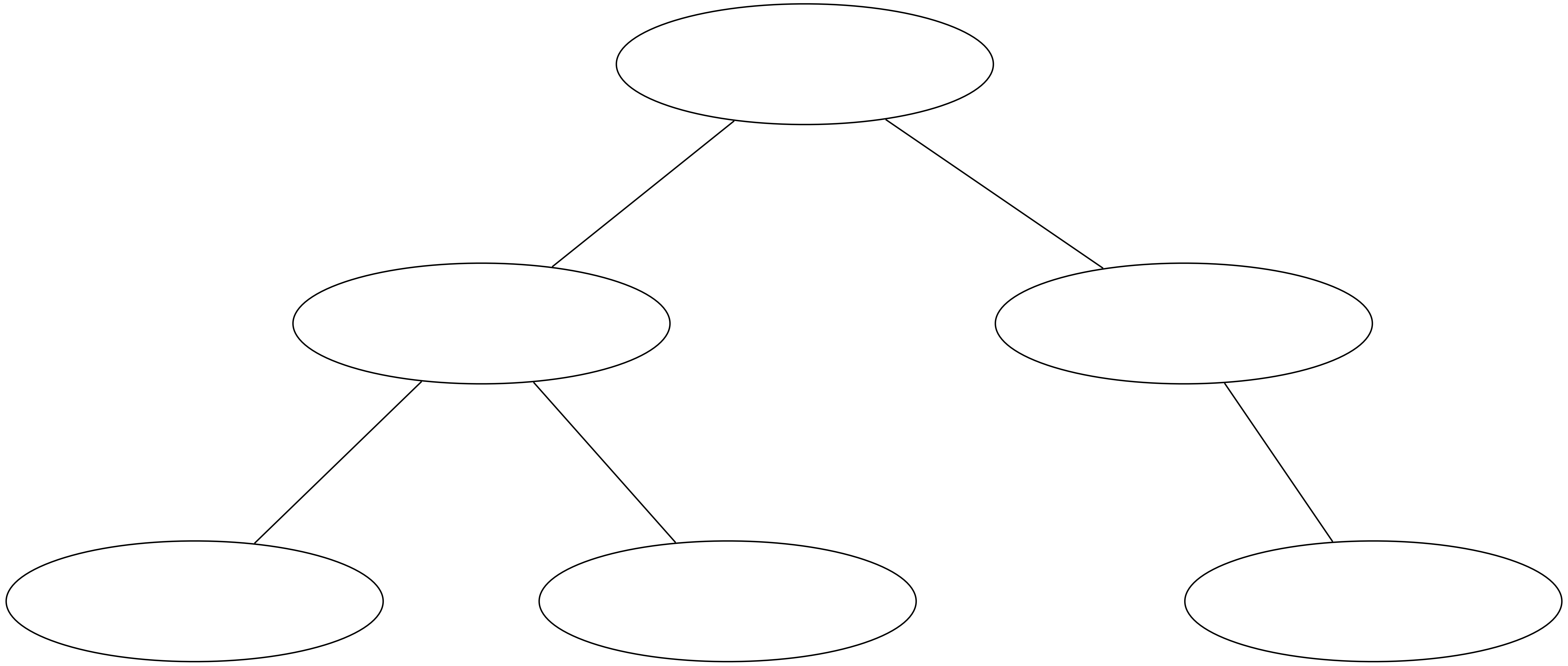
1. Compute a (nice) tree decomposition.



Dynamic Programming on Tree Decompositions

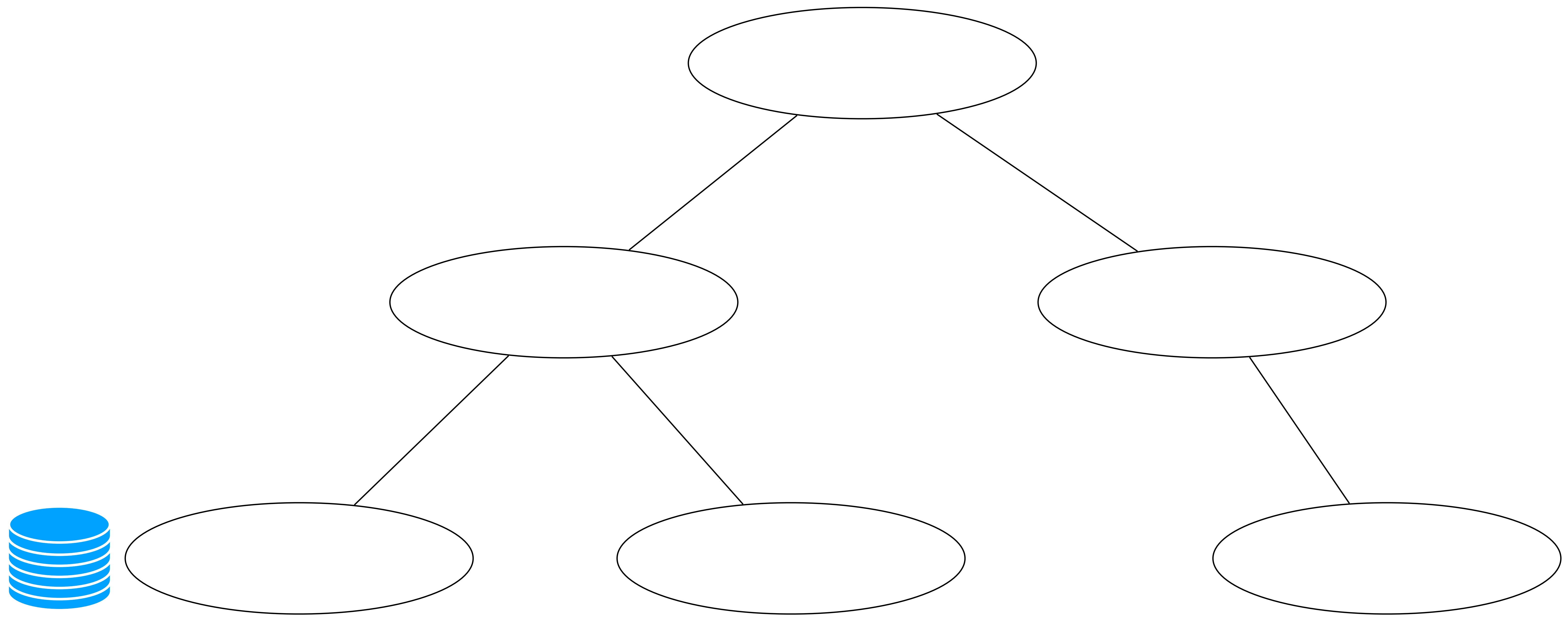
1. Compute a (nice) tree decomposition.

2. Go from leaves to root and compute table for each node.



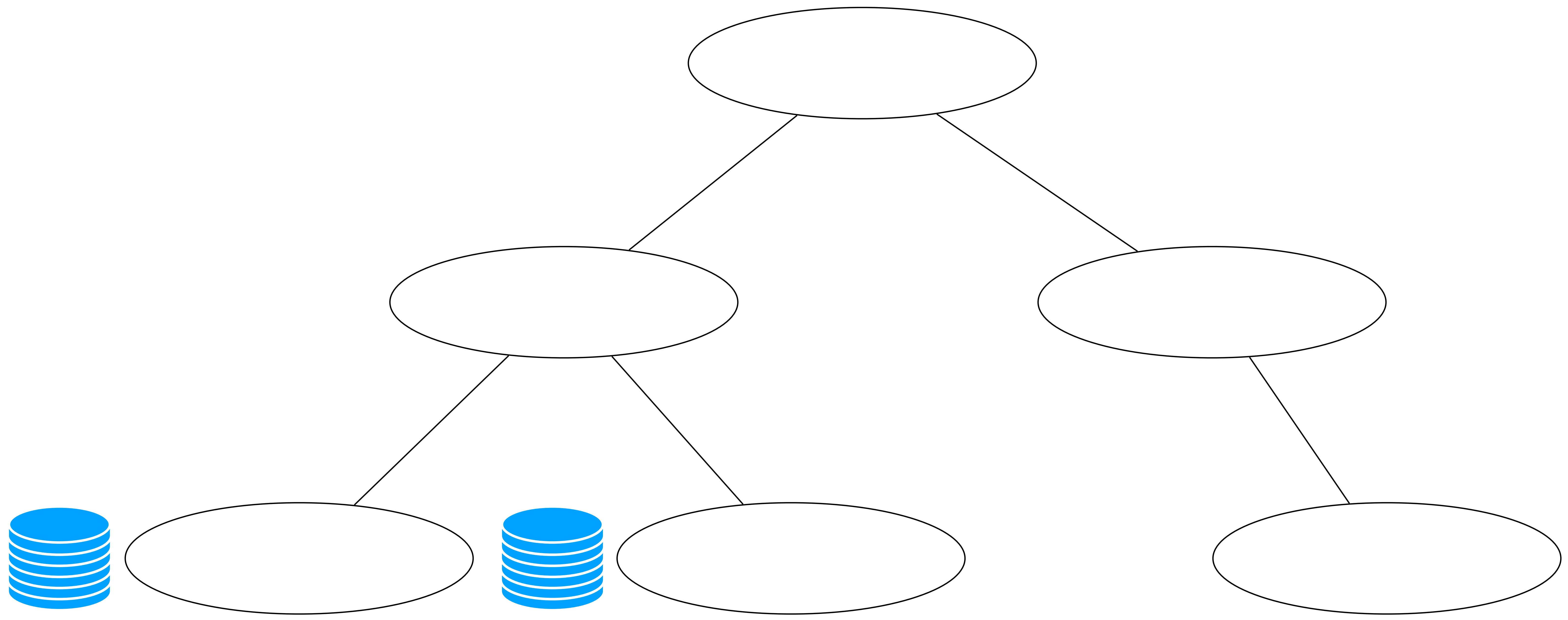
Dynamic Programming on Tree Decompositions

1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node.



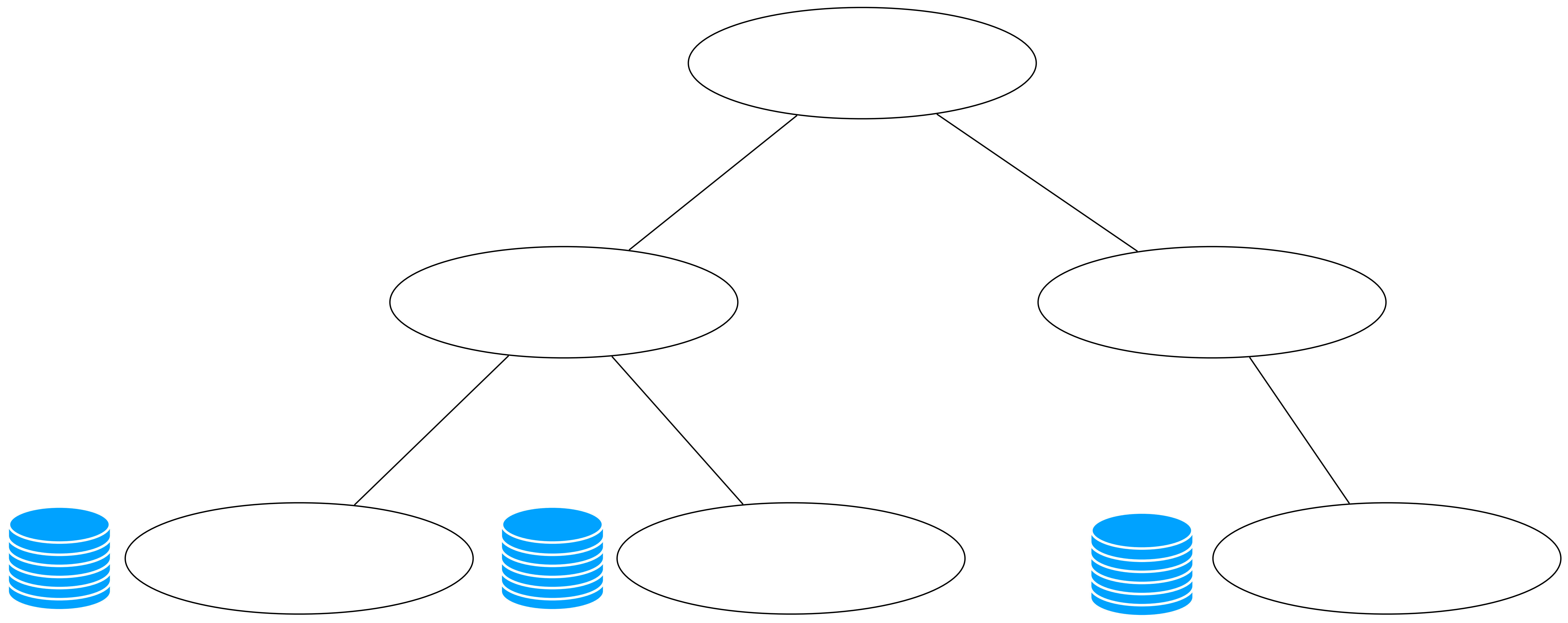
Dynamic Programming on Tree Decompositions

1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node.



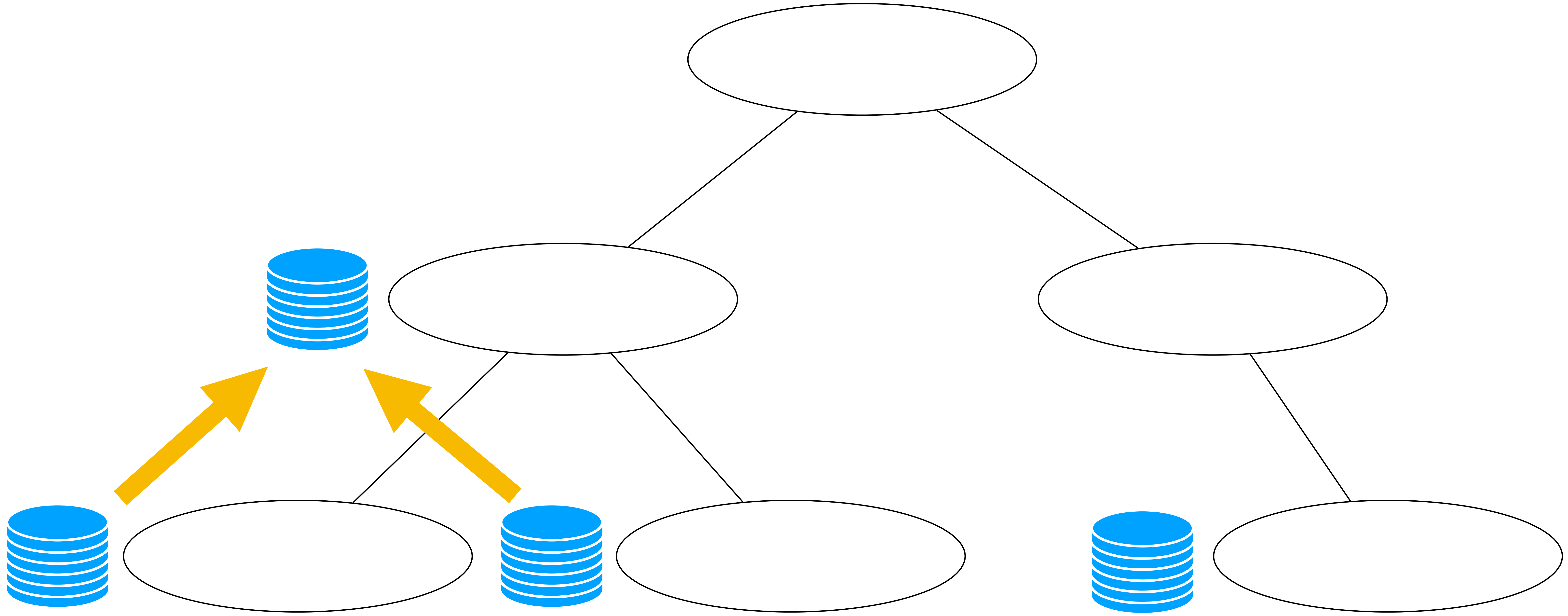
Dynamic Programming on Tree Decompositions

1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node.



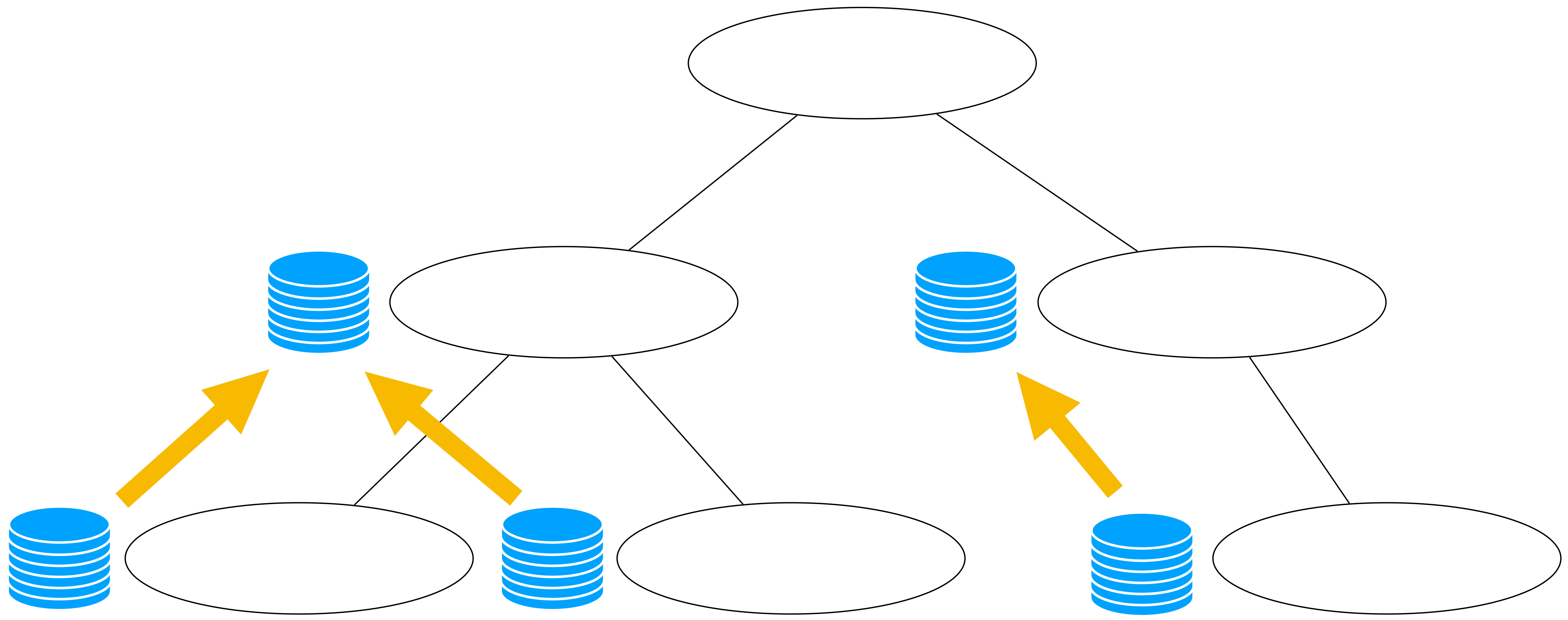
Dynamic Programming on Tree Decompositions

1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node.



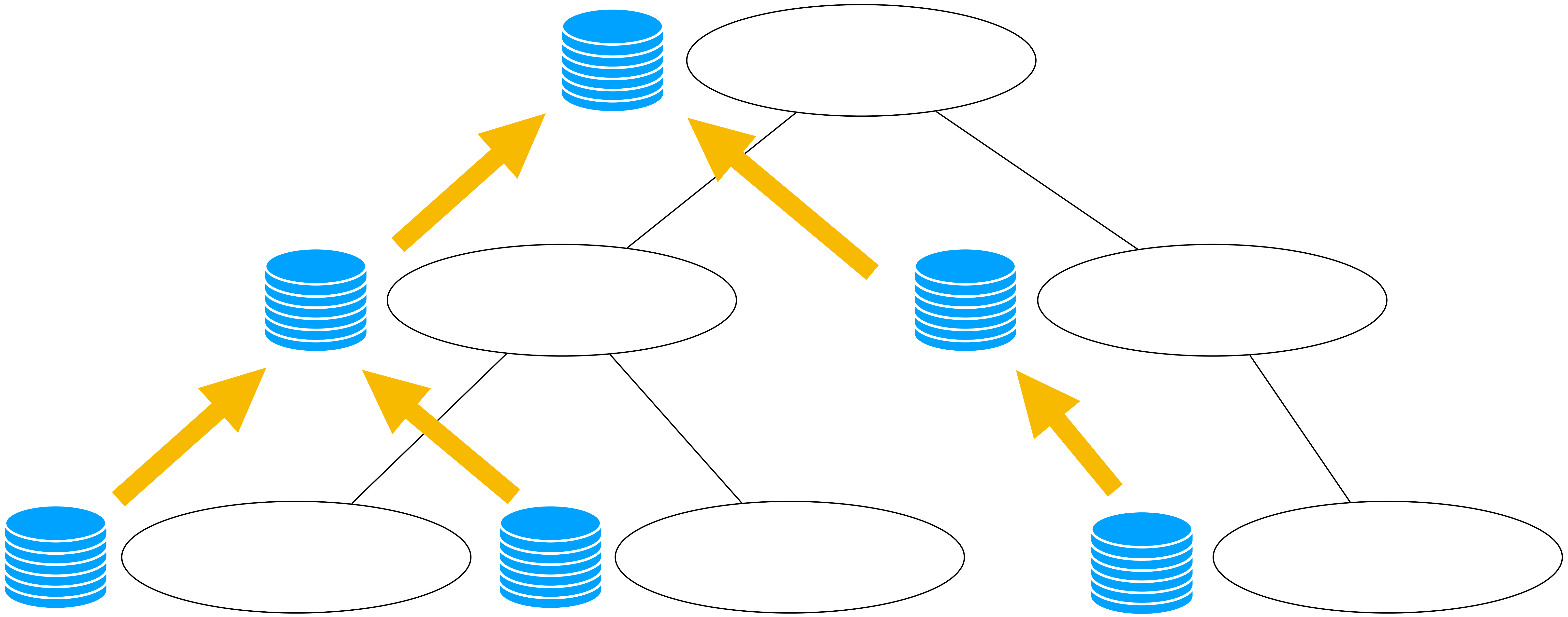
Dynamic Programming on Tree Decompositions

1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node.



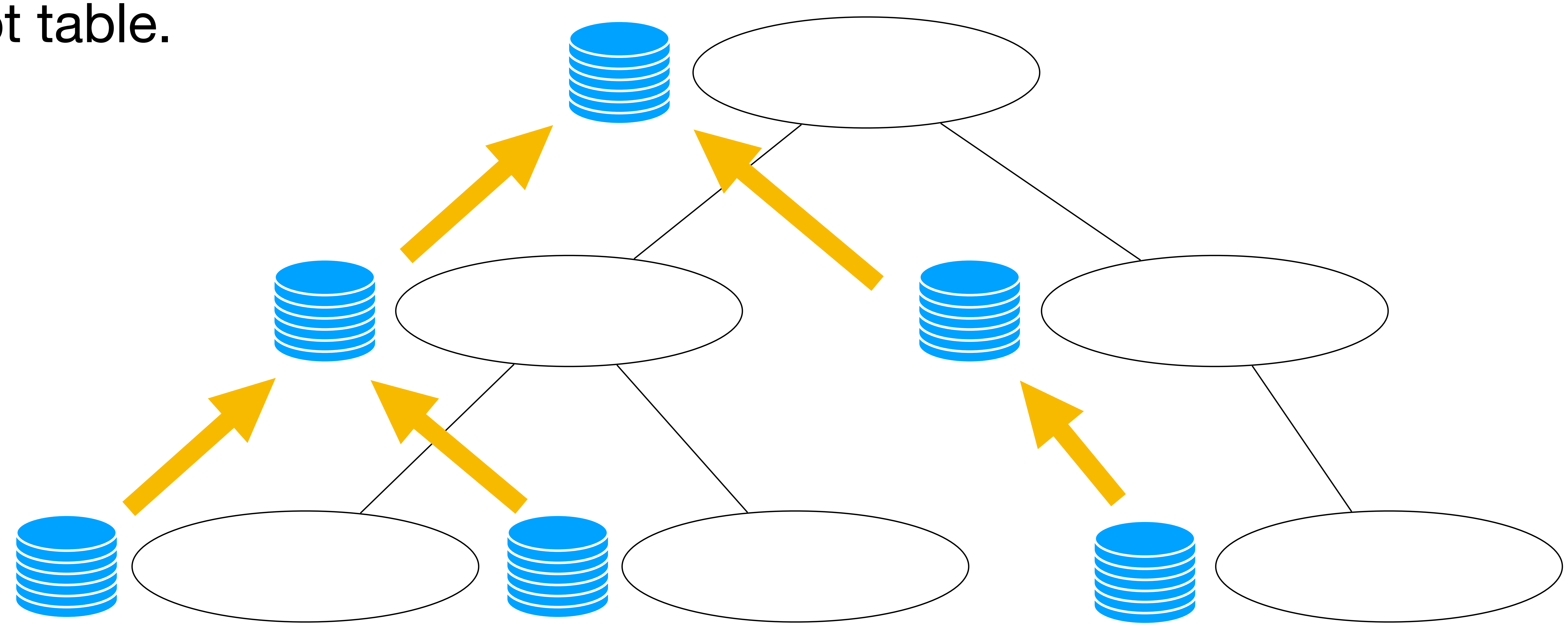
Dynamic Programming on Tree Decompositions

1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node.



Dynamic Programming on Tree Decompositions

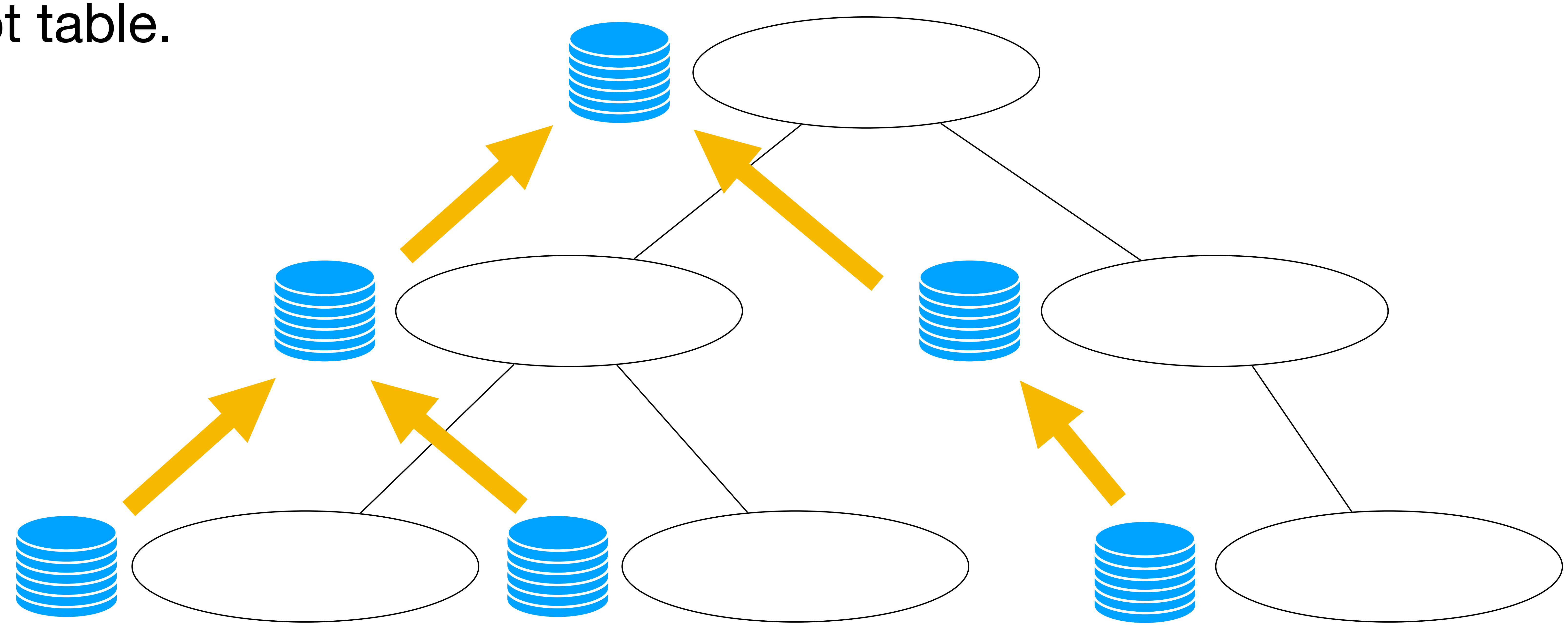
1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node.
3. Retrieve solution from root table.



Dynamic Programming on Tree Decompositions

Treewidth k

1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node.
3. Retrieve solution from root table.

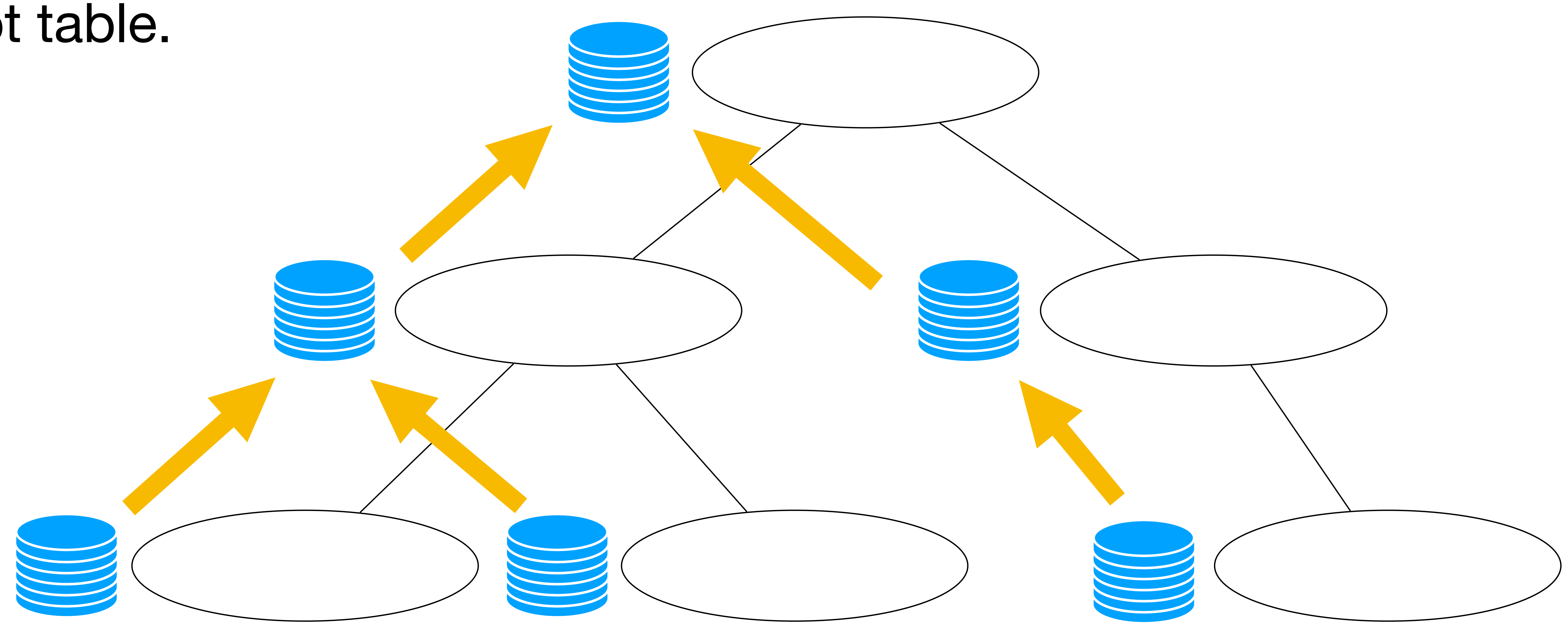


Dynamic Programming on Tree Decompositions

$f(k)\,p(n)$

Treewidth k

1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node.
3. Retrieve solution from root table.

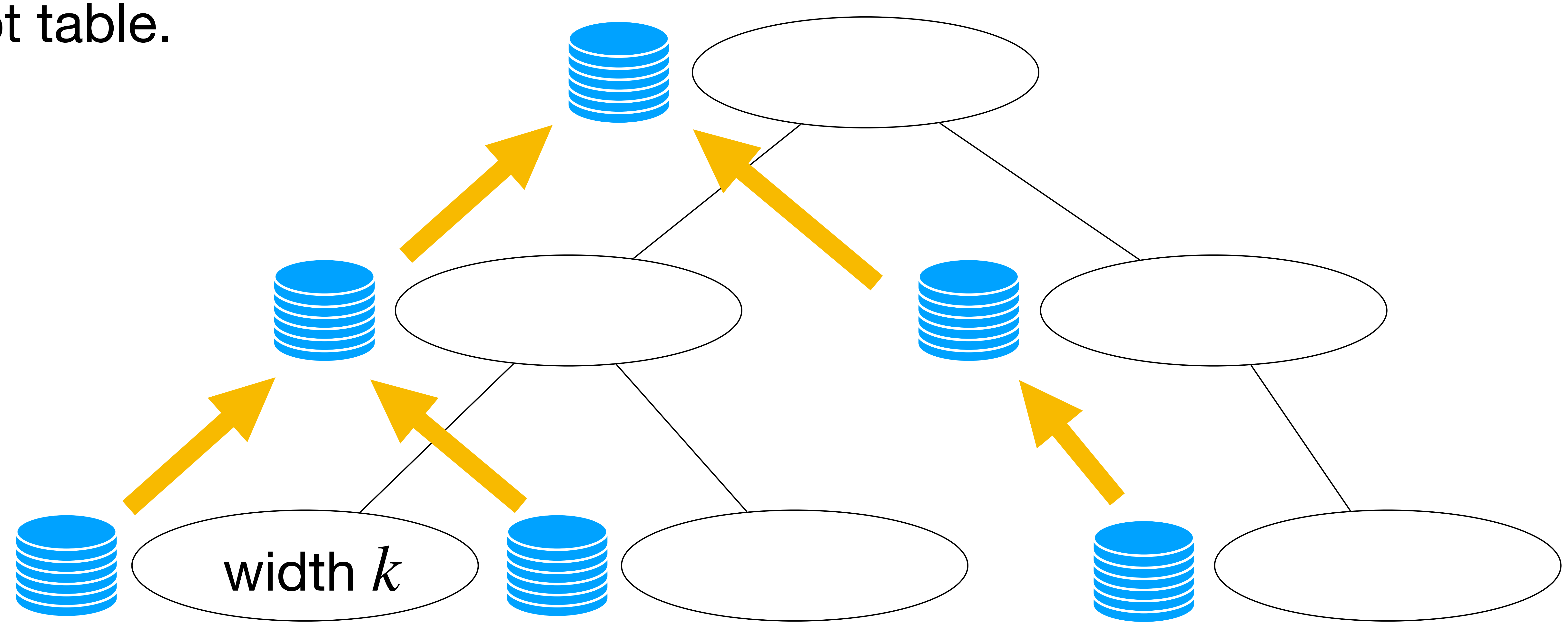


Dynamic Programming on Tree Decompositions

$f(k)\,p(n)$

Treewidth k

1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node.
3. Retrieve solution from root table.

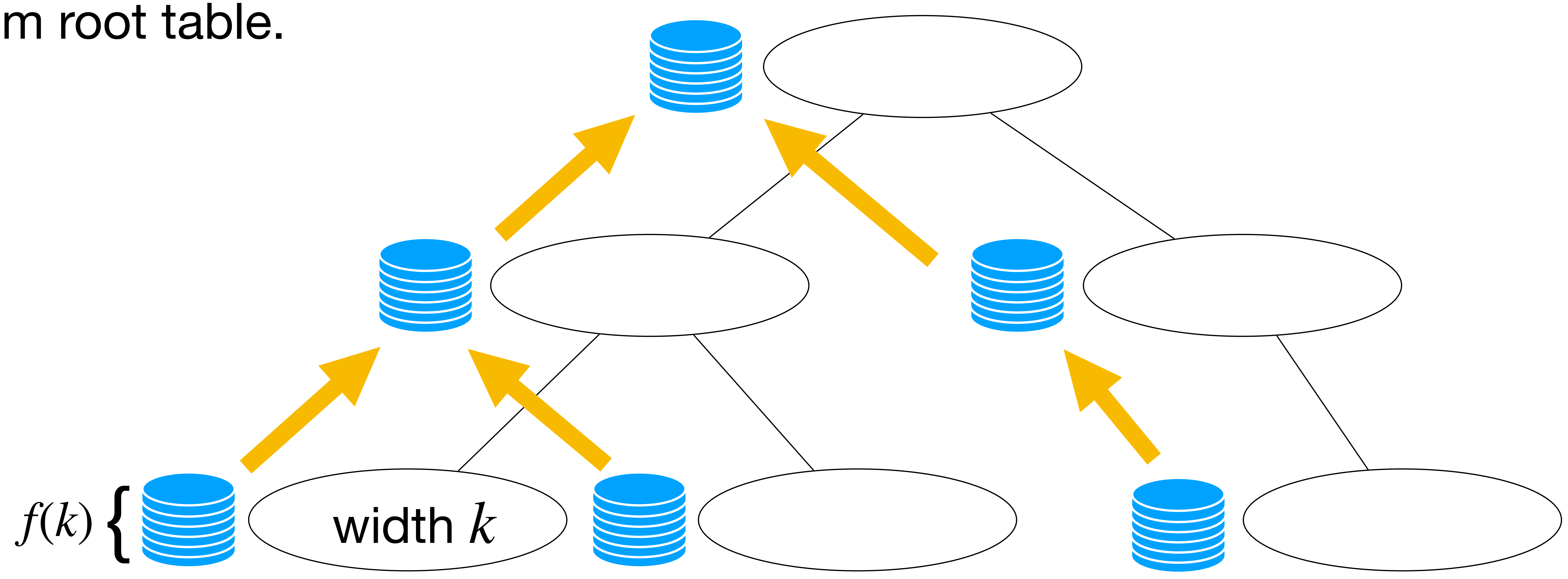


Dynamic Programming on Tree Decompositions

$f(k)\,p(n)$

Treewidth k

1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node.
3. Retrieve solution from root table.

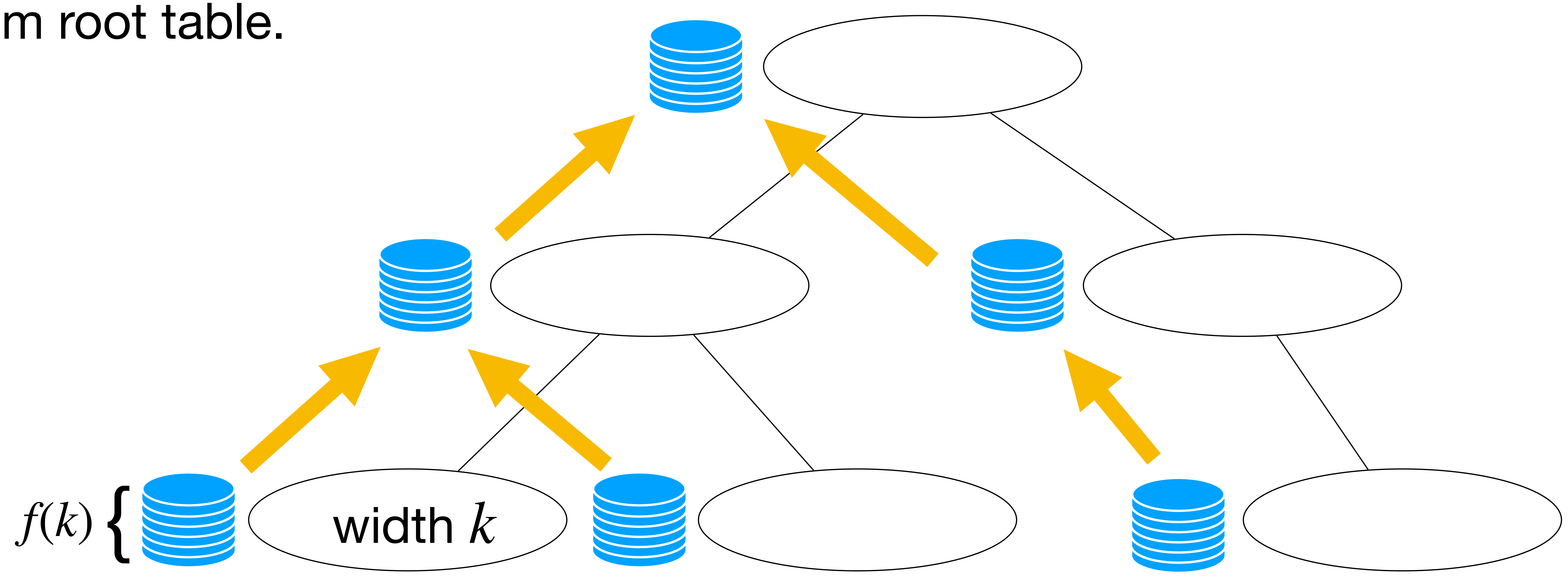


Dynamic Programming on Tree Decompositions

$f(k)\,p(n)$

Treewidth k

1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node. $g(k)\,p'(n)$
3. Retrieve solution from root table.



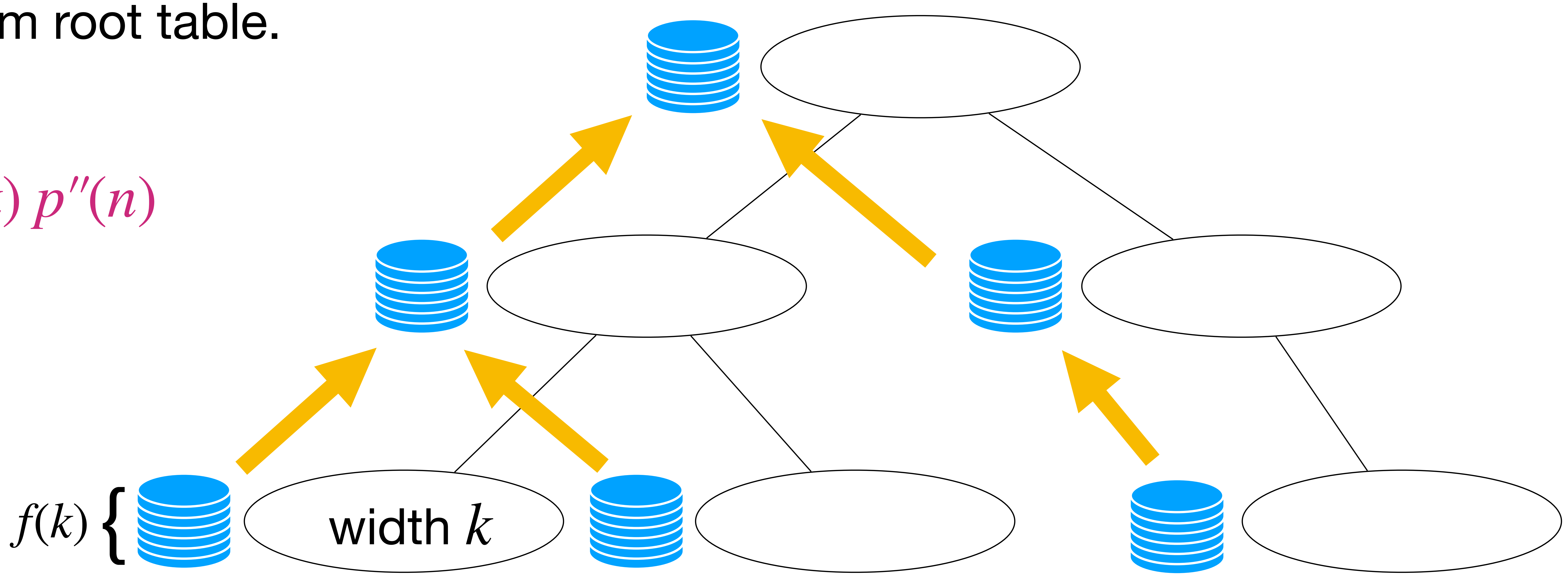
Dynamic Programming on Tree Decompositions

$f(k)\, p(n)$

Treewidth k

1. Compute a (nice) tree decomposition.
2. Go from leaves to root and compute table for each node. $g(k)\, p'(n)$
3. Retrieve solution from root table.

$h(k)\, p''(n)$



Independent Set

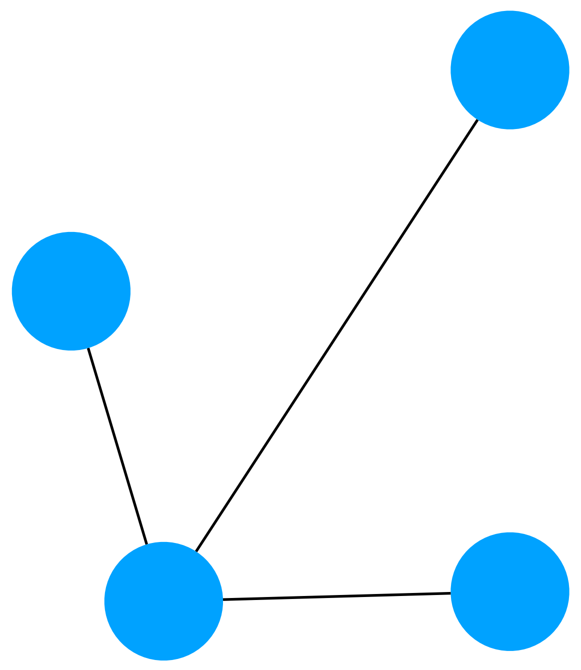
Definition

An **independent set** of a graph is a subset of vertices such that no two vertices are adjacent.

Independent Set

Definition

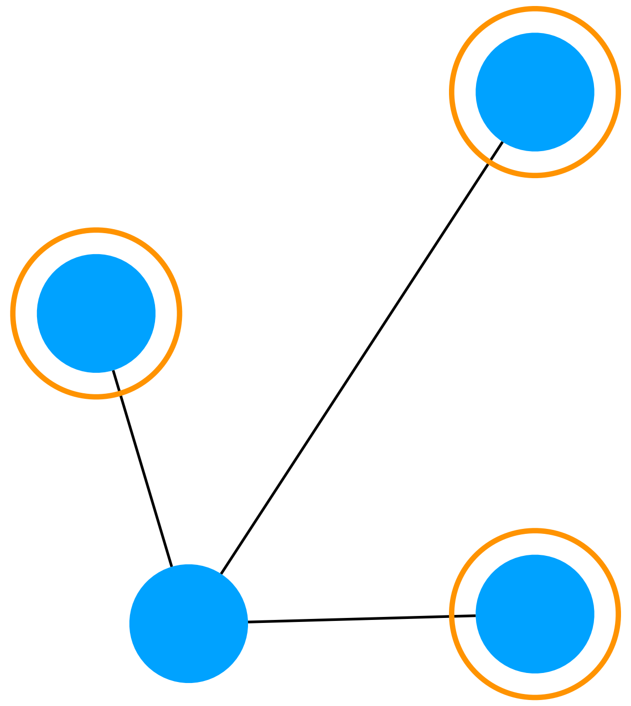
An **independent set** of a graph is a subset of vertices such that no two vertices are adjacent.



Independent Set

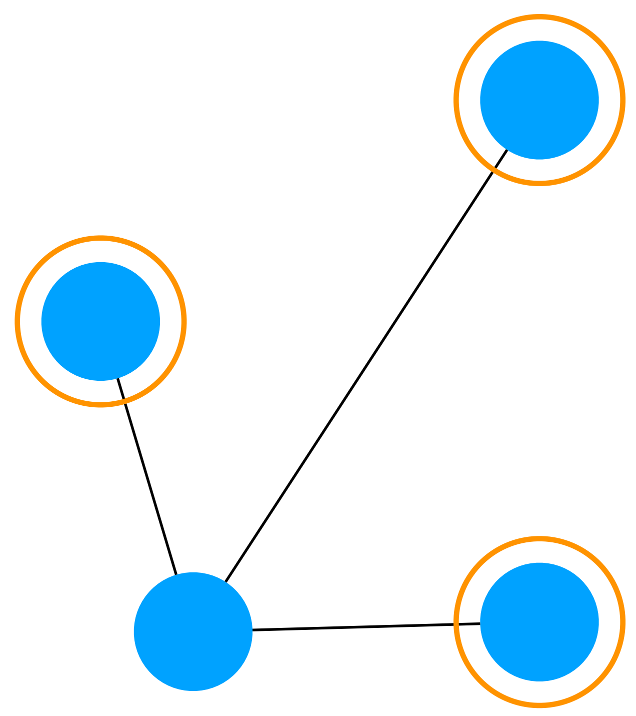
Definition

An **independent set** of a graph is a subset of vertices such that no two vertices are adjacent.



Independent Set

Definition
An **independent set** of a graph is a subset of vertices such that no two vertices are adjacent.



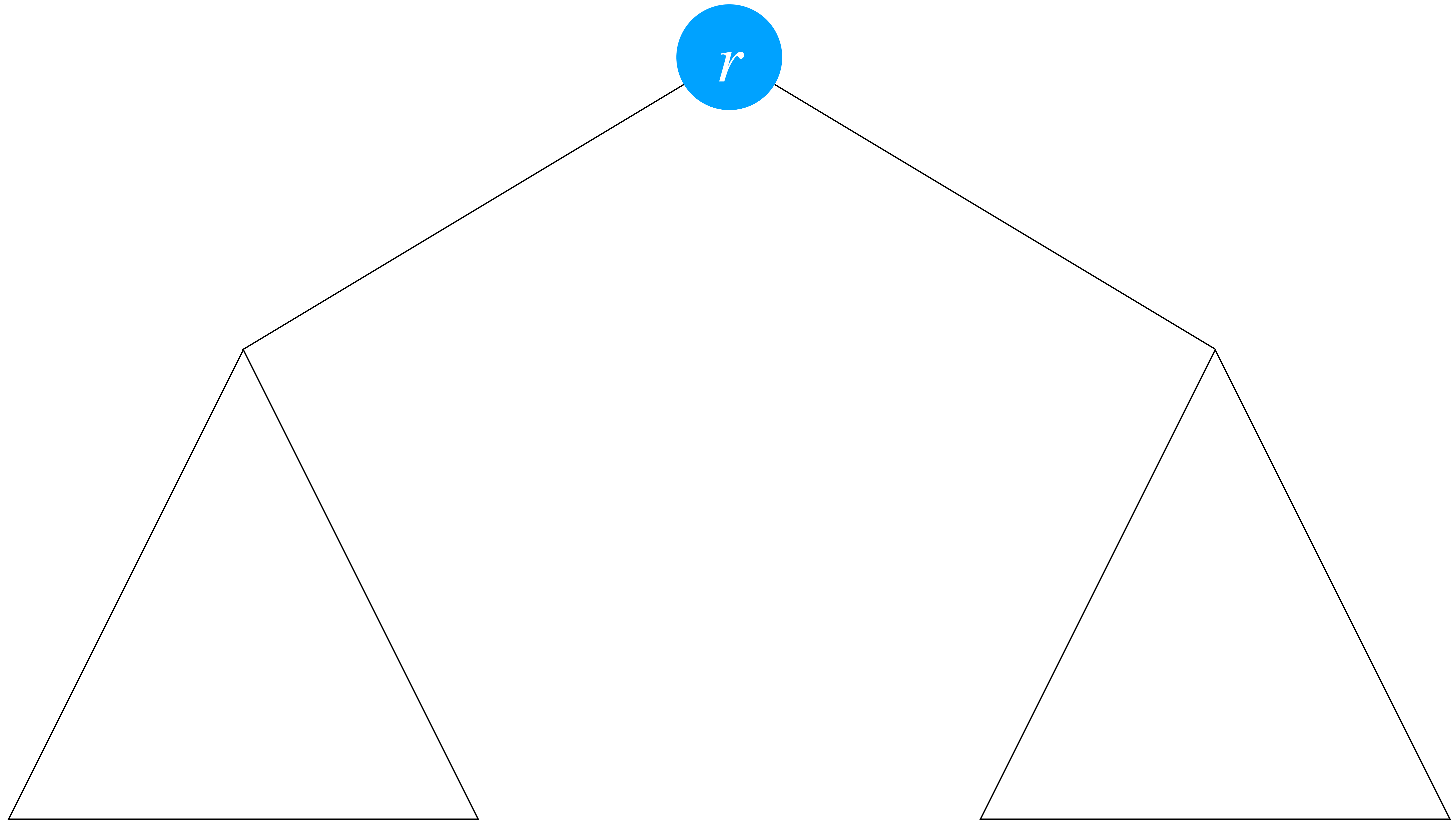
INDEPENDENT SET

Input: A graph G and an integer k.

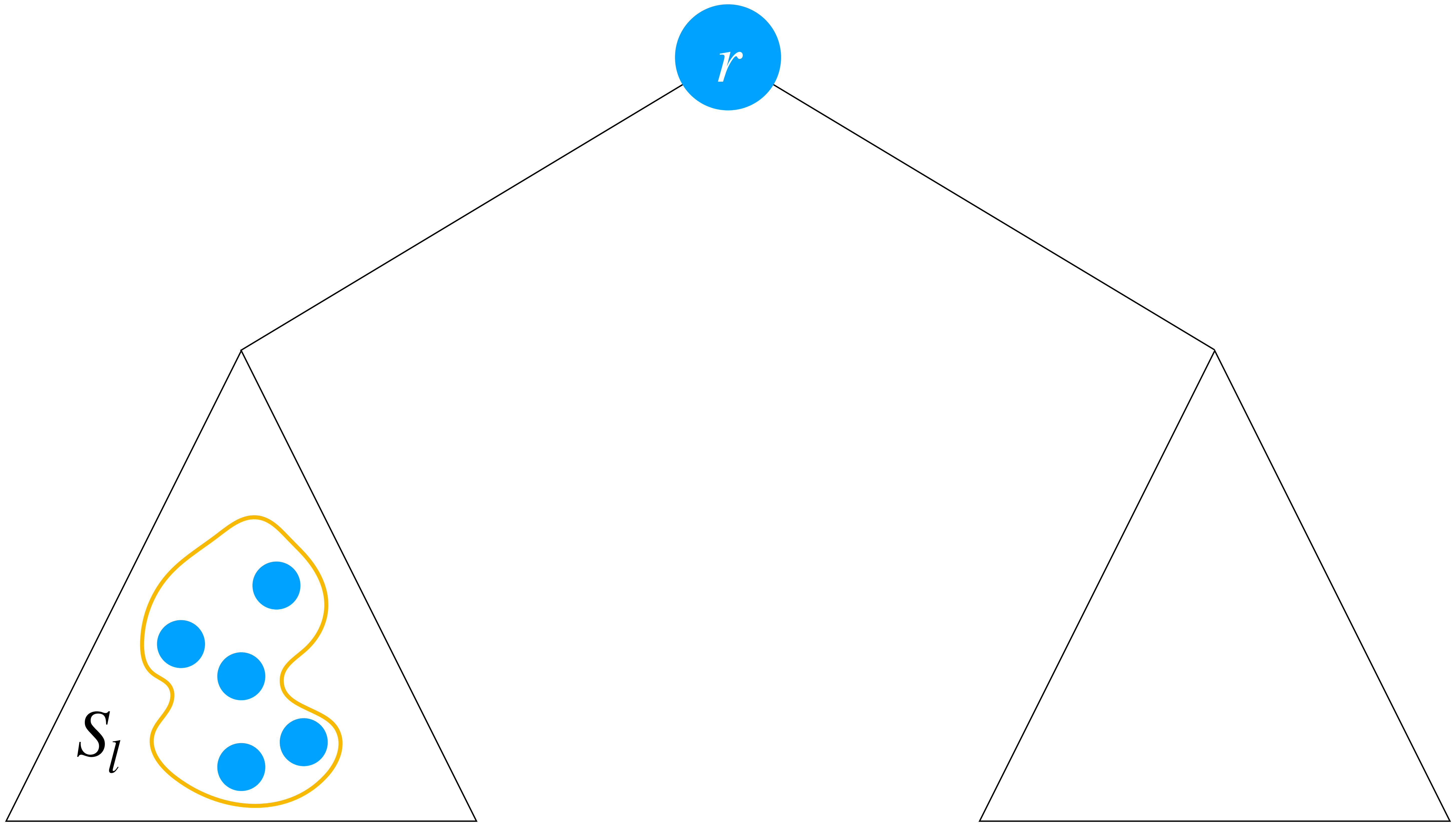
Question: Does G have an **IS** of size k?

Independent Set on Trees

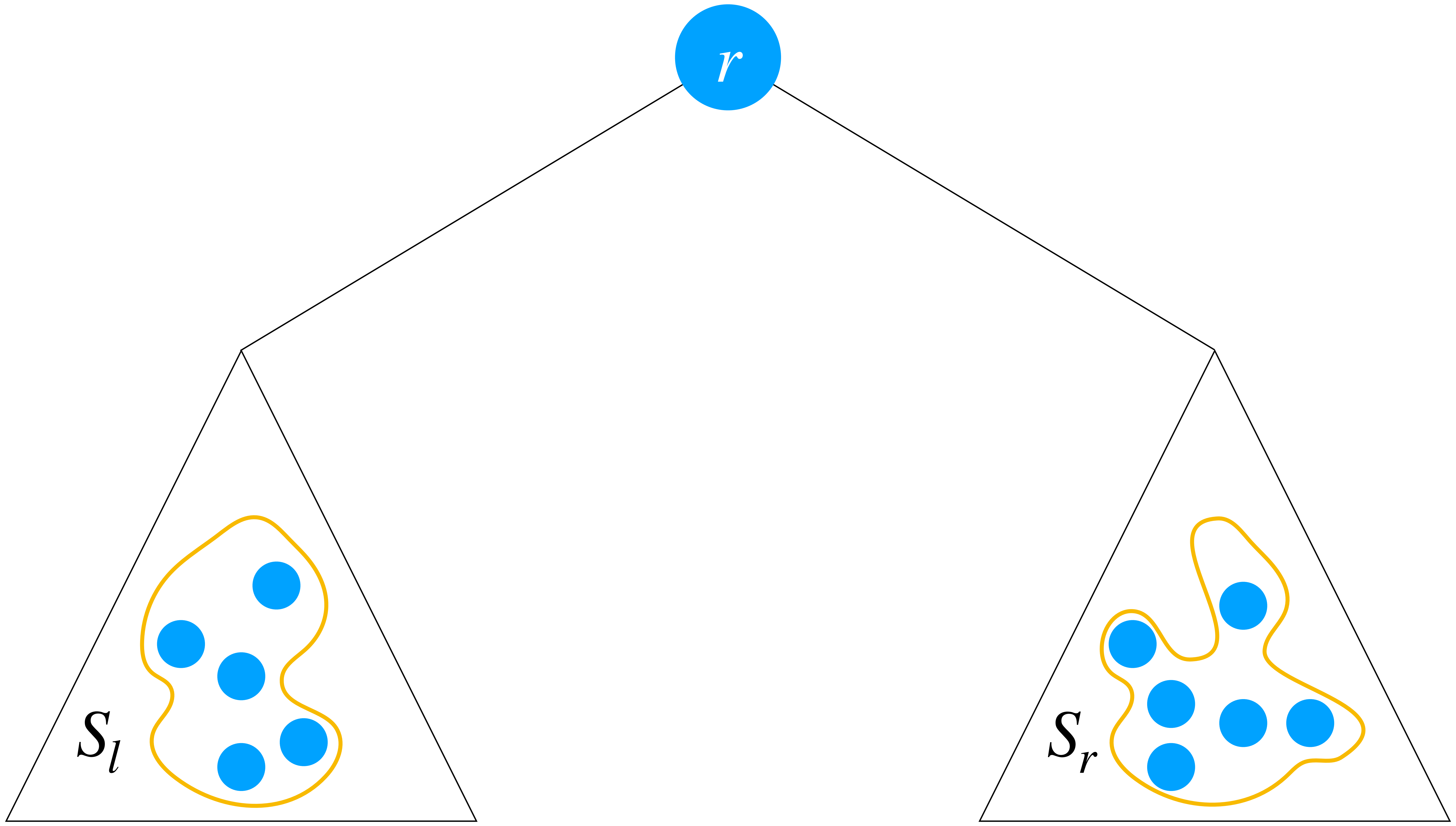
Independent Set on Trees



Independent Set on Trees

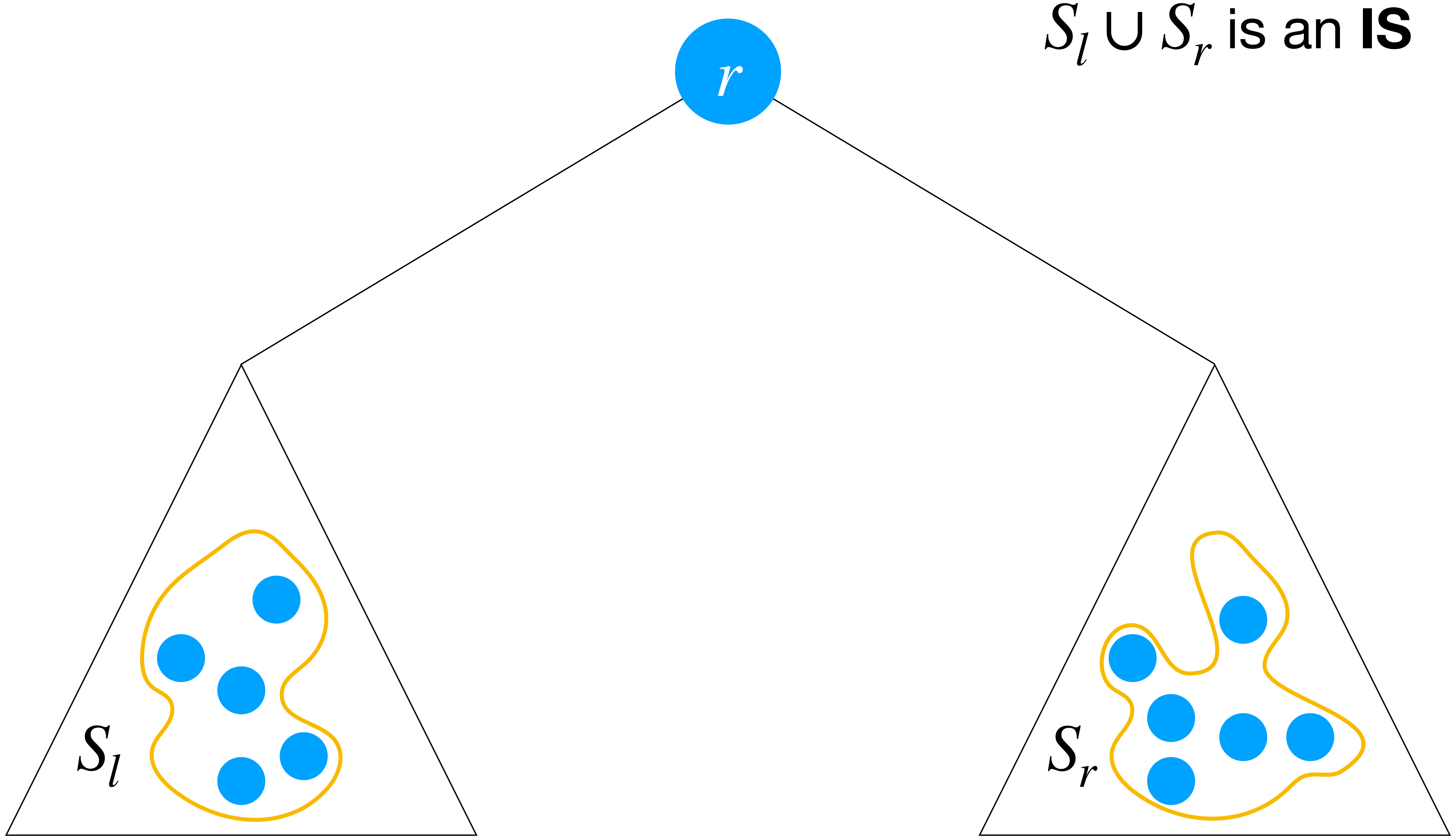


Independent Set on Trees



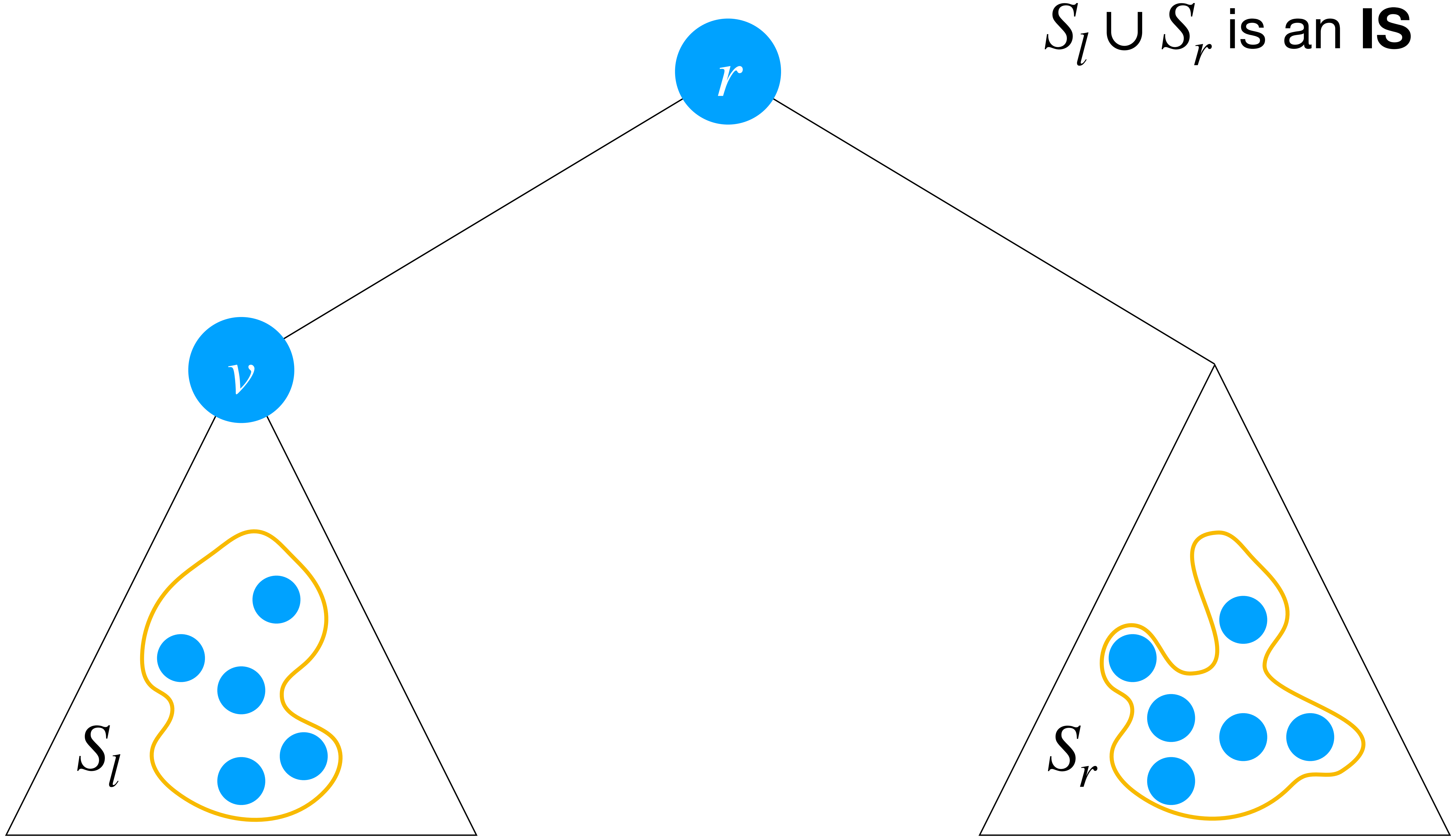
Independent Set on Trees

$S_l \cup S_r$ is an **IS**



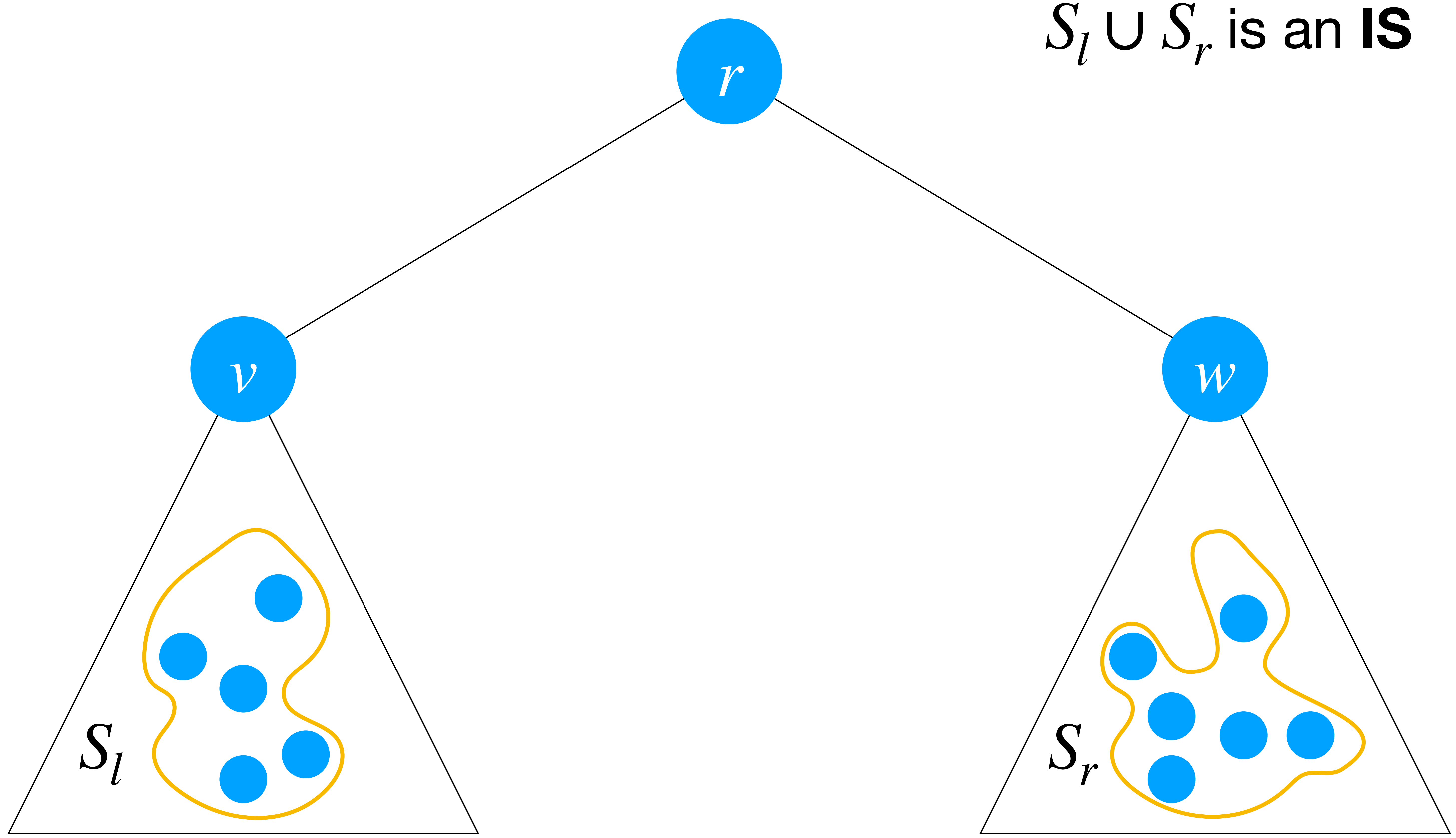
Independent Set on Trees

$S_l \cup S_r$ is an **IS**

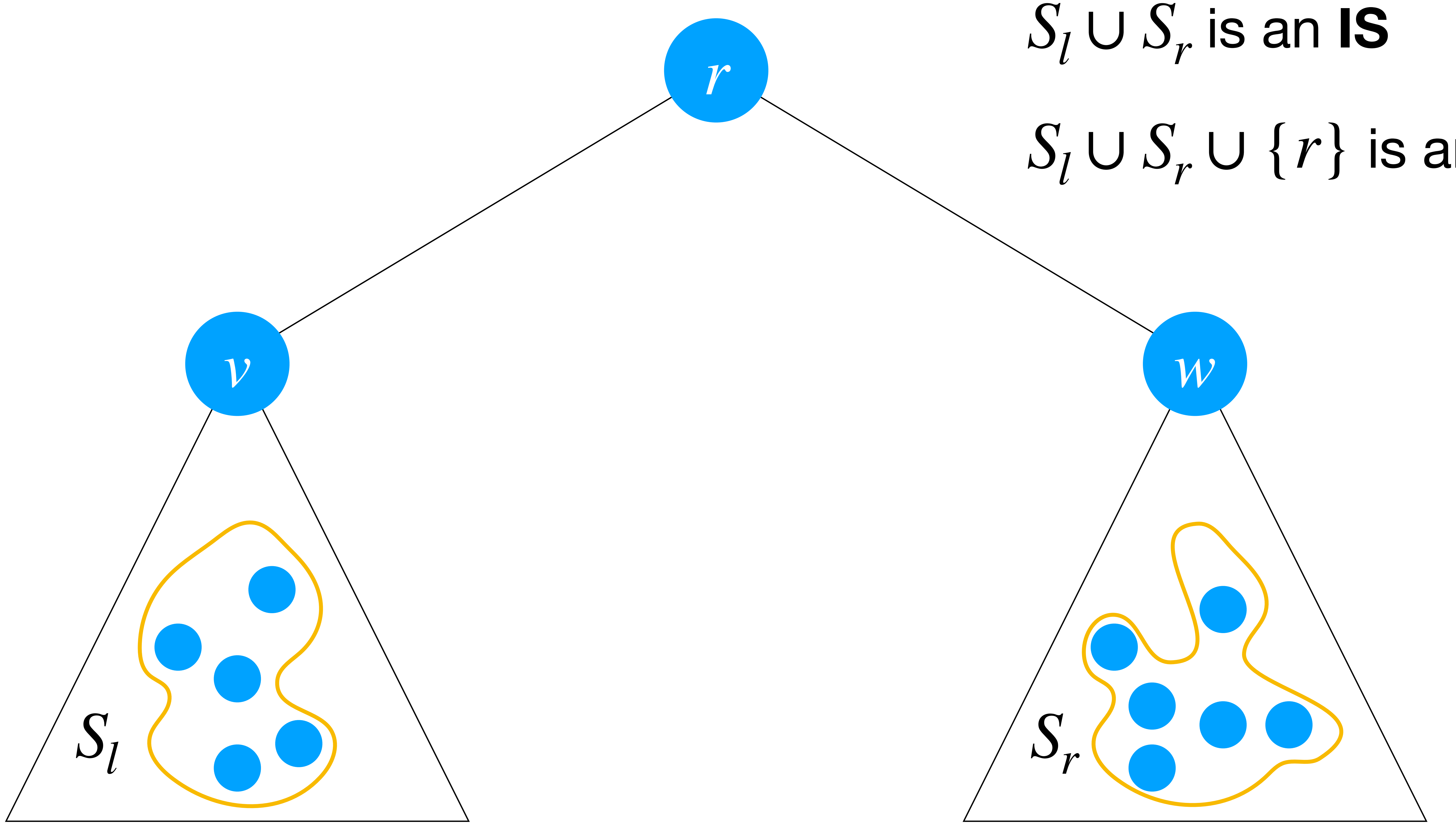


Independent Set on Trees

$S_l \cup S_r$ is an **IS**



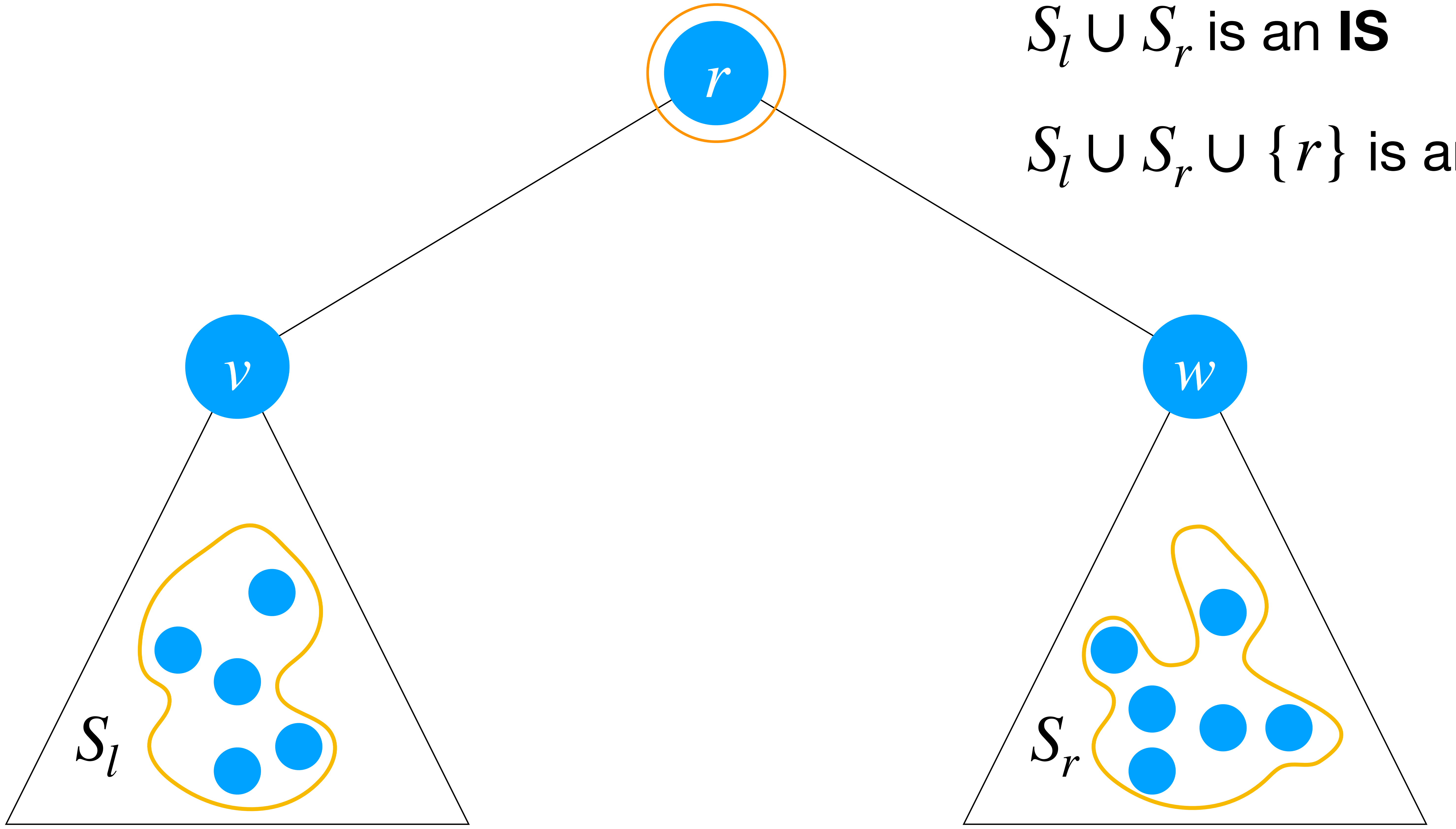
Independent Set on Trees



$S_l \cup S_r$ is an **IS**

$S_l \cup S_r \cup \{r\}$ is an **IS** if $v \notin S_l$ and $w \notin S_r$

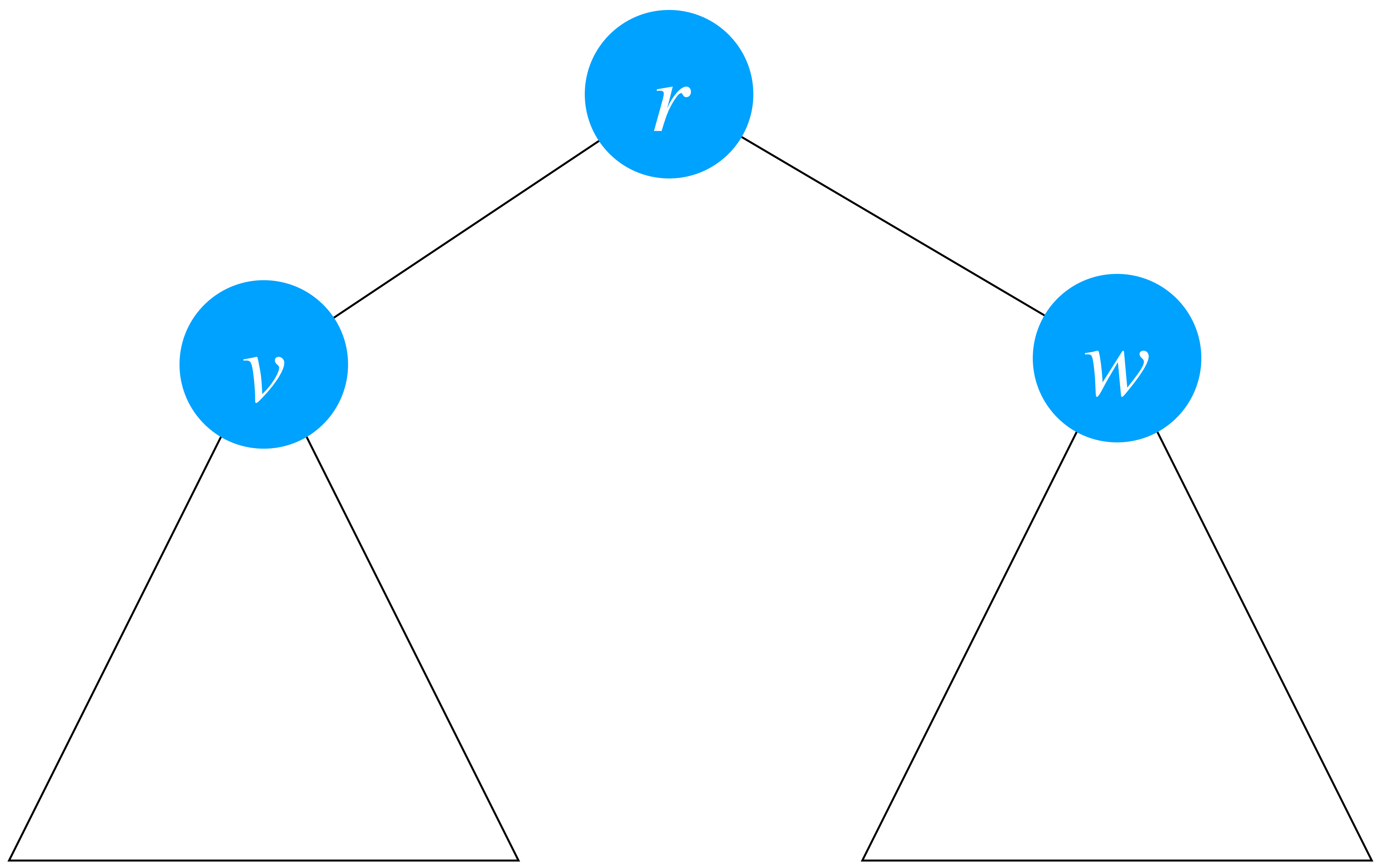
Independent Set on Trees



$S_l \cup S_r$ is an **IS**

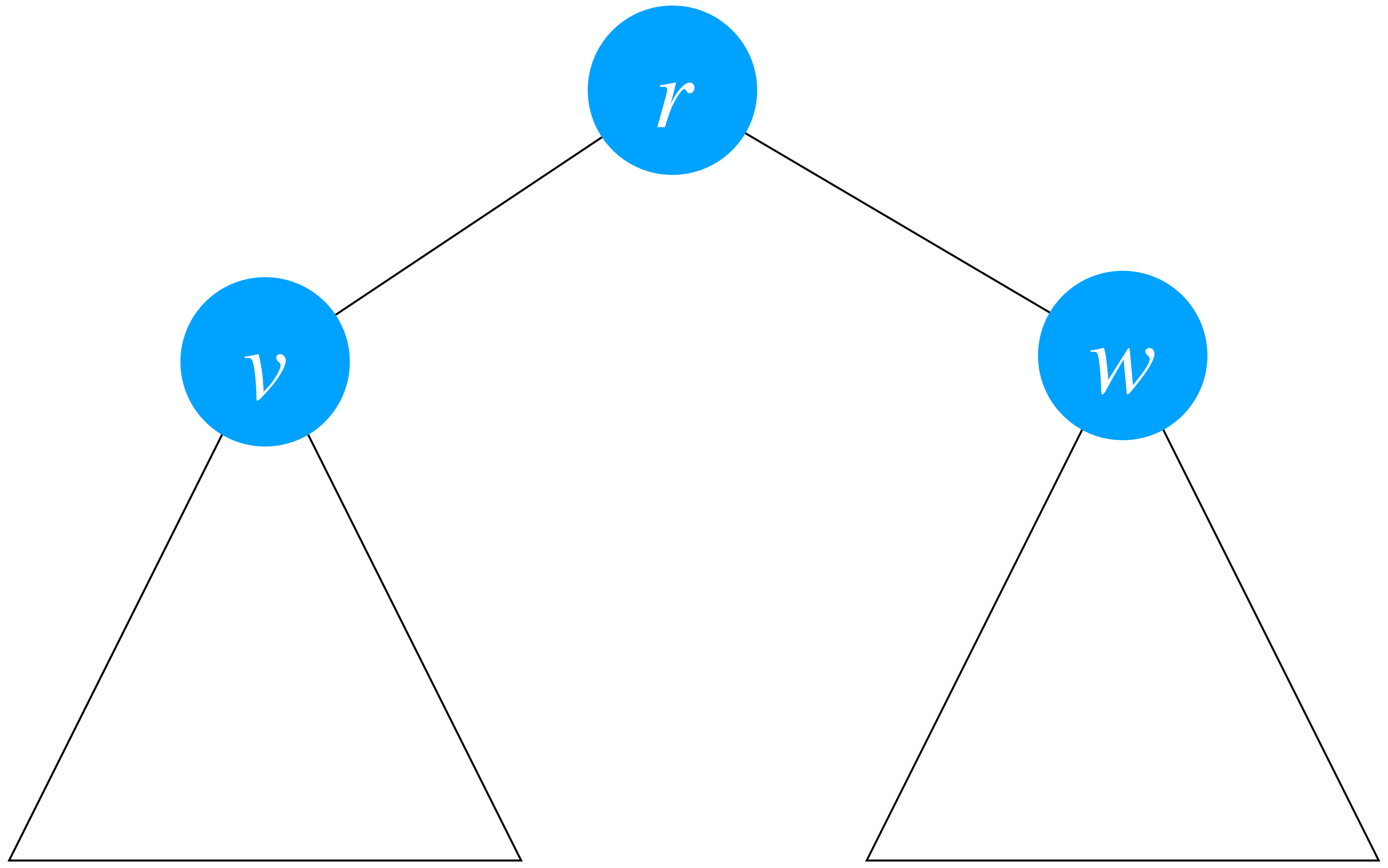
$S_l \cup S_r \cup \{r\}$ is an **IS** if $v \notin S_l$ and $w \notin S_r$

Independent Set on Trees



Independent Set on Trees

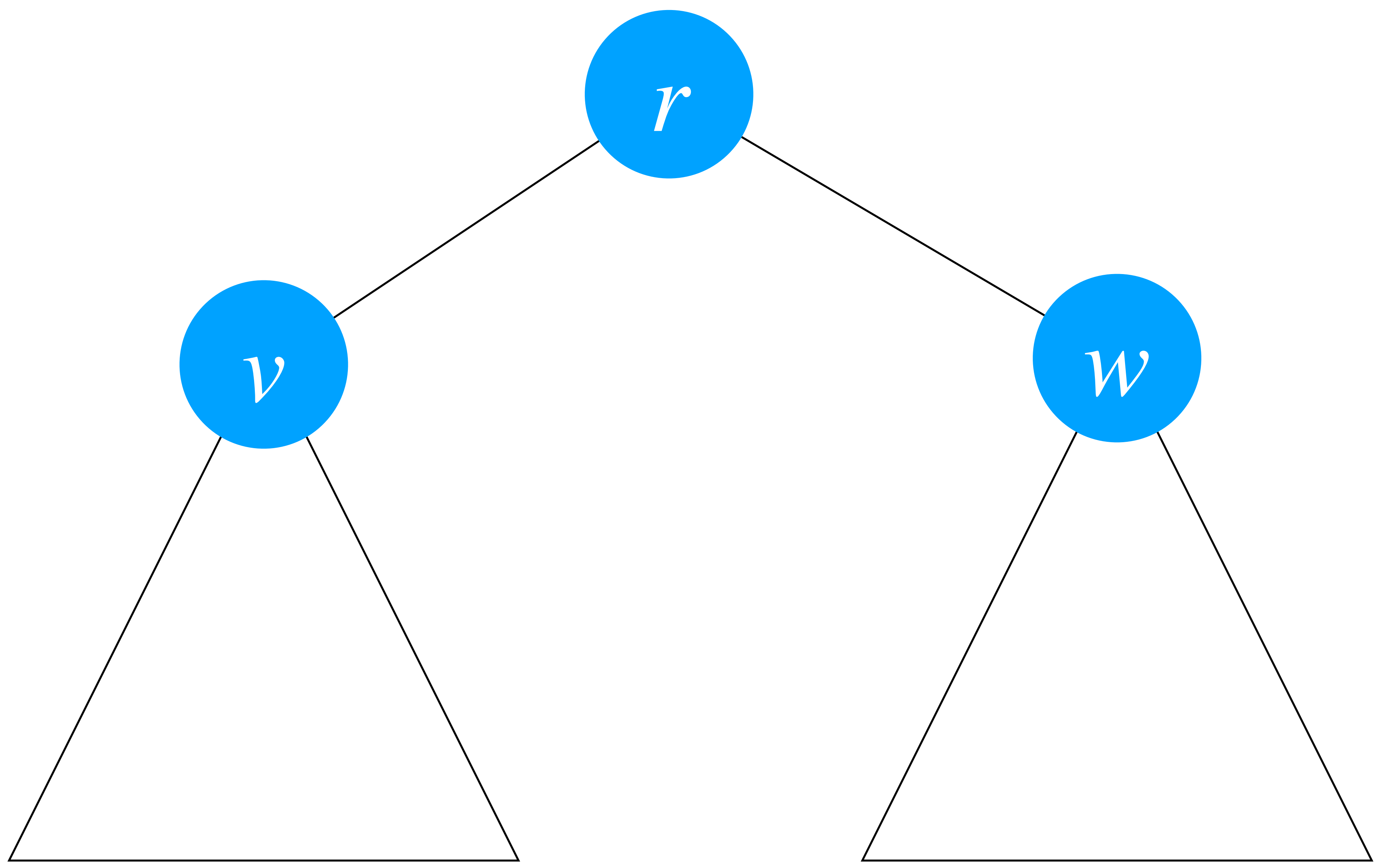
$n(u)$ max **IS** of the subtree **containing** u.



Independent Set on Trees

$n(u)$ max **IS** of the subtree **containing** u.

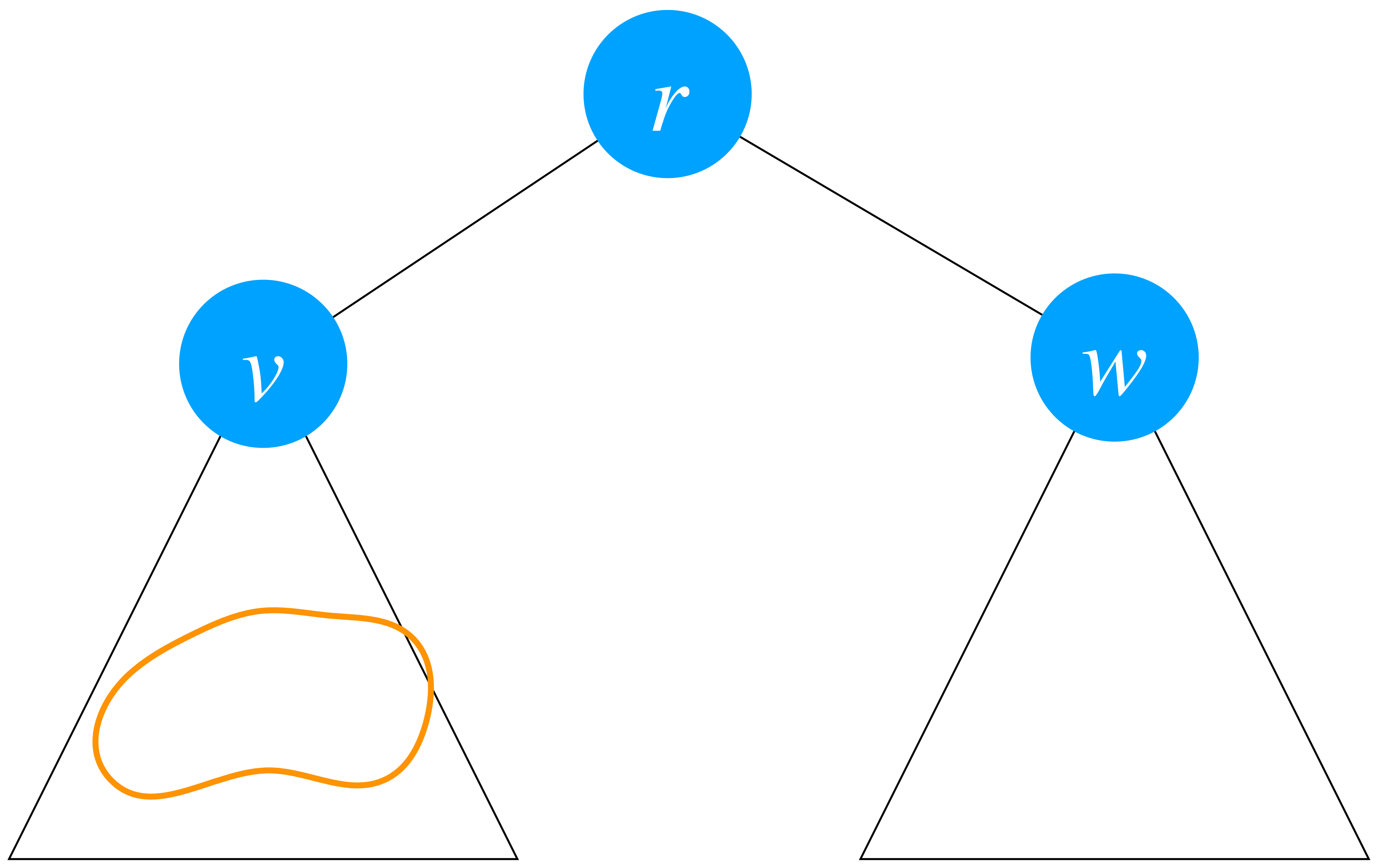
$n'(u)$ max **IS** of the subtree **not containing** u.



Independent Set on Trees

$n(u)$ max **IS** of the subtree **containing** u.

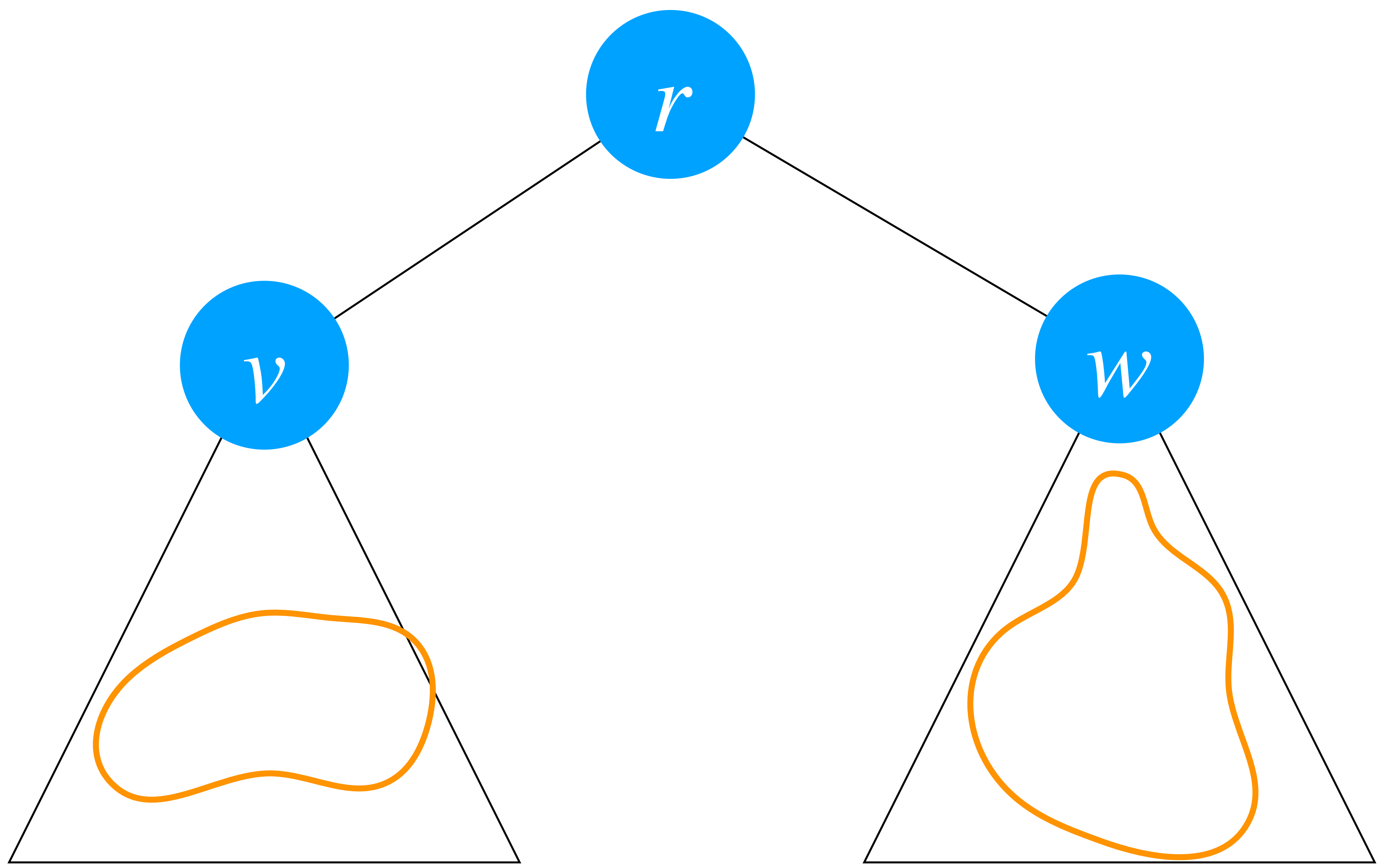
$n'(u)$ max **IS** of the subtree **not containing** u.



Independent Set on Trees

$n(u)$ max **IS** of the subtree **containing** u.

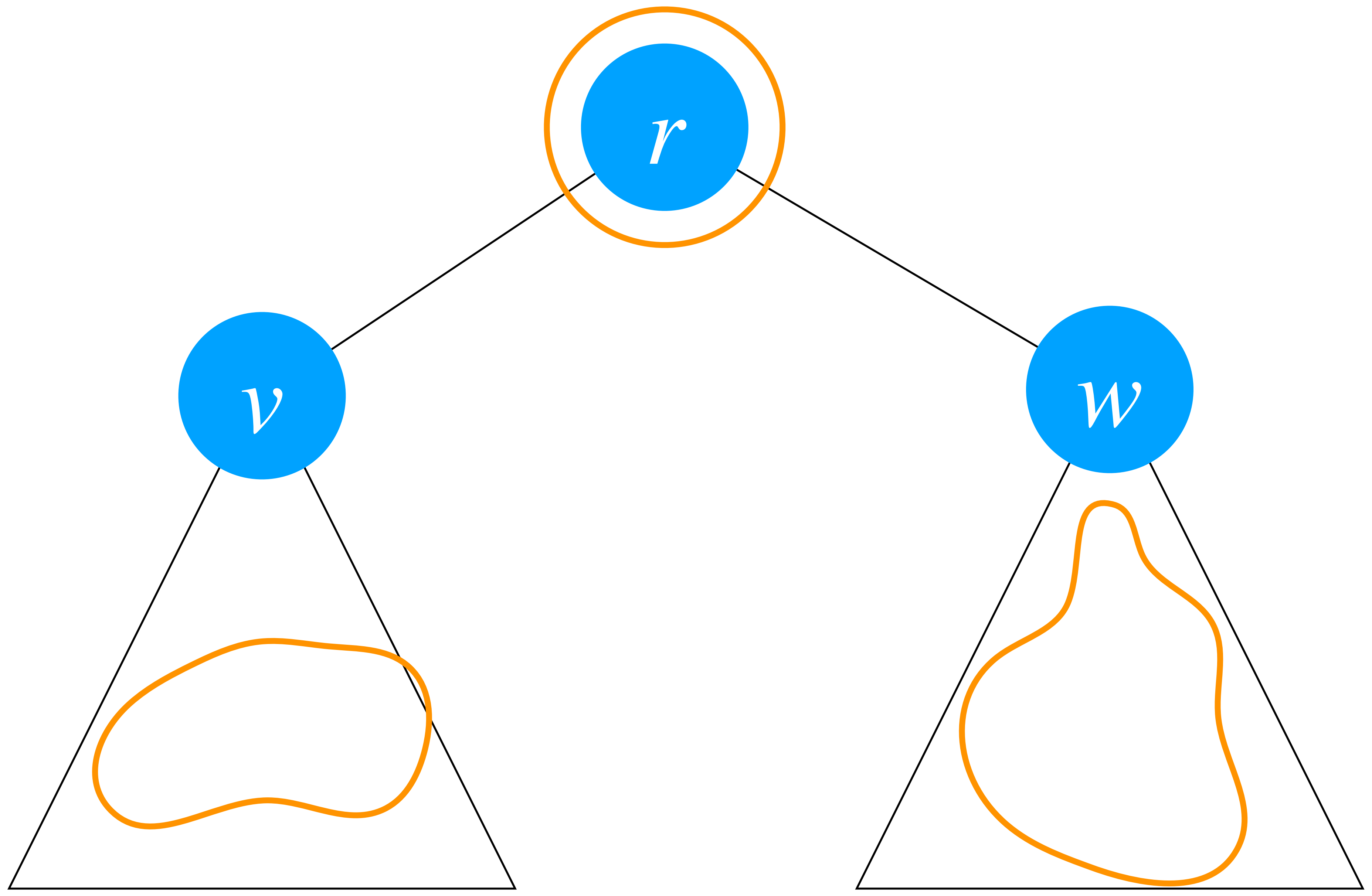
$n'(u)$ max **IS** of the subtree **not containing** u.



Independent Set on Trees

$n(u)$ max **IS** of the subtree **containing** u.

$n'(u)$ max **IS** of the subtree **not containing** u.

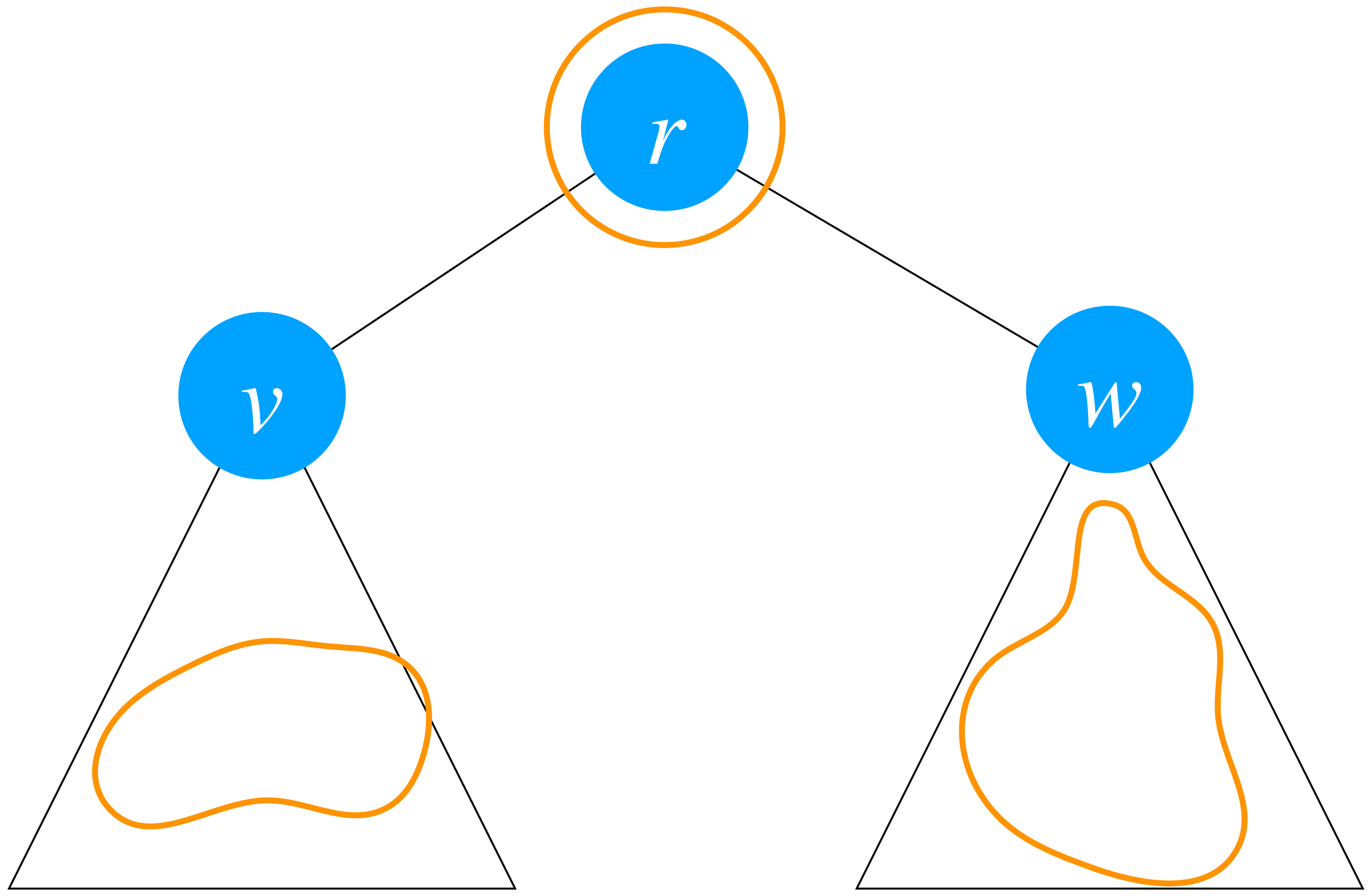


Independent Set on Trees

$n(u)$ max **IS** of the subtree **containing** u.

$n'(u)$ max **IS** of the subtree **not containing** u.

$n(r) = n'(v) + n'(w) + 1$

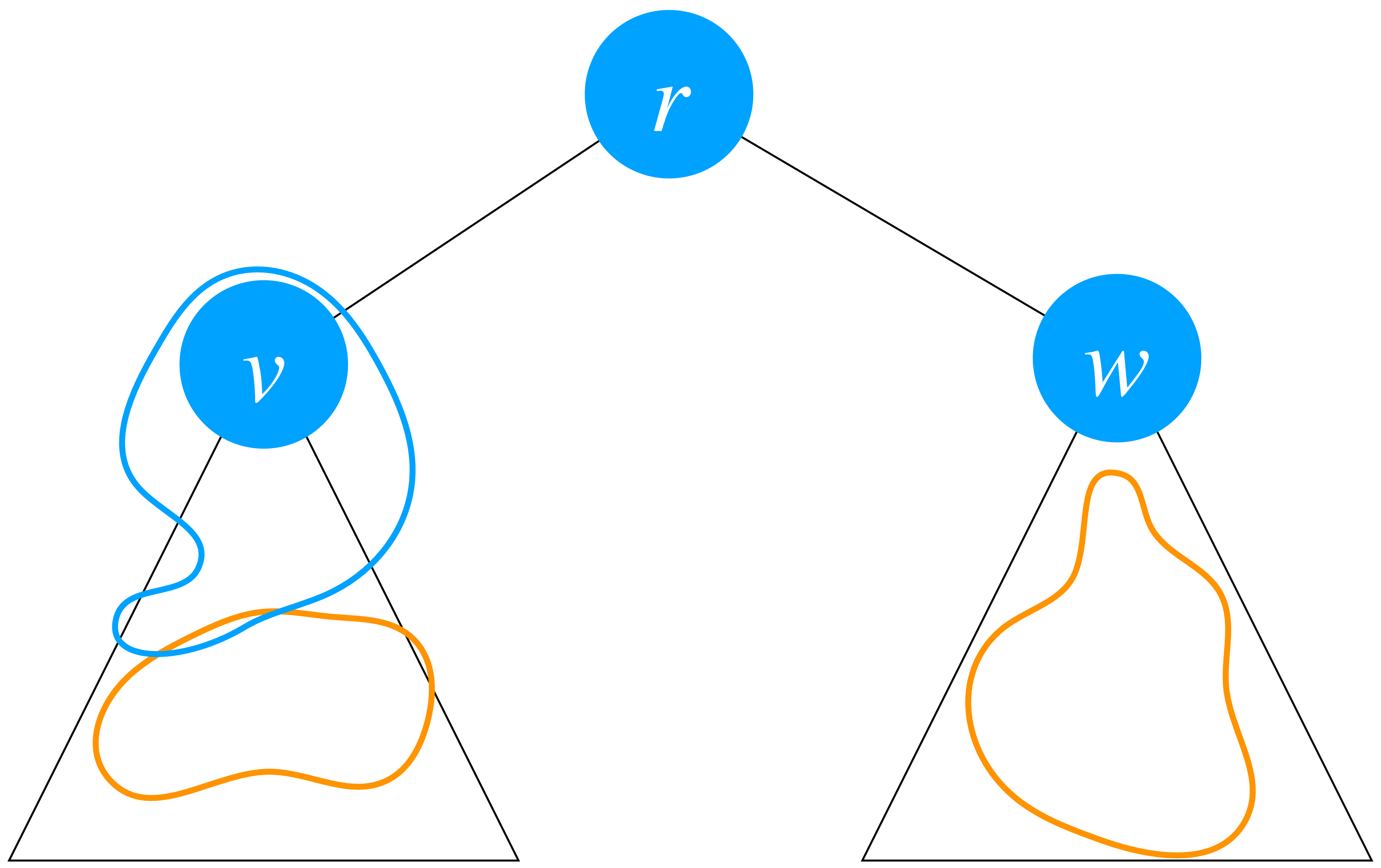


Independent Set on Trees

$n(u)$ max **IS** of the subtree **containing** u.

$n'(u)$ max **IS** of the subtree **not containing** u.

$n(r) = n'(v) + n'(w) + 1$

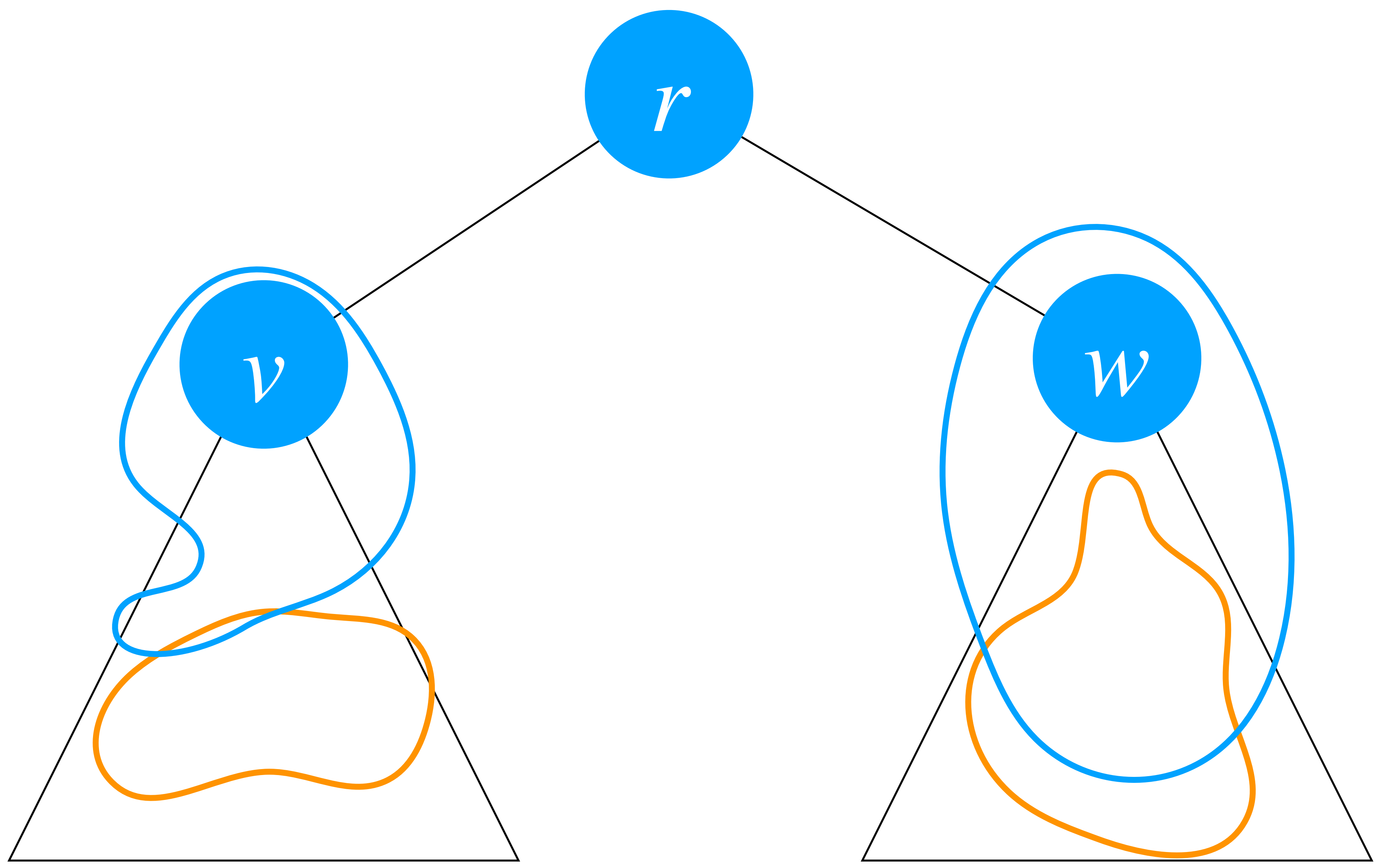


Independent Set on Trees

$n(u)$ max **IS** of the subtree **containing** u.

$n'(u)$ max **IS** of the subtree **not containing** u.

$n(r) = n'(v) + n'(w) + 1$



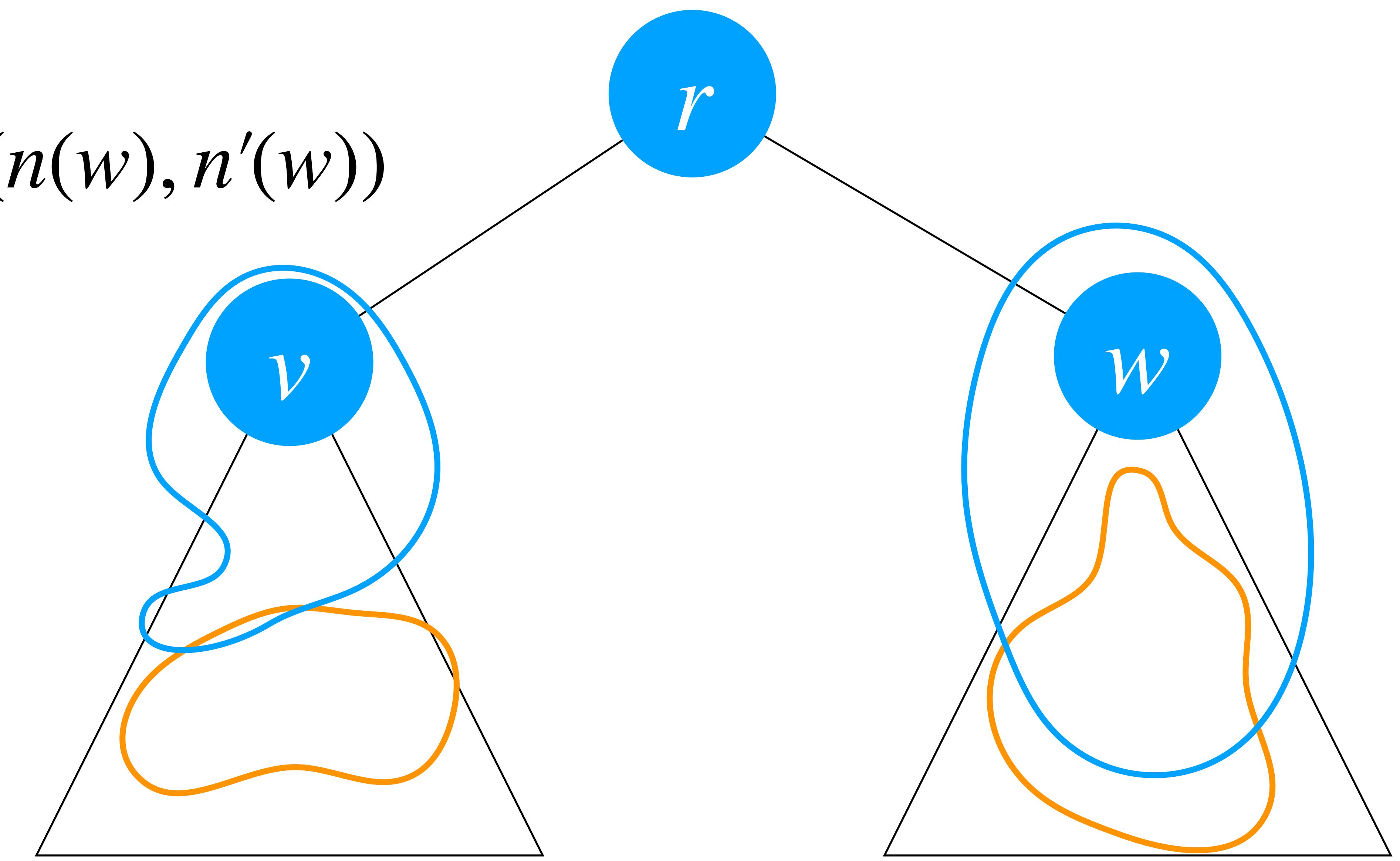
Independent Set on Trees

$n(u)$ max **IS** of the subtree **containing** u.

$n'(u)$ max **IS** of the subtree **not containing** u.

$n(r) = n'(v) + n'(w) + 1$

$n'(r) = \max(n(v), n'(v)) \ + \max((n(w), n'(w))$



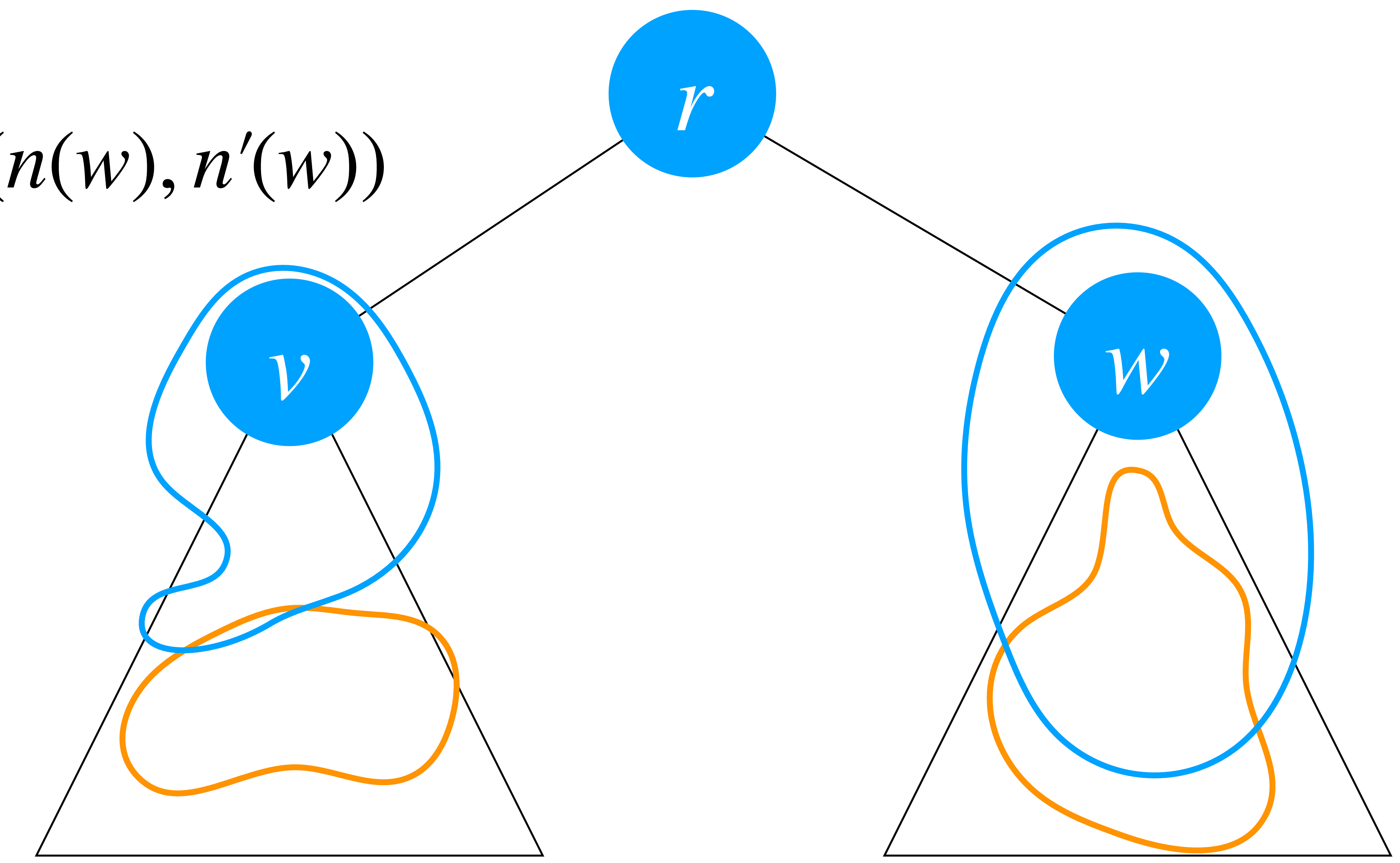
Independent Set on Trees

$n(u)$ max **IS** of the subtree **containing** u.

$n'(u)$ max **IS** of the subtree **not containing** u.

$n(r) = n'(v) + n'(w) + 1$

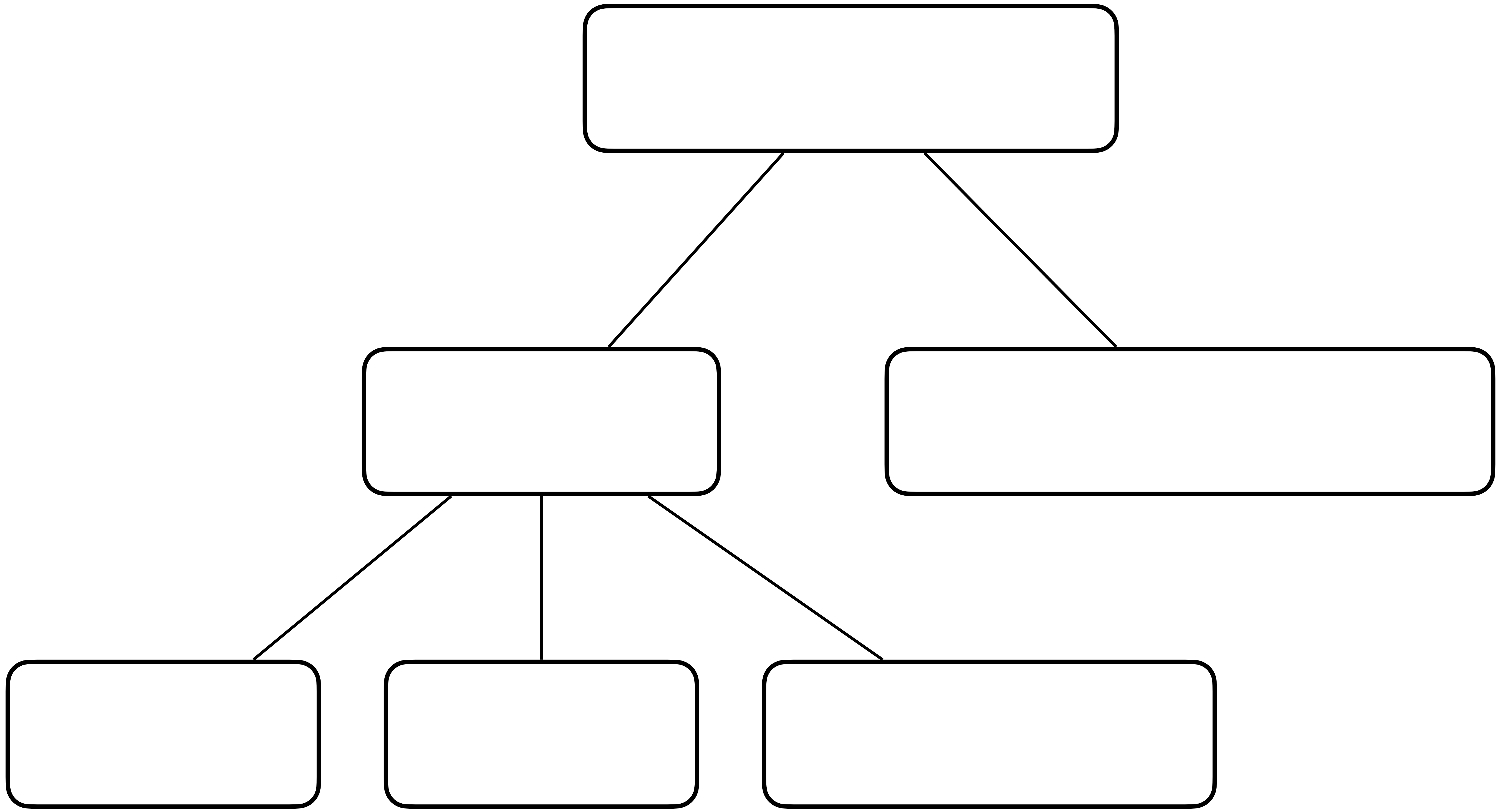
$n'(r) = \max(n(v), n'(v)) + \max((n(w), n'(w))$



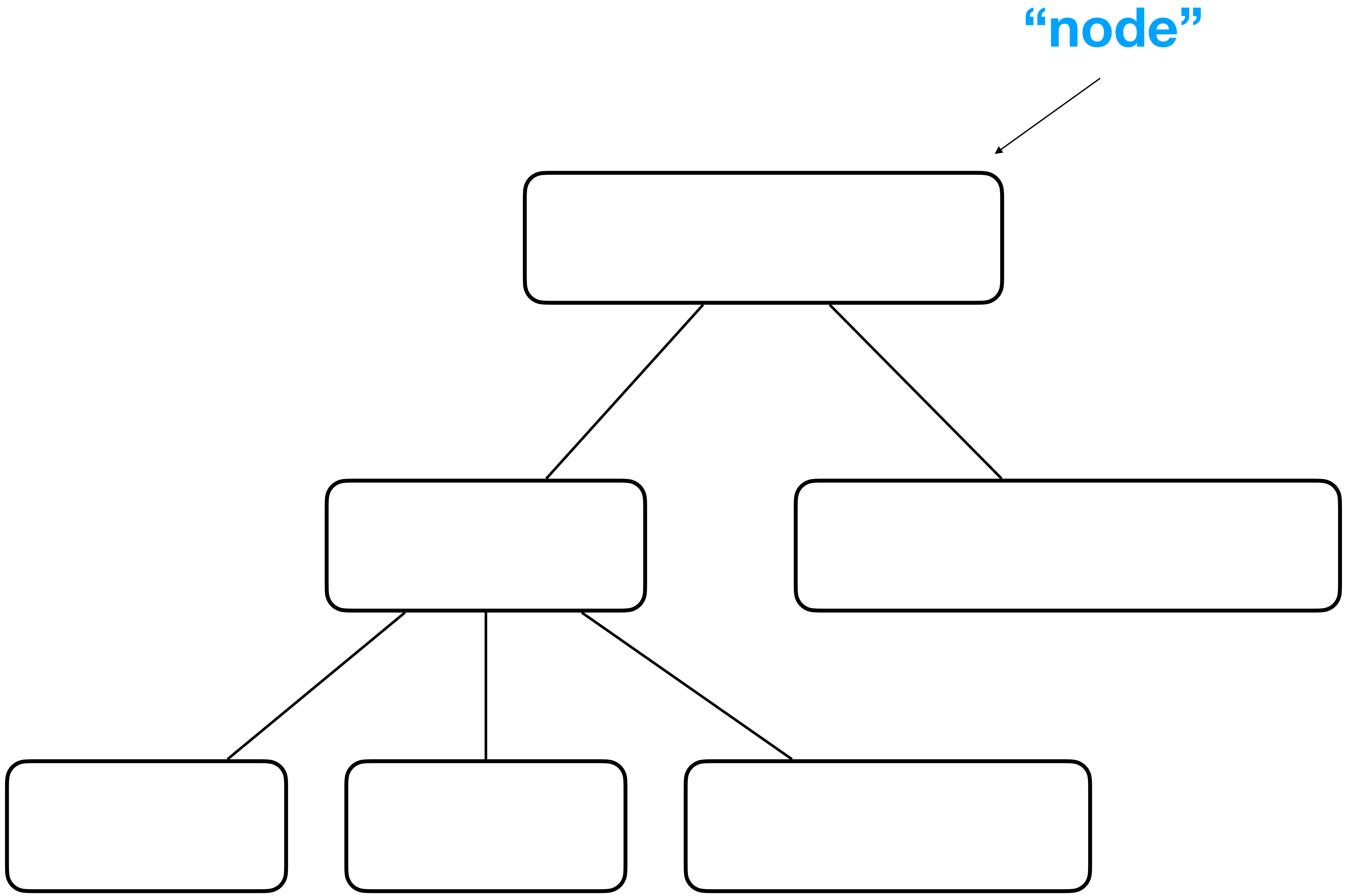
$\max(n(r), n'(r))$ is the size of the largest **IS**.

Tree Decomposition

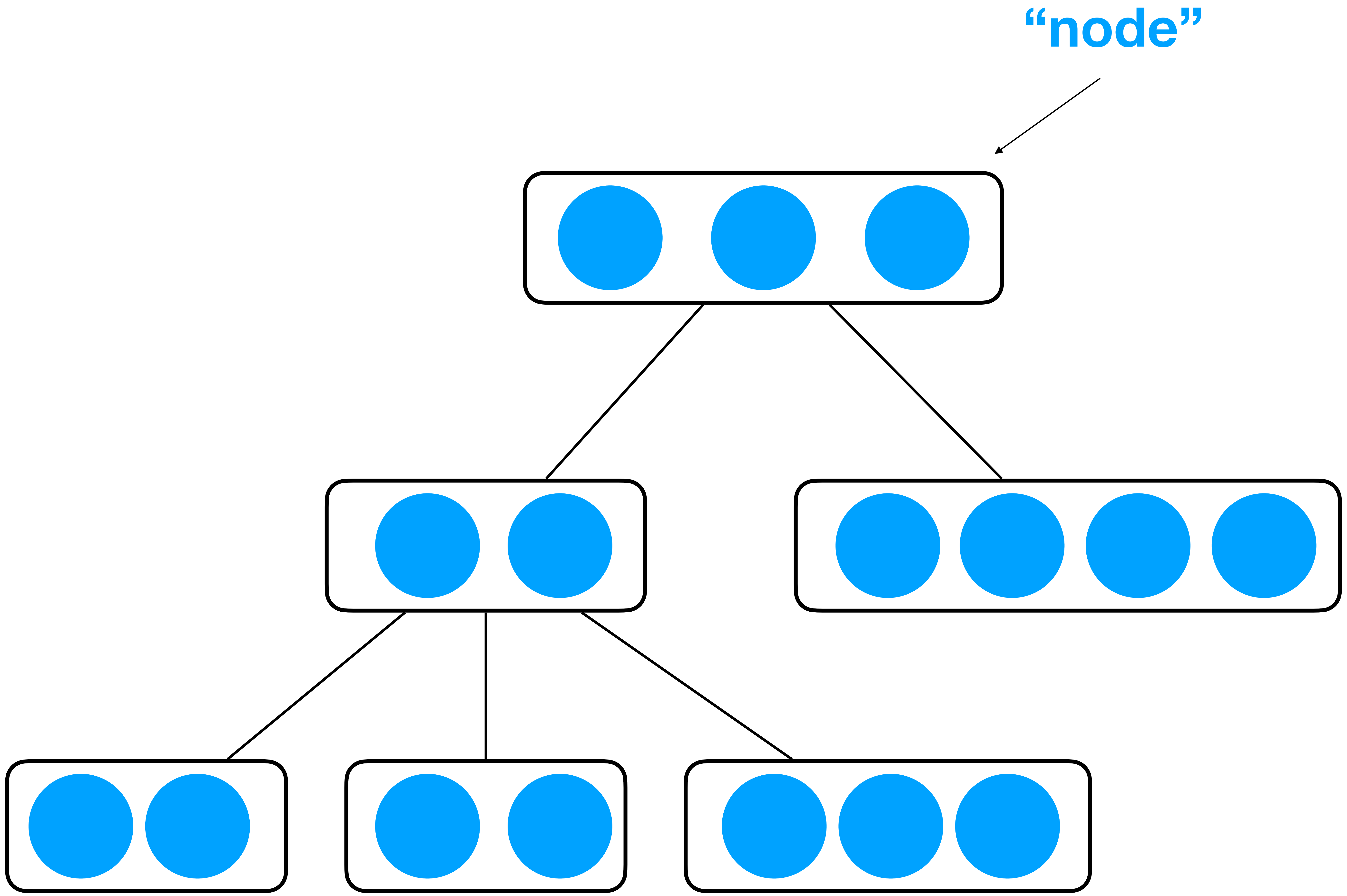
Tree Decomposition



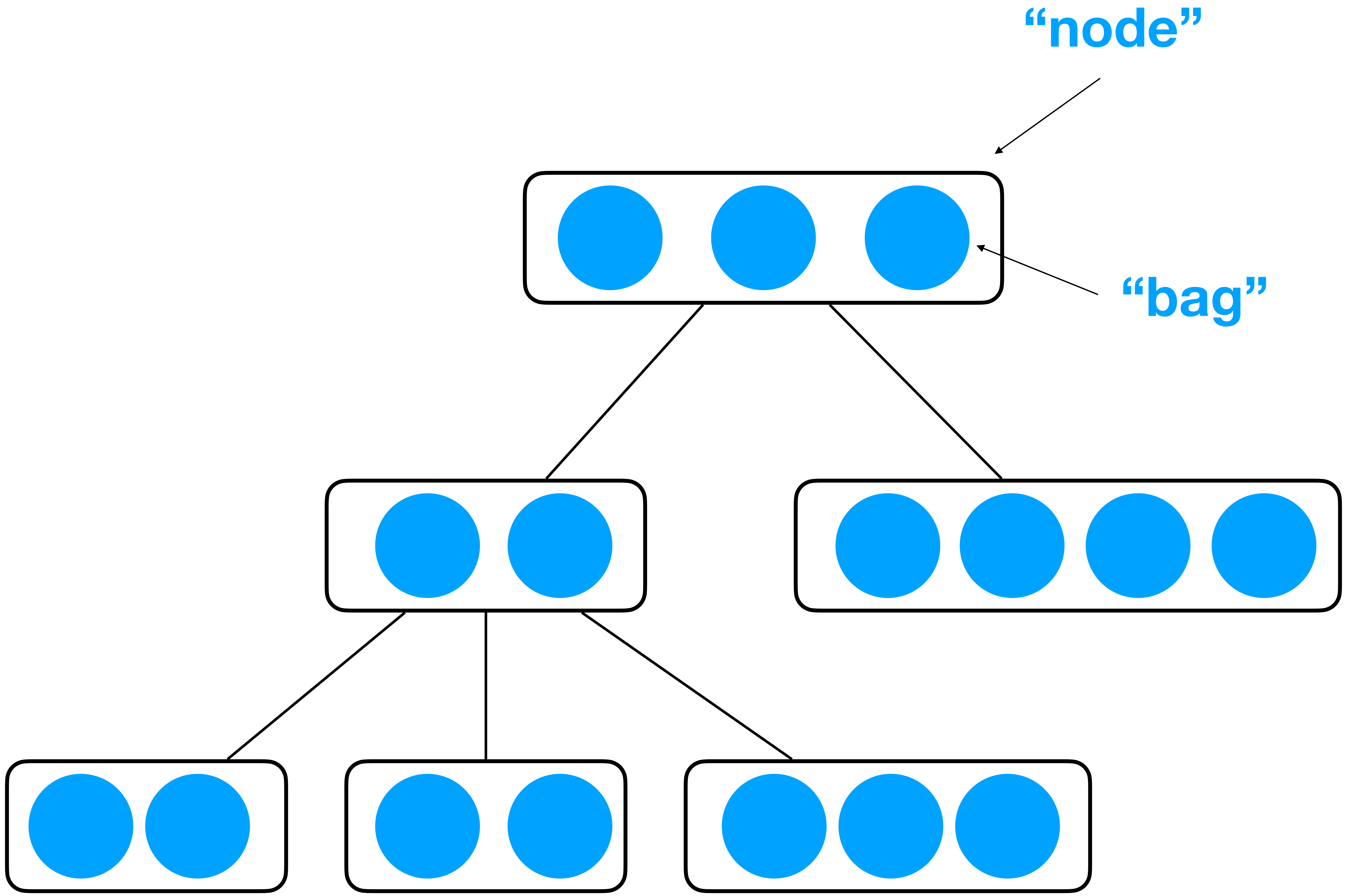
Tree Decomposition



Tree Decomposition

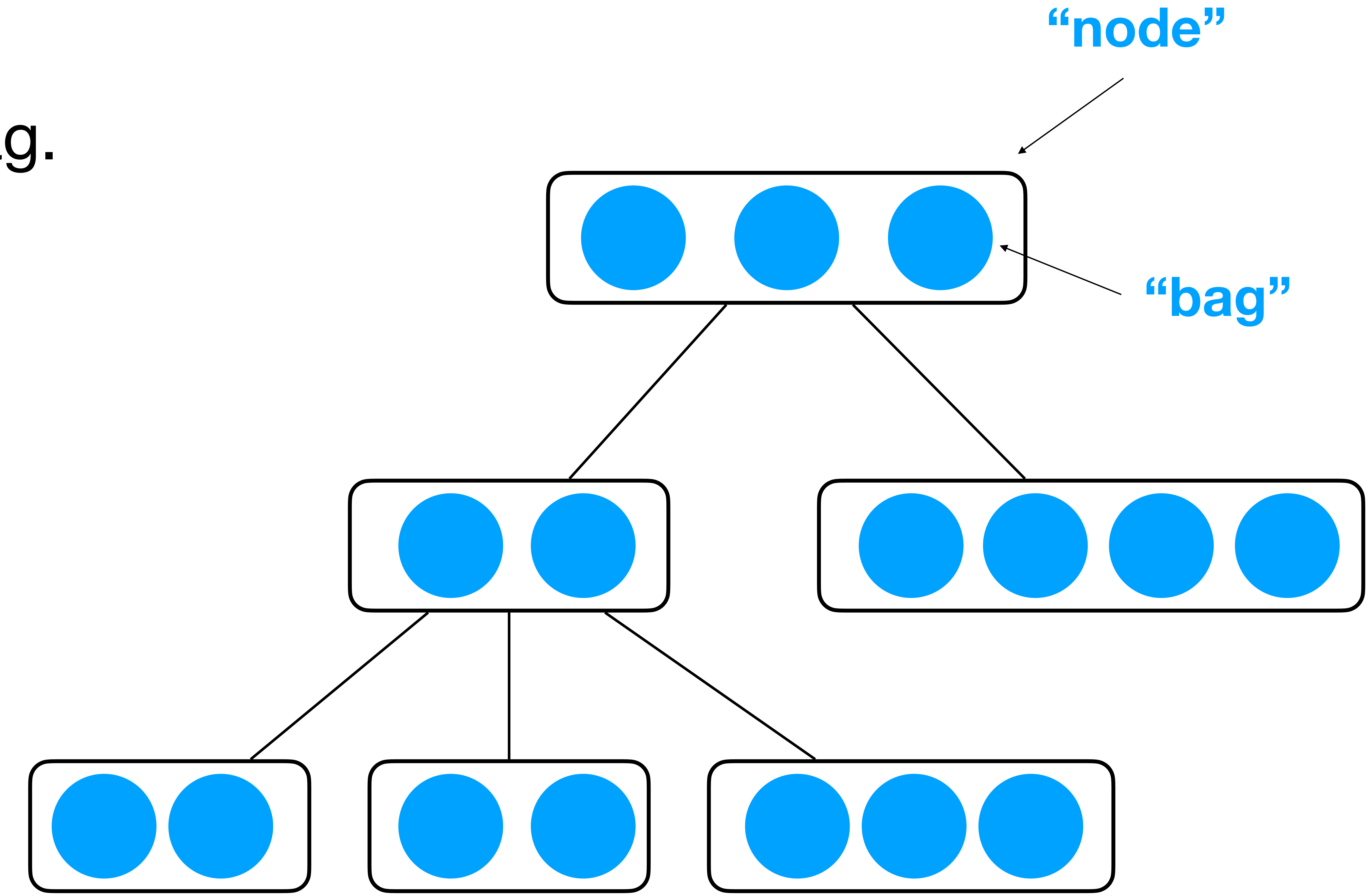


Tree Decomposition



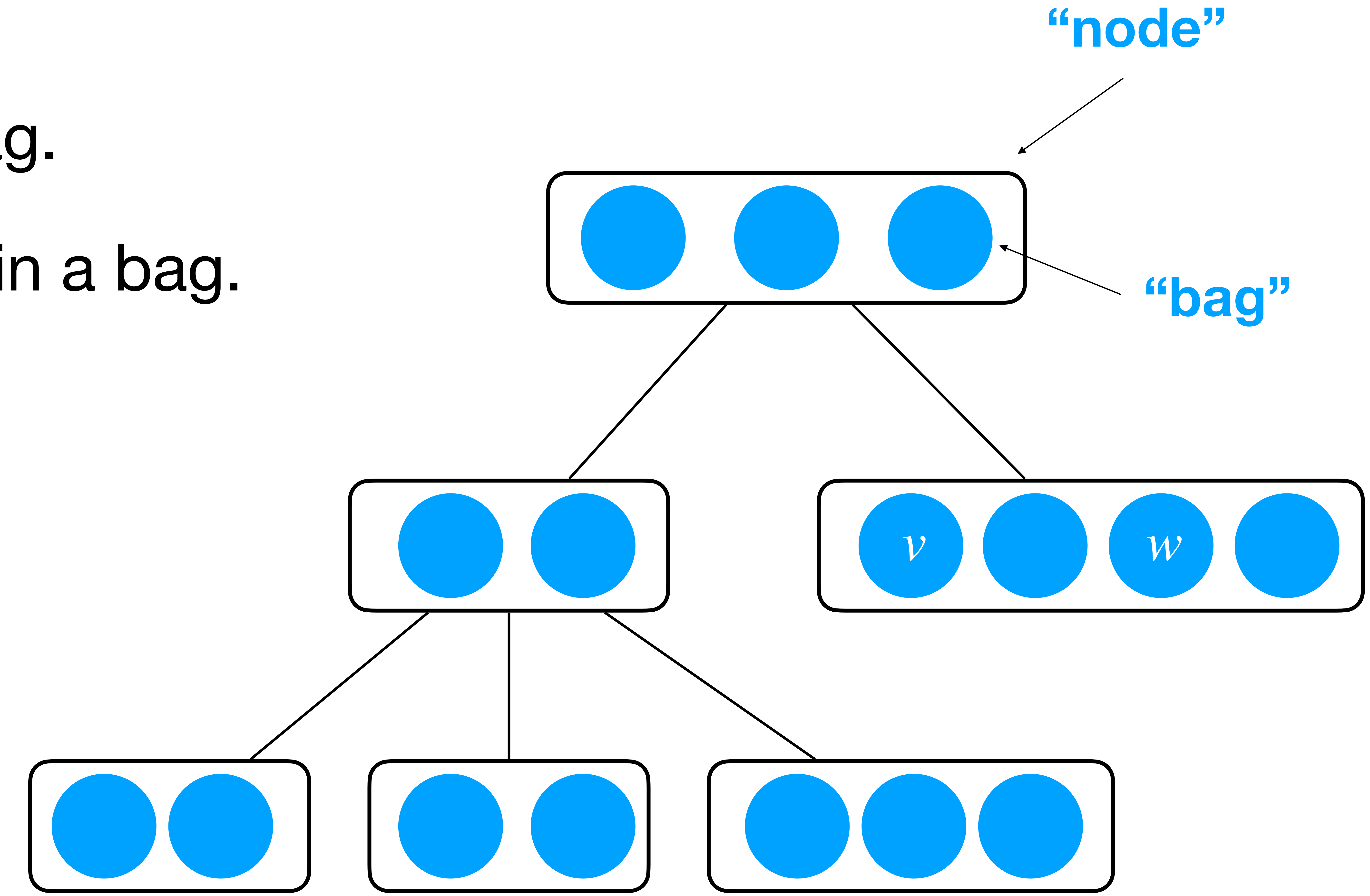
Tree Decomposition

1. Each **vertex** appears in a bag.



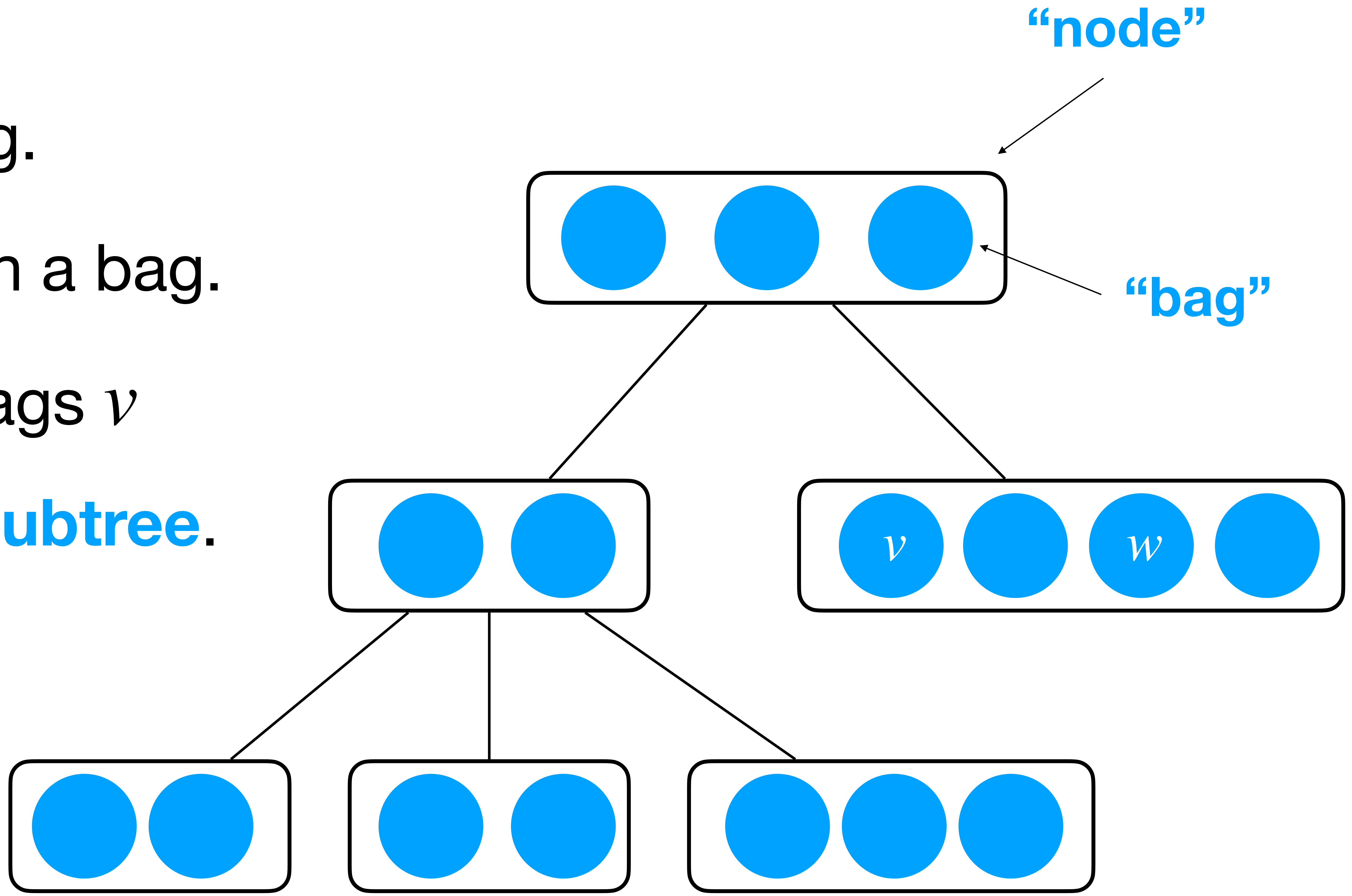
Tree Decomposition

1. Each **vertex** appears in a bag.
2. Each **edge** vw is contained in a bag.



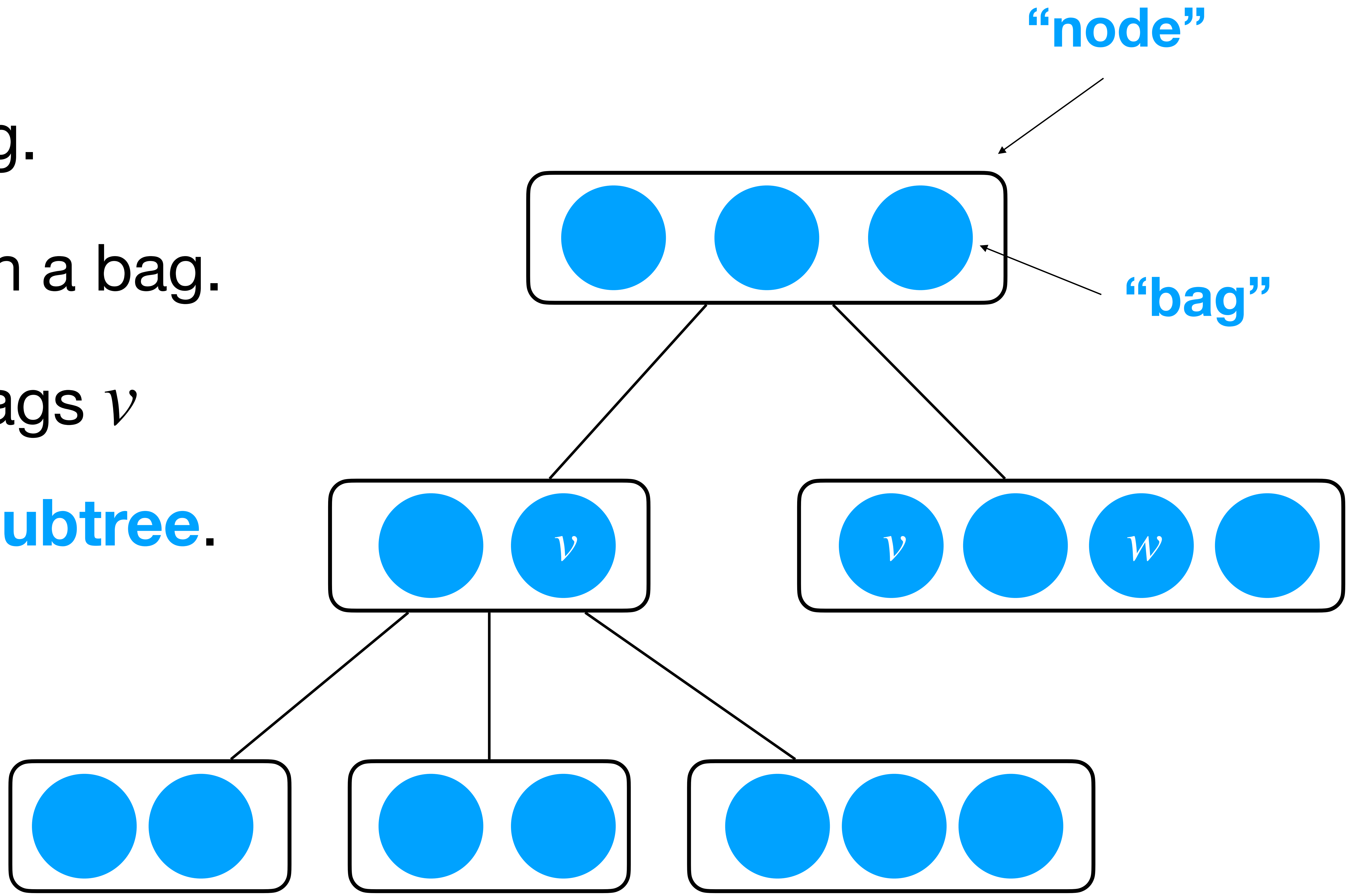
Tree Decomposition

1. Each **vertex** appears in a bag.
2. Each **edge** vw is contained in a bag.
3. The set of nodes in whose bags v appears form a **connected subtree**.



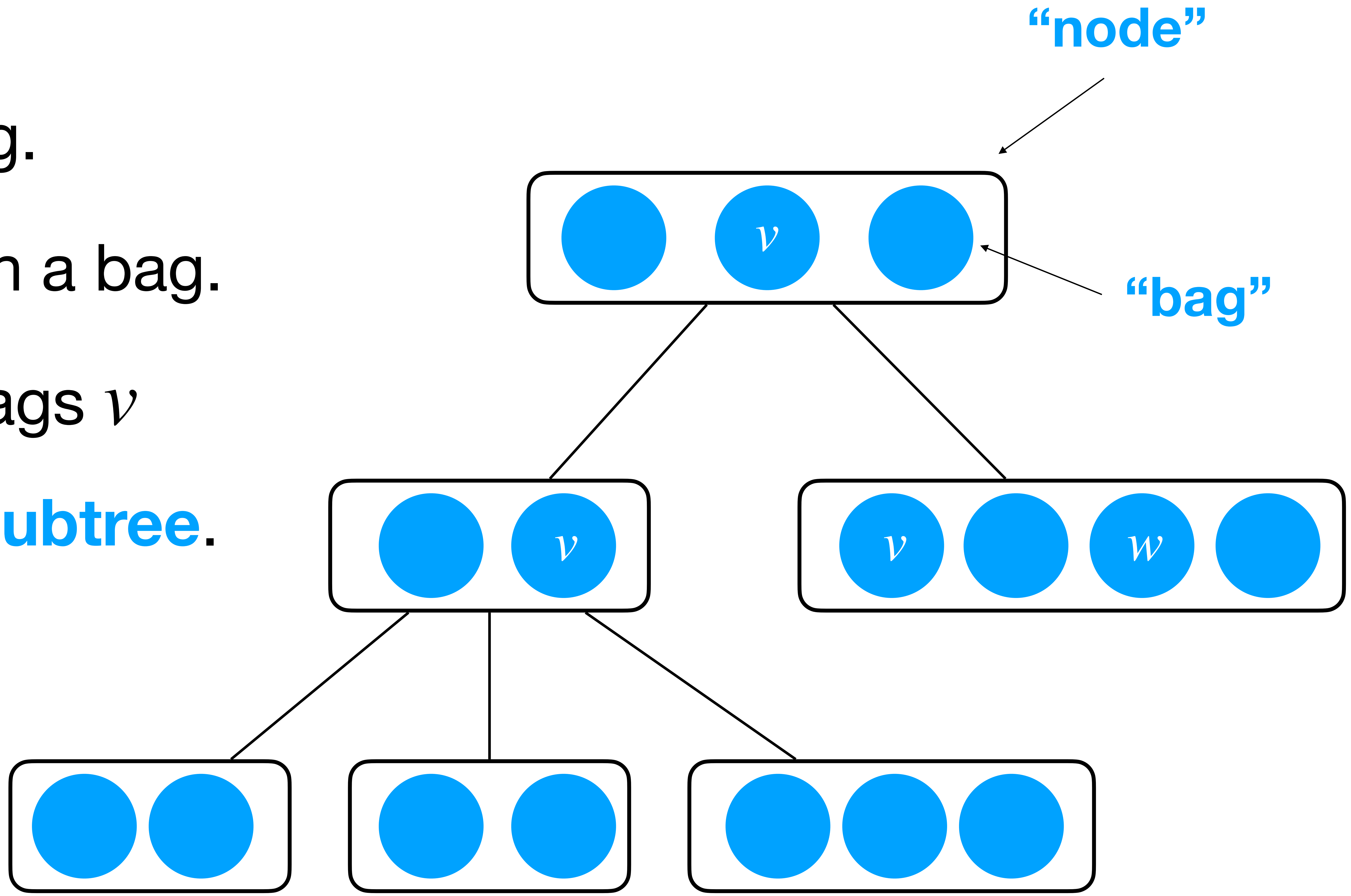
Tree Decomposition

1. Each **vertex** appears in a bag.
2. Each **edge** vw is contained in a bag.
3. The set of nodes in whose bags v appears form a **connected subtree**.



Tree Decomposition

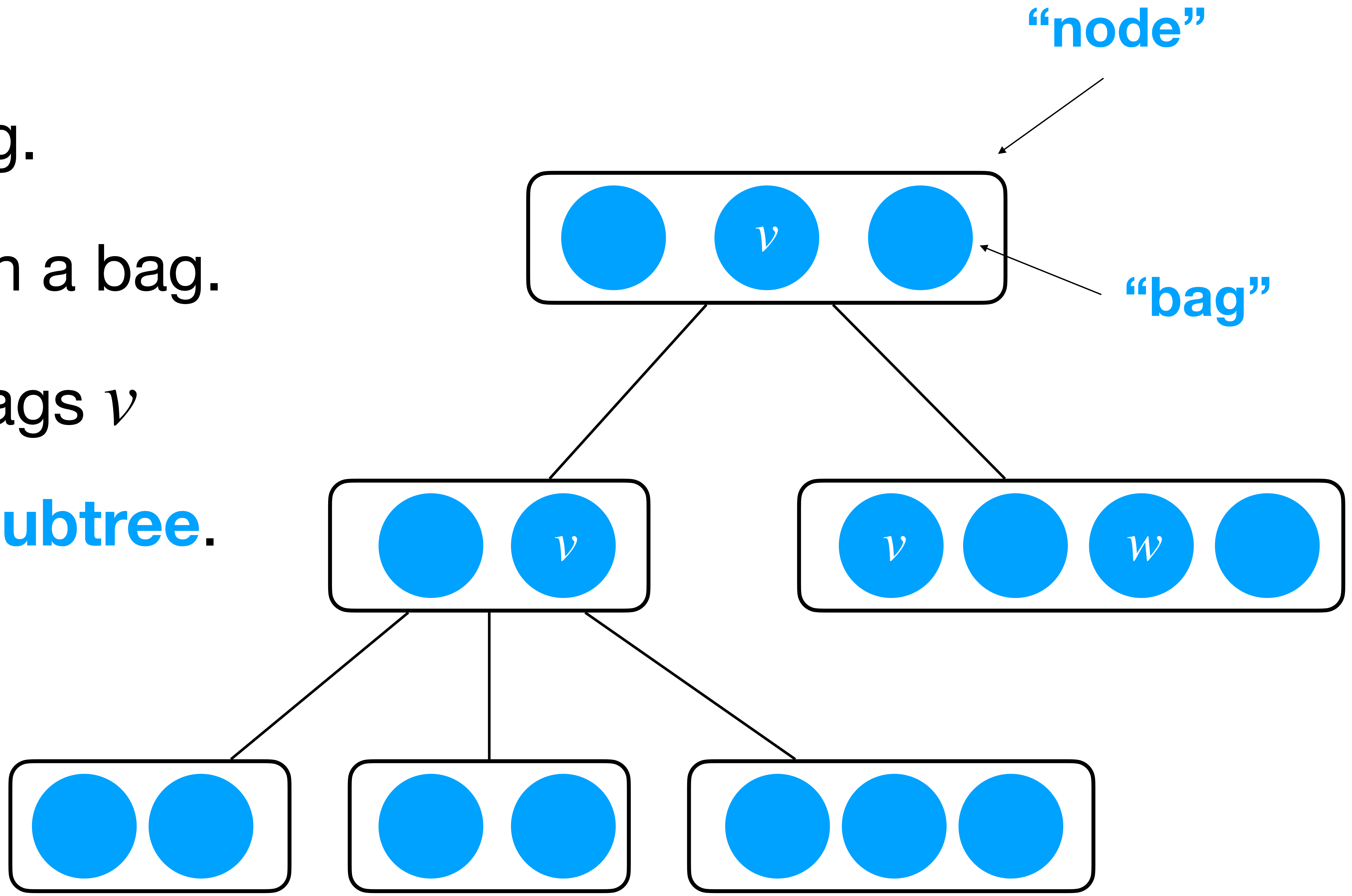
1. Each **vertex** appears in a bag.
2. Each **edge** vw is contained in a bag.
3. The set of nodes in whose bags v appears form a **connected subtree**.



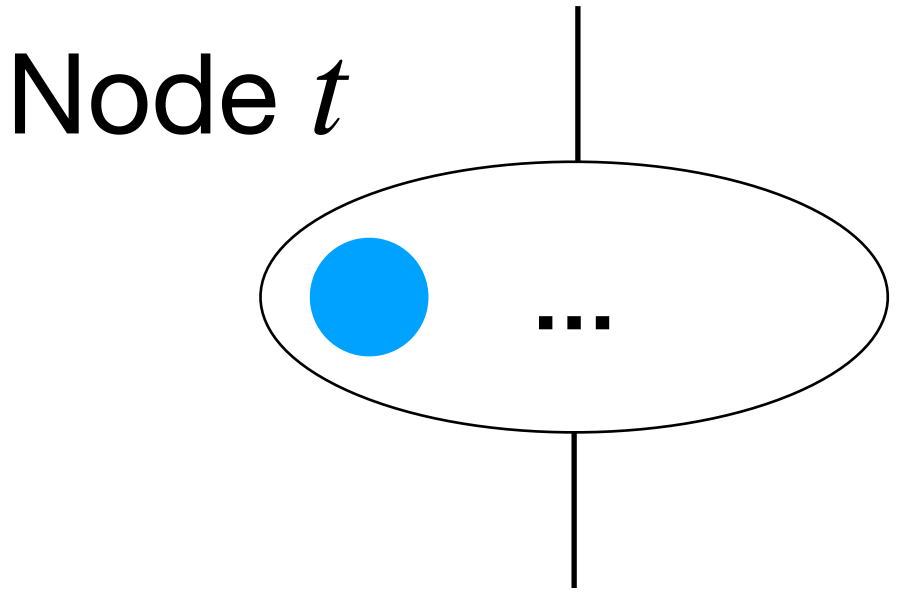
Tree Decomposition

1. Each **vertex** appears in a bag.
2. Each **edge** vw is contained in a bag.
3. The set of nodes in whose bags v appears form a **connected subtree**.

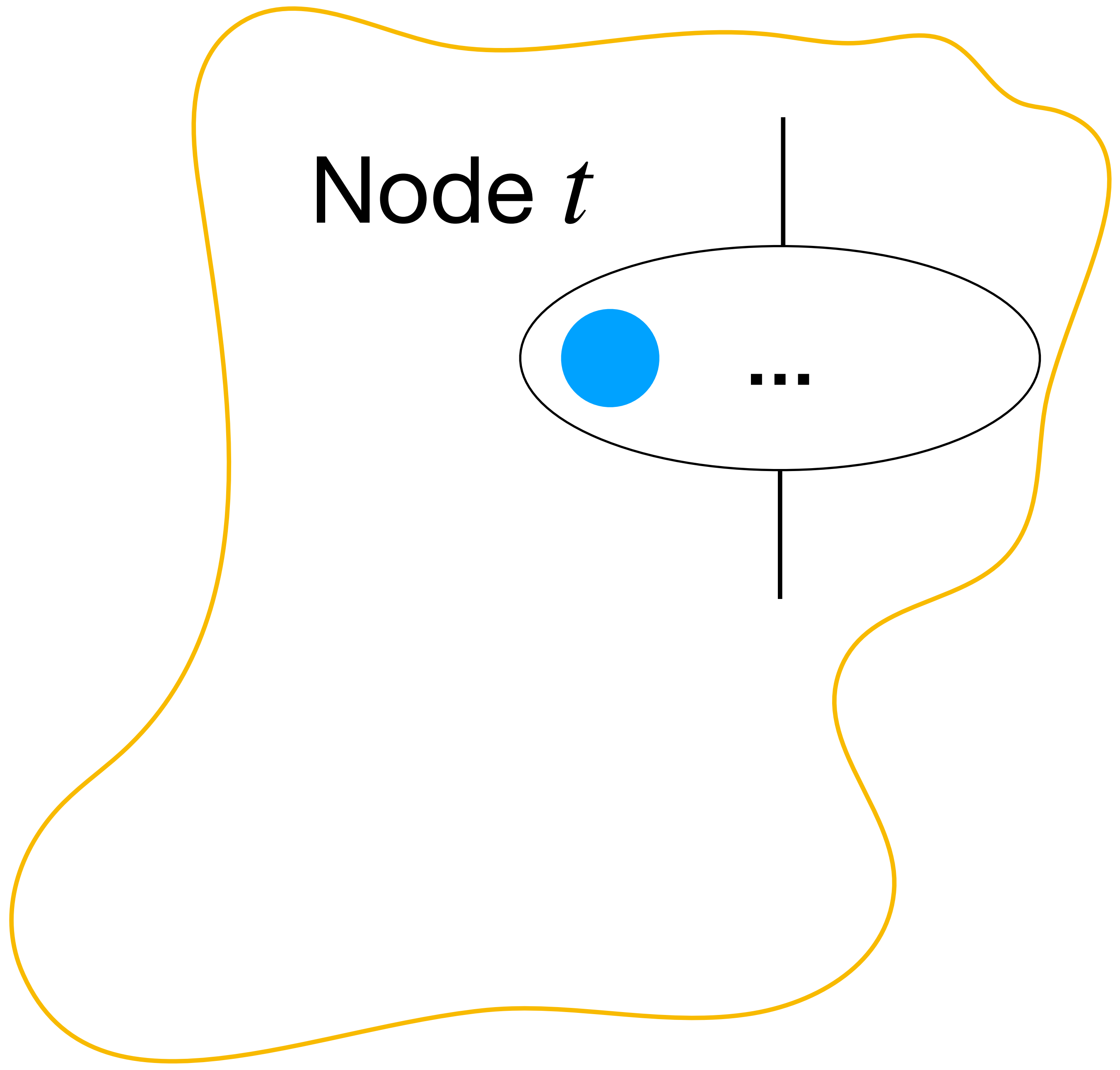
The **width** of a tree decomposition is the size of its largest bag - 1.



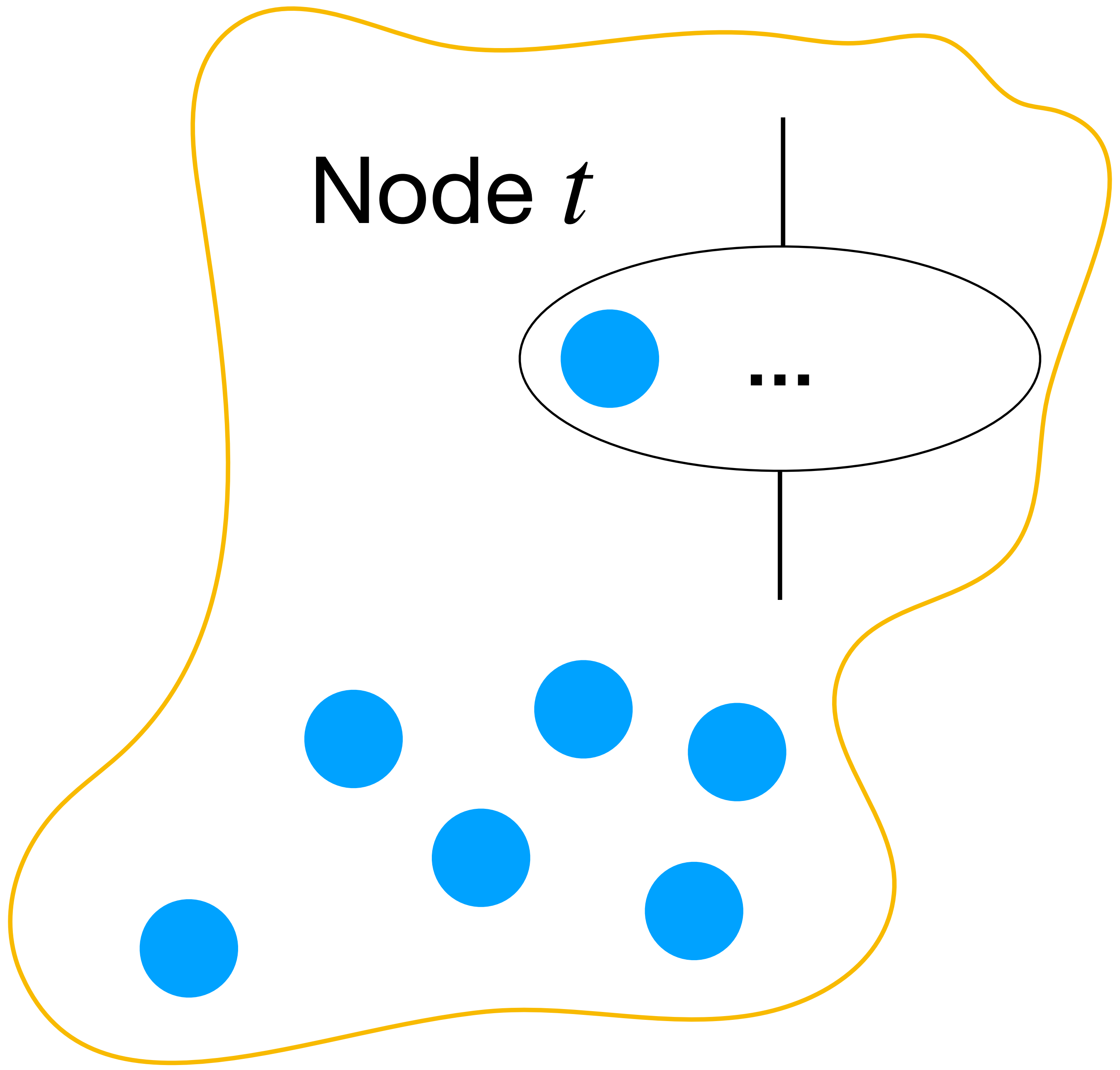
Independent Set with Tree Decompositions



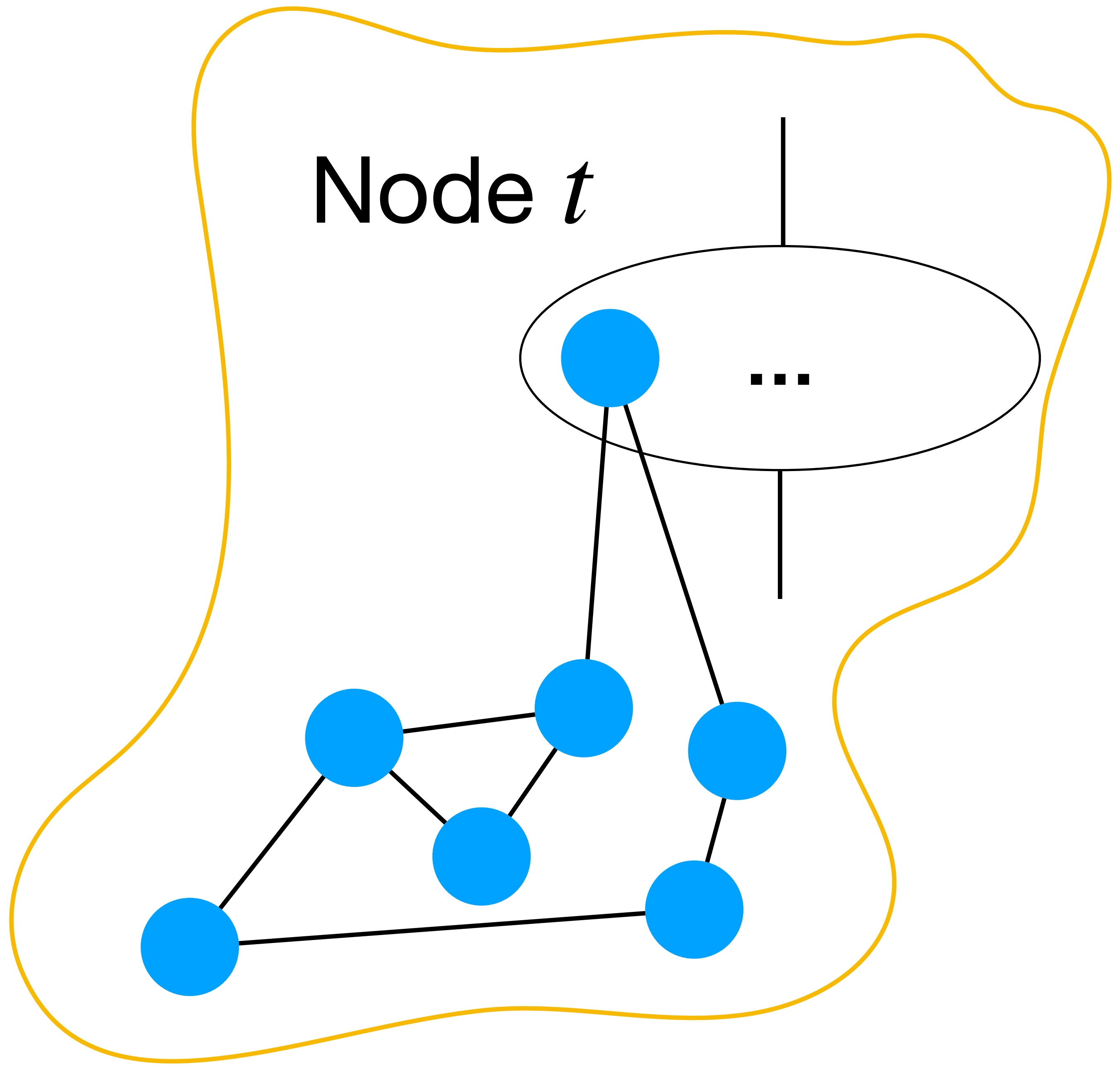
Independent Set with Tree Decompositions



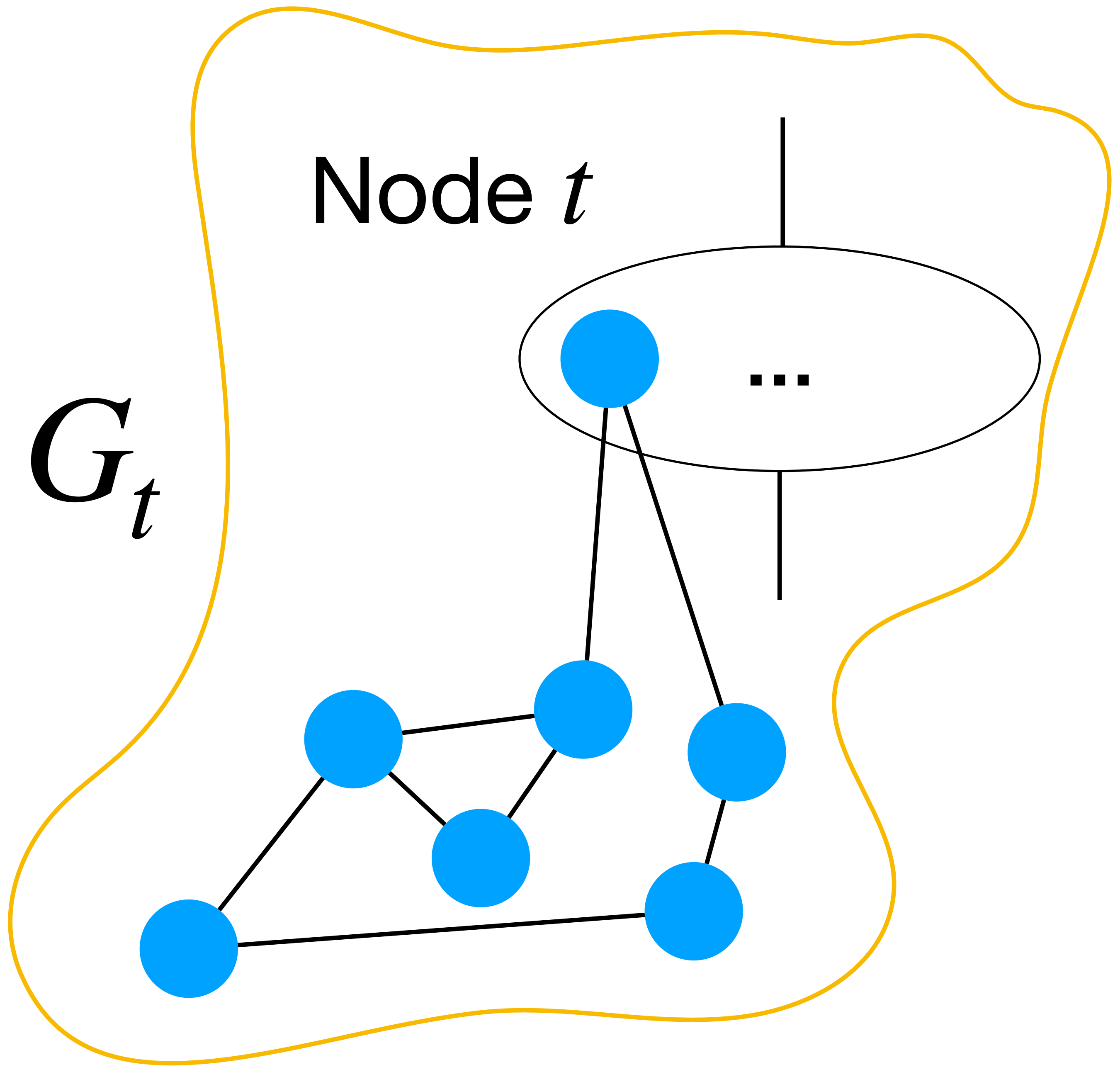
Independent Set with Tree Decompositions



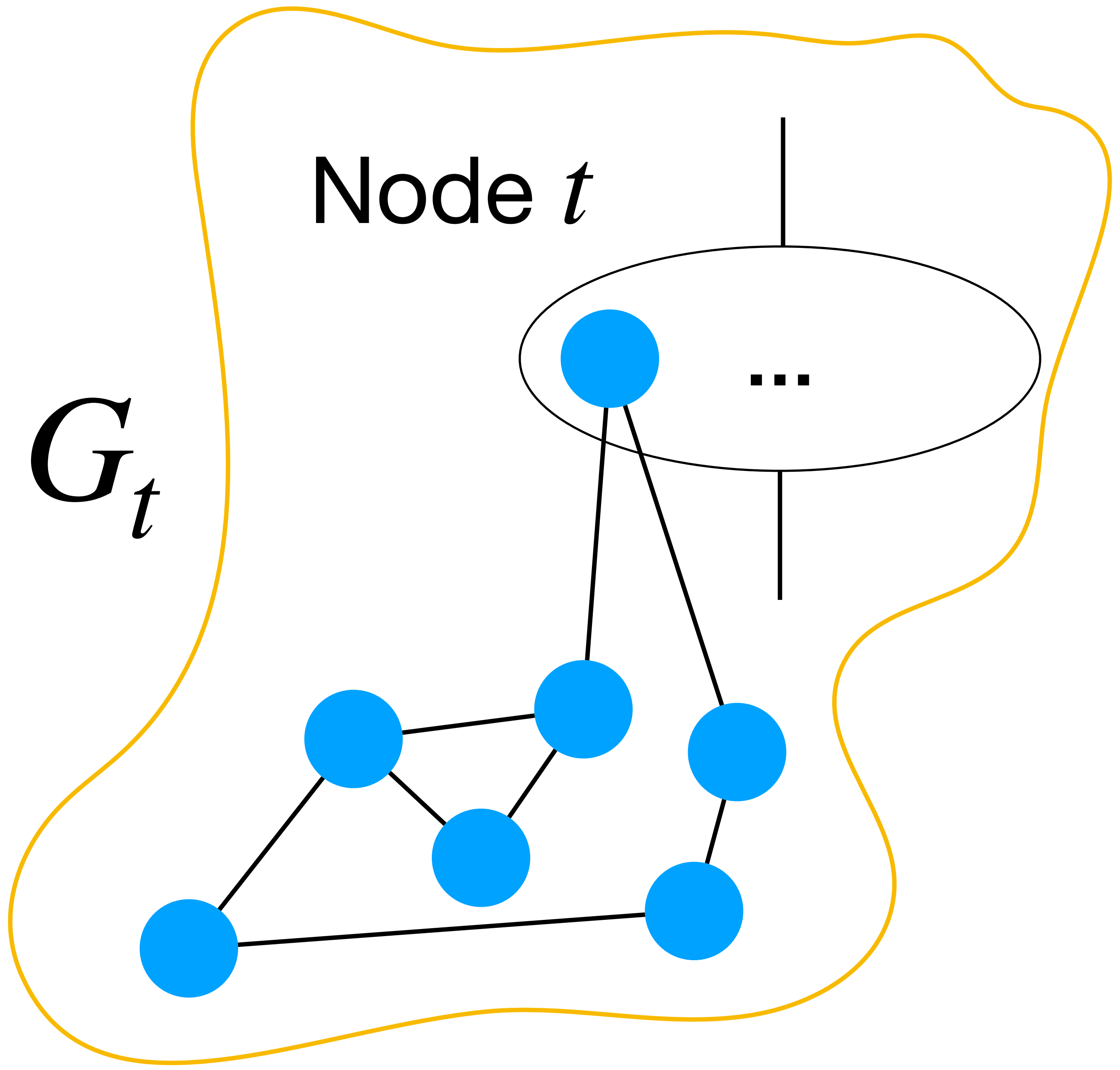
Independent Set with Tree Decompositions



Independent Set with Tree Decompositions

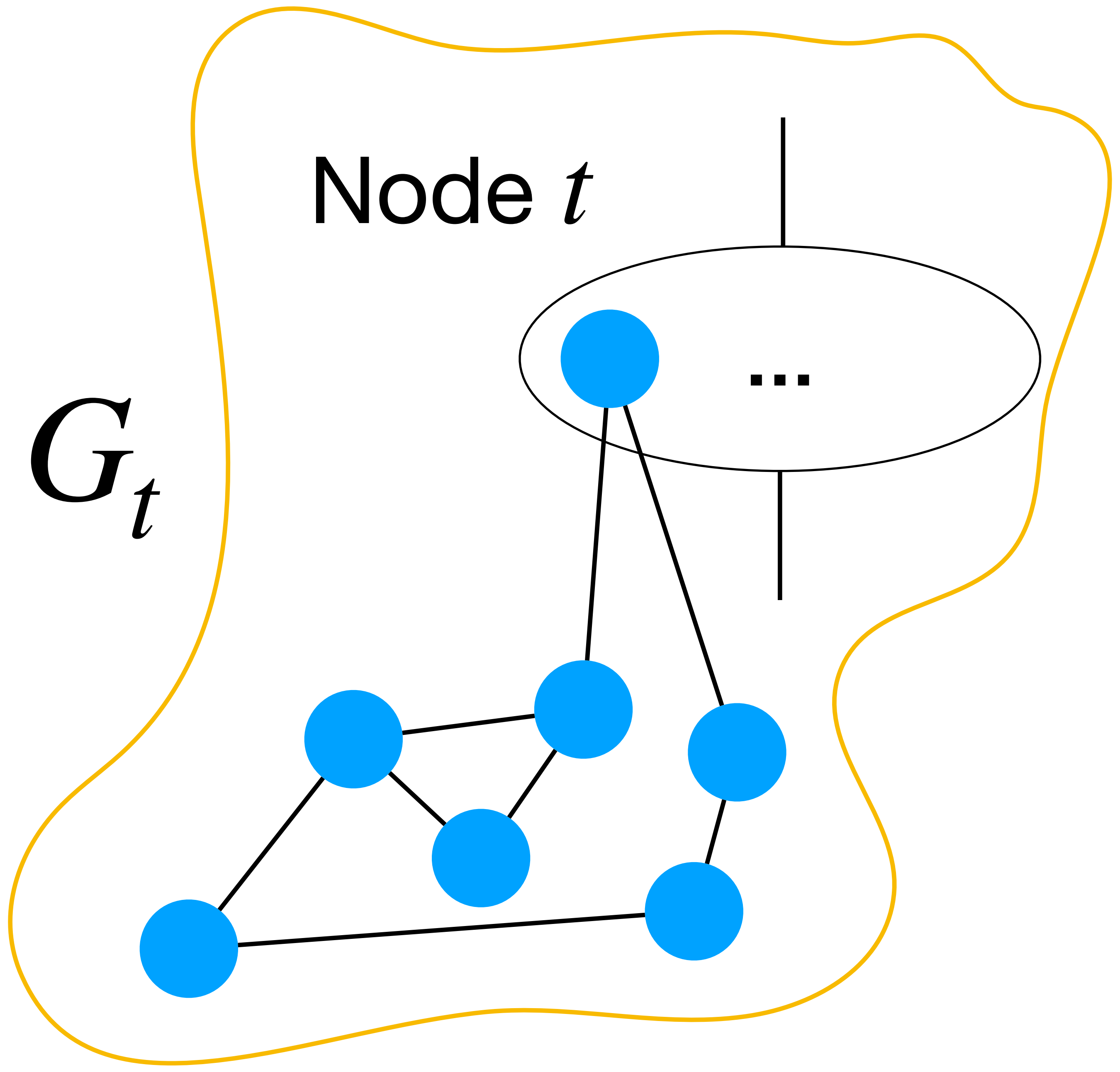


Independent Set with Tree Decompositions



For each subset $S \subseteq \chi(t)$ of bag vertices,
$n_t(S)$ = | max. **IS** of G_t containing S |.

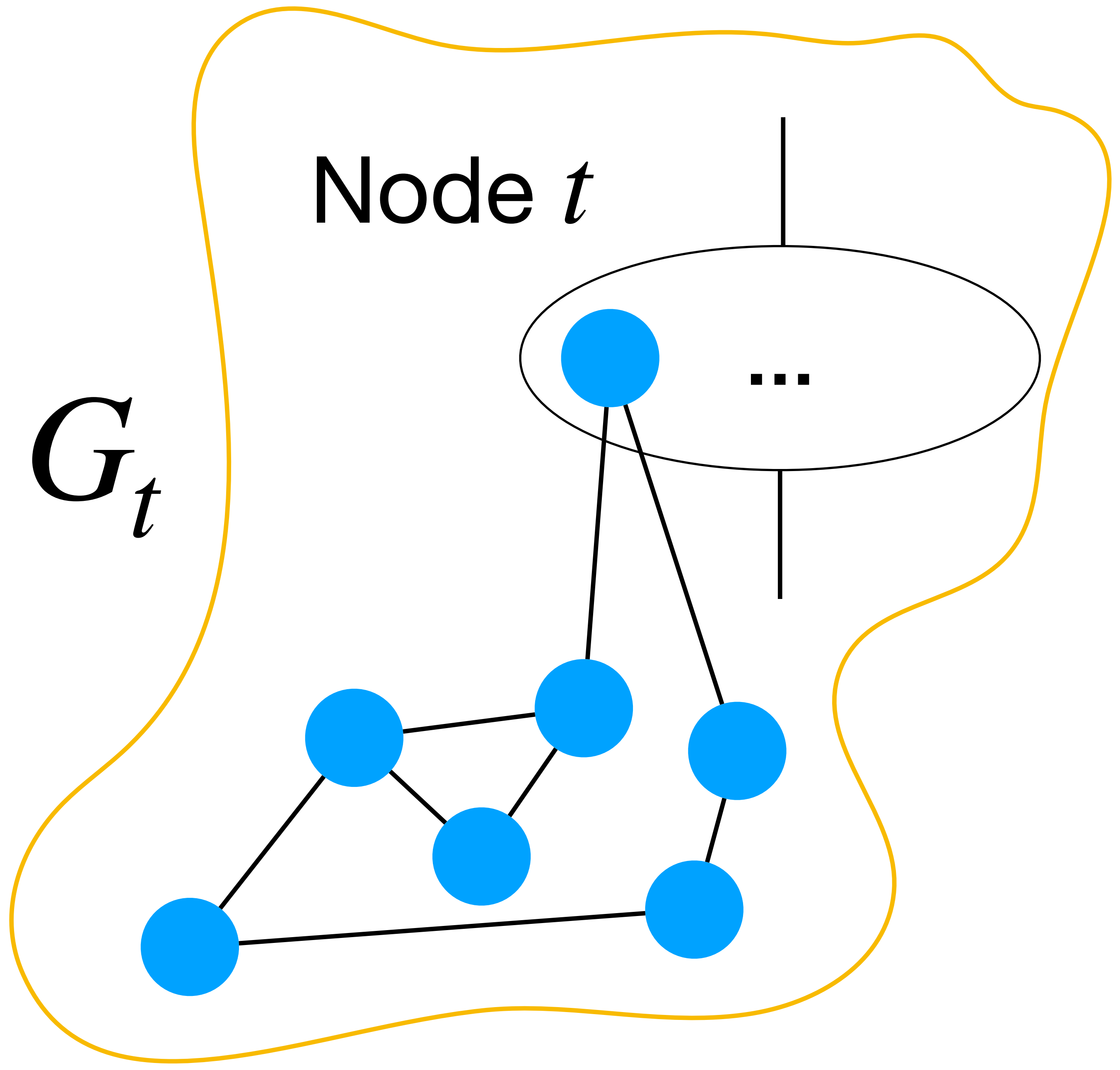
Independent Set with Tree Decompositions



Treewidth k

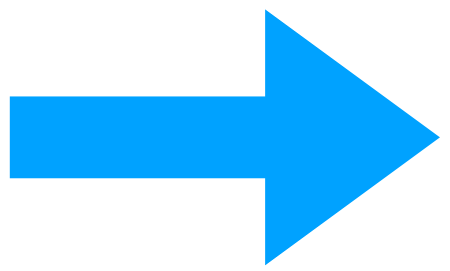
For each subset $S \subseteq \chi(t)$ of bag vertices,
$n_t(S)$ = | max. **IS** of G_t containing S |.

Independent Set with Tree Decompositions

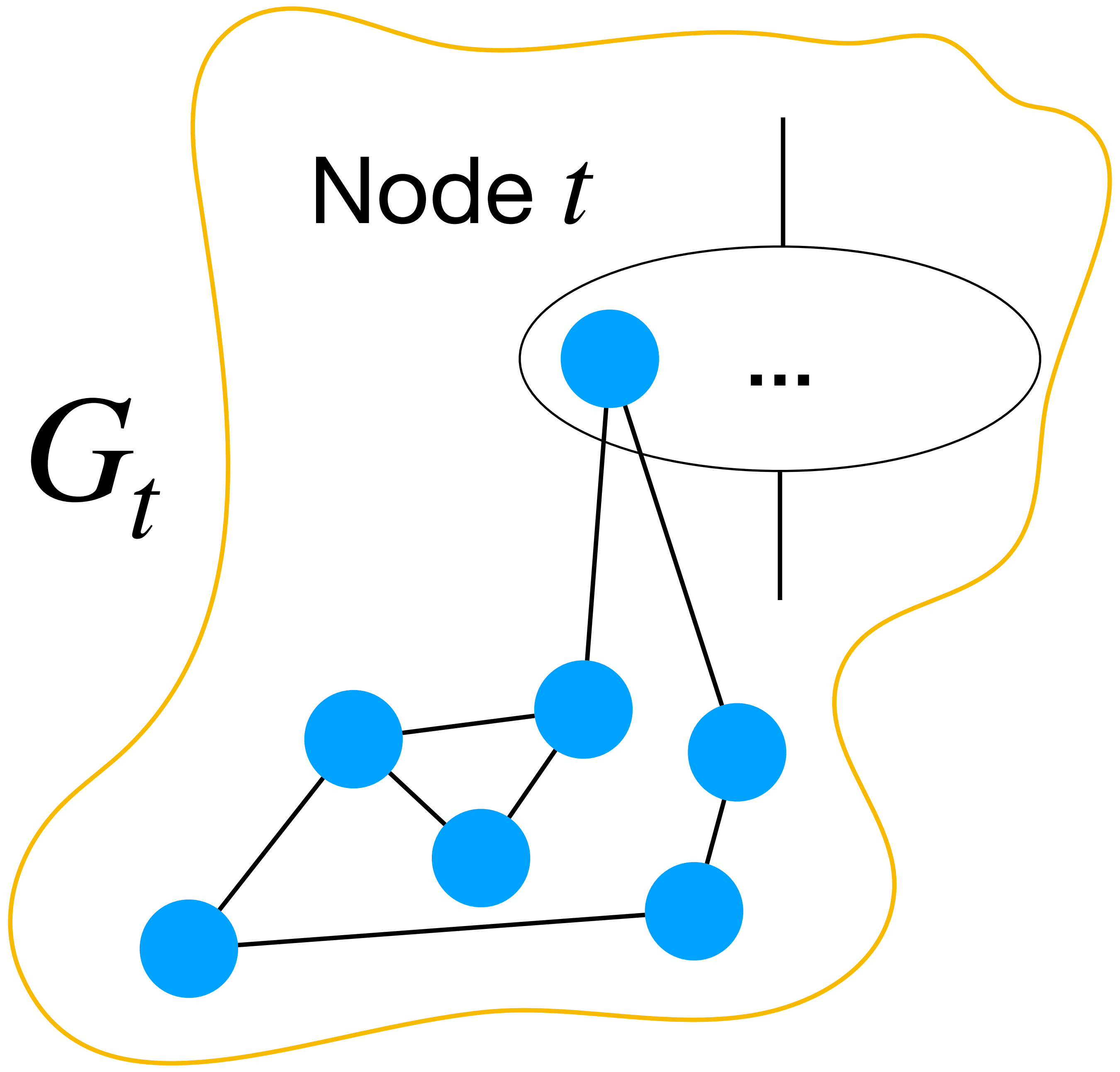


For each subset $S \subseteq \chi(t)$ of bag vertices,
$n_t(S)$ = | max. **IS** of G_t containing S |.

Treewidth k



Independent Set with Tree Decompositions

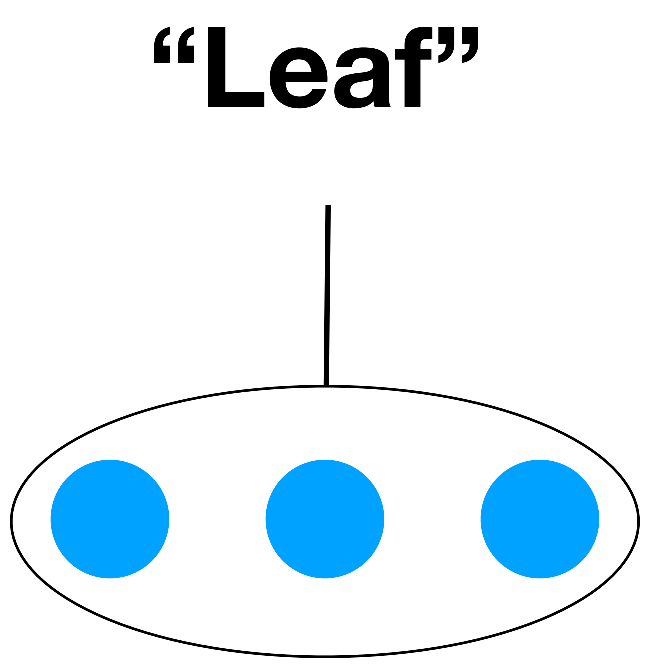


For each subset $S \subseteq \chi(t)$ of bag vertices,
$n_t(S)$ = | max. **IS** of G_t containing S |.

Treewidth k ➡ $\leq 2^{k+1}$ subsets

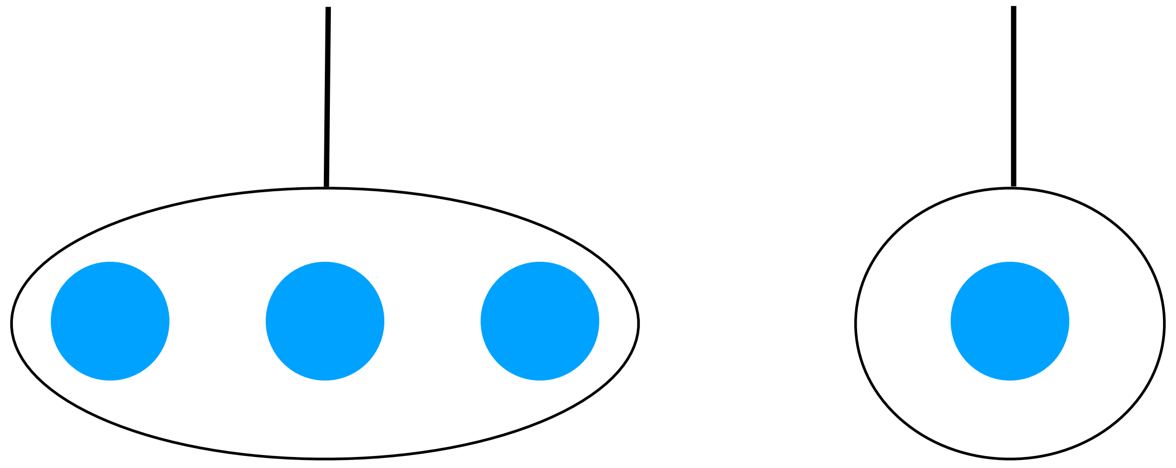
Nice Tree Decompositions

Nice Tree Decompositions



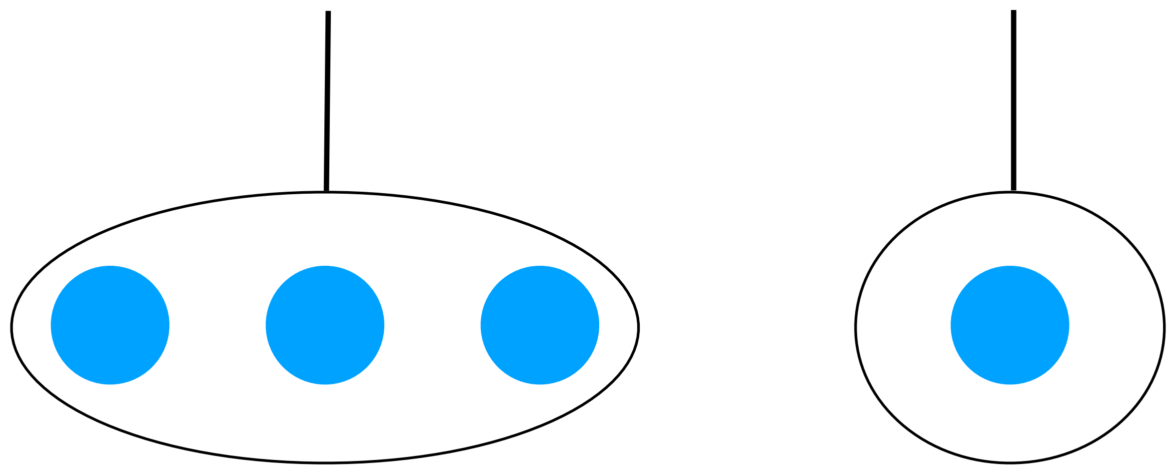
Nice Tree Decompositions

“Leaf”

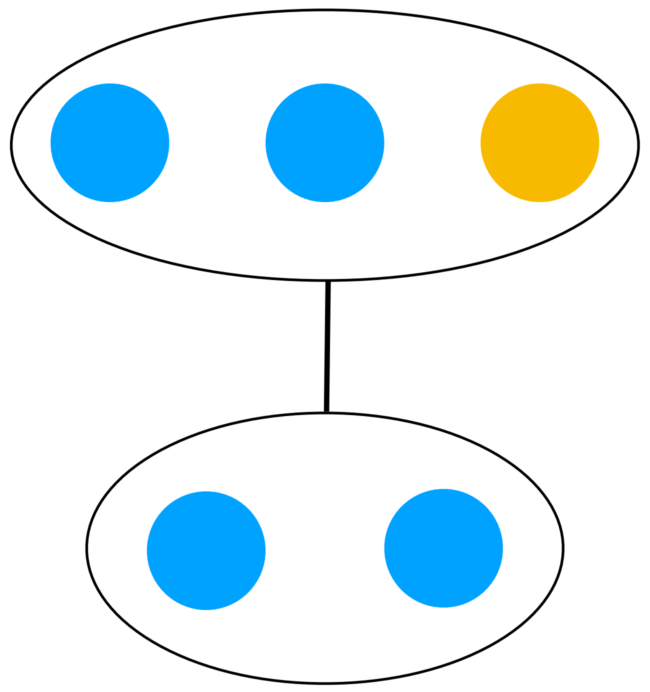


Nice Tree Decompositions

“Leaf”

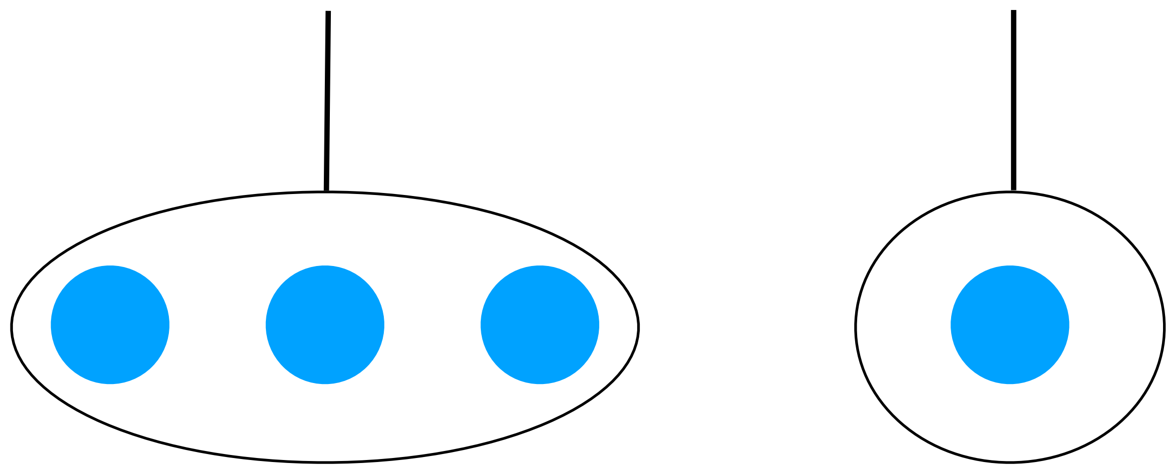


“Introduce”

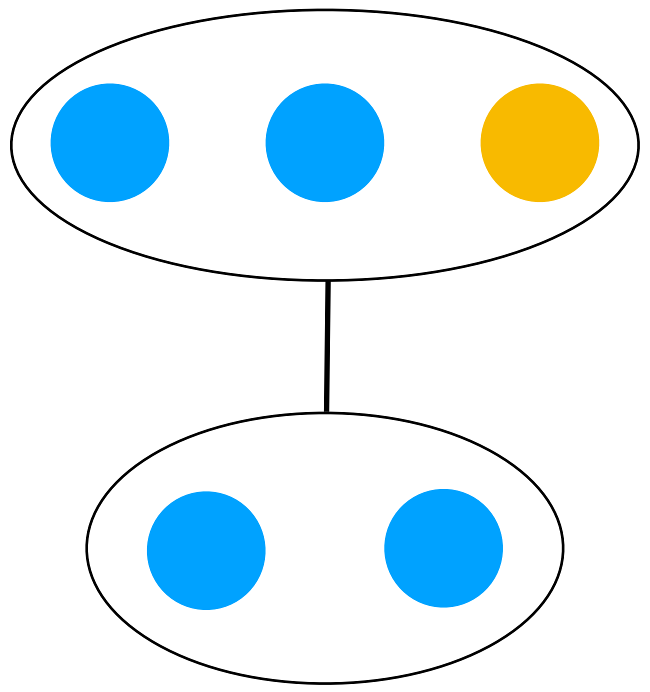


Nice Tree Decompositions

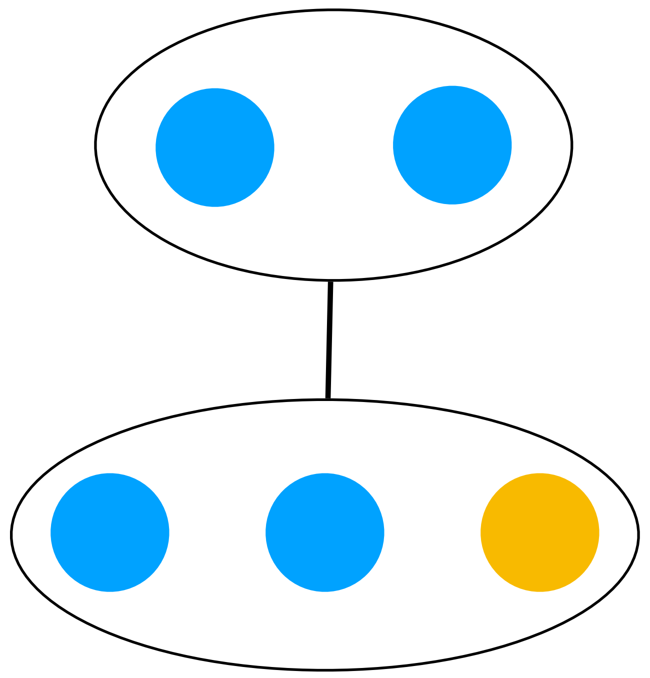
“Leaf”



“Introduce”

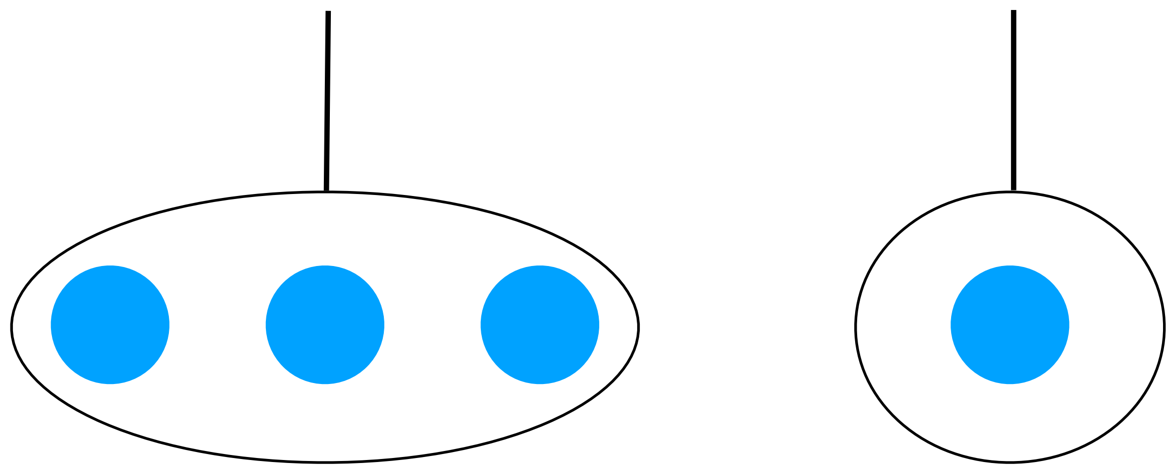


“Forget”

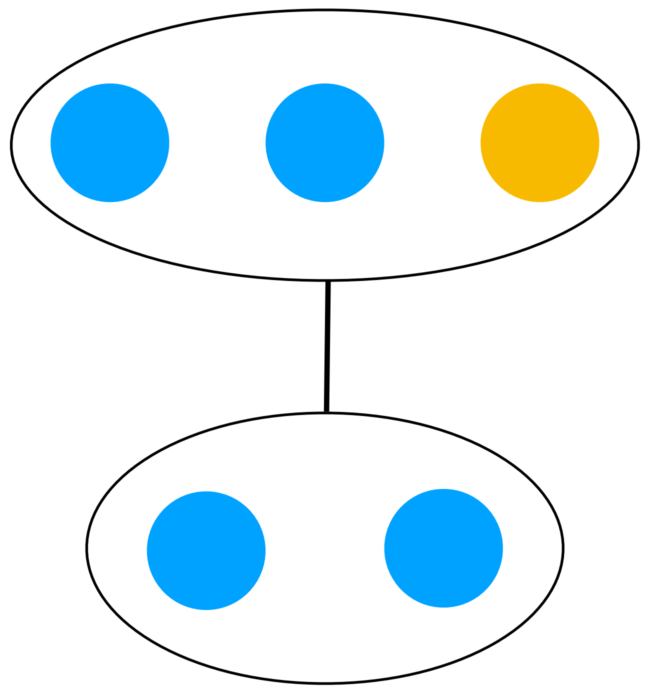


Nice Tree Decompositions

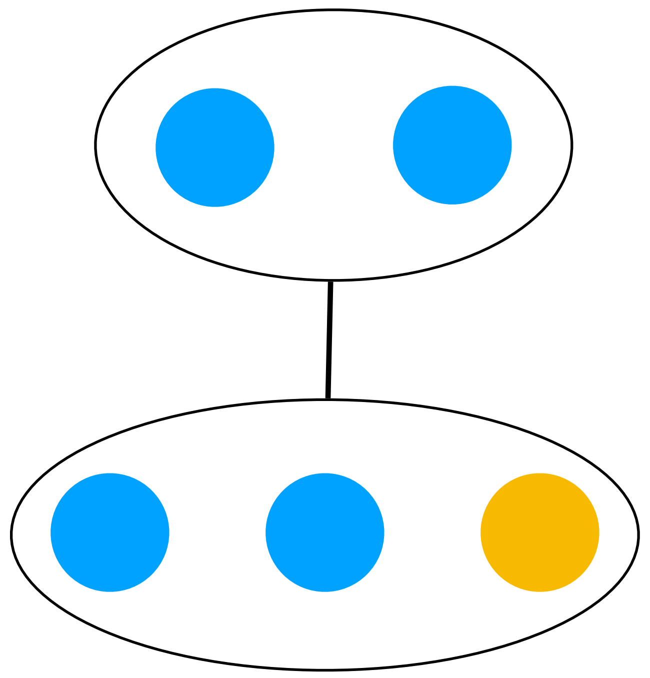
“Leaf”



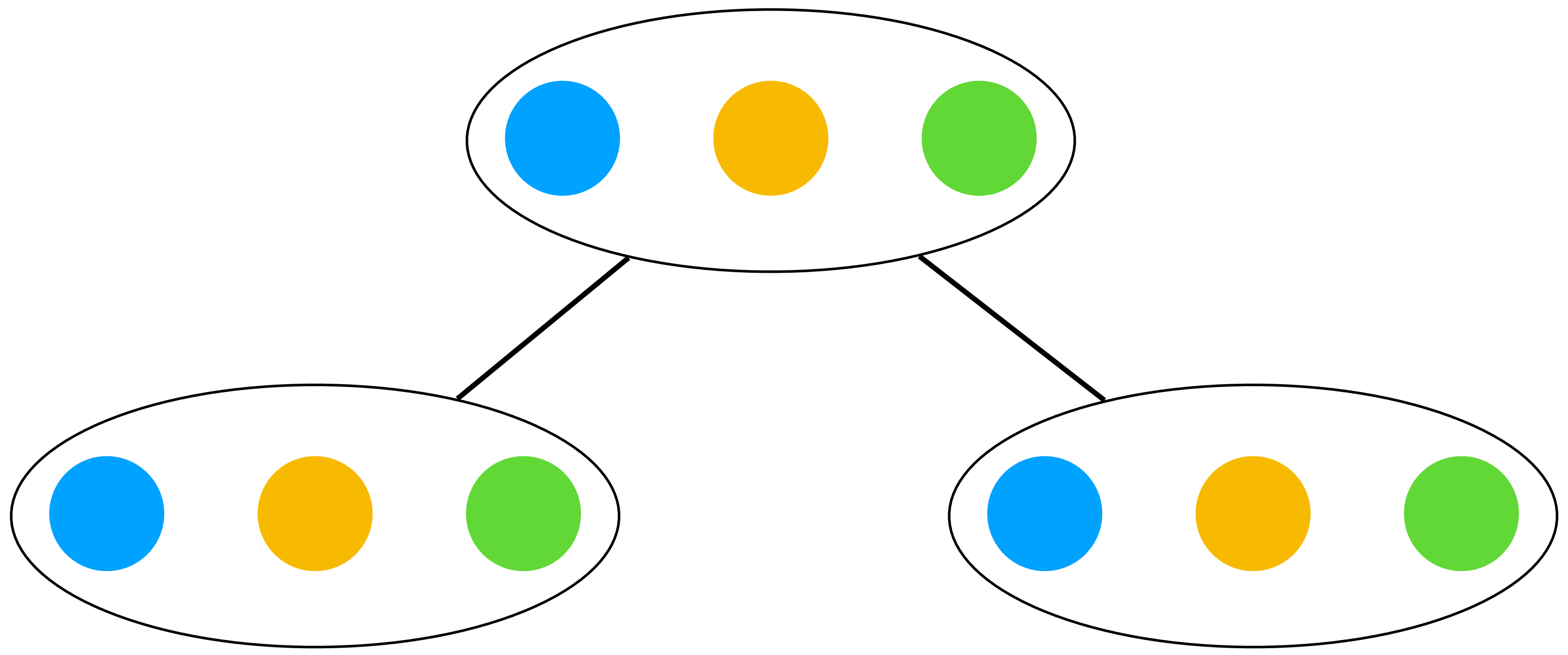
“Introduce”



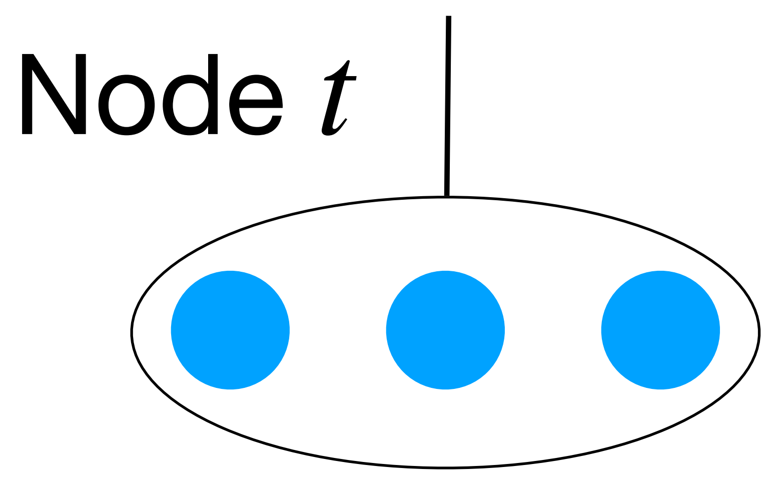
“Forget”



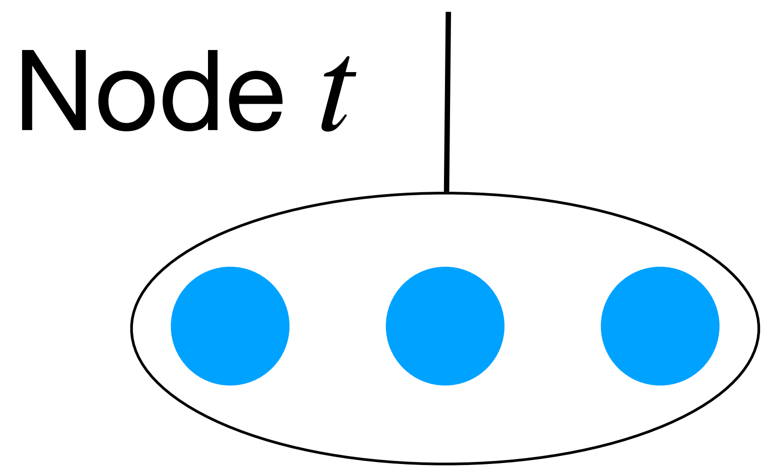
“Join”



Leaf Nodes

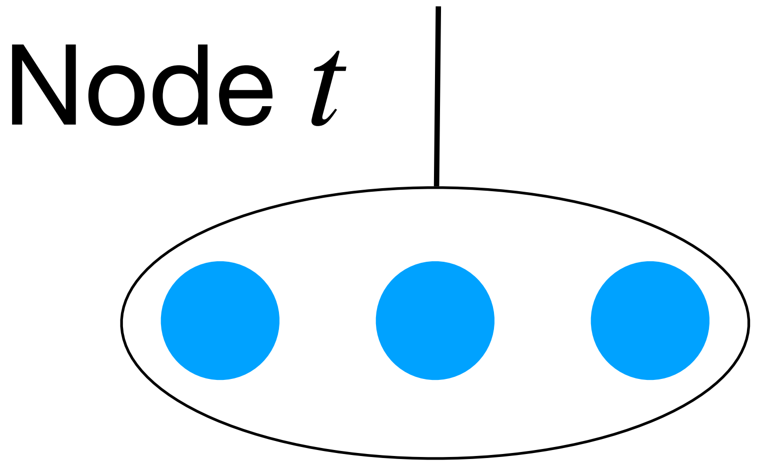


Leaf Nodes



for $S \subseteq \chi(t)$**:**

Leaf Nodes

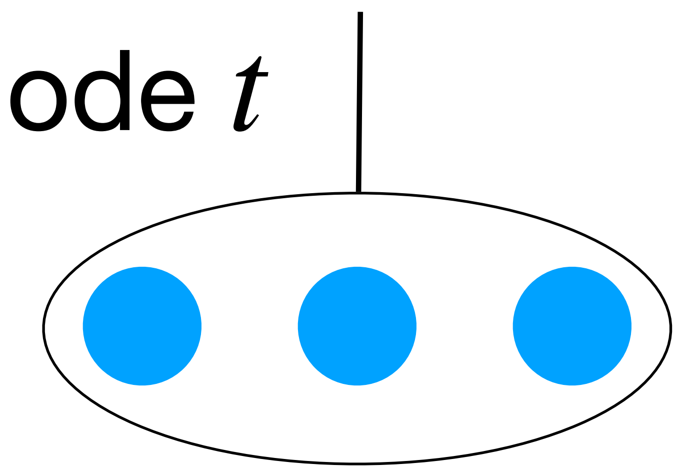


for $S \subseteq \chi(t)$:

  if S is an independent set

Leaf Nodes

Node t

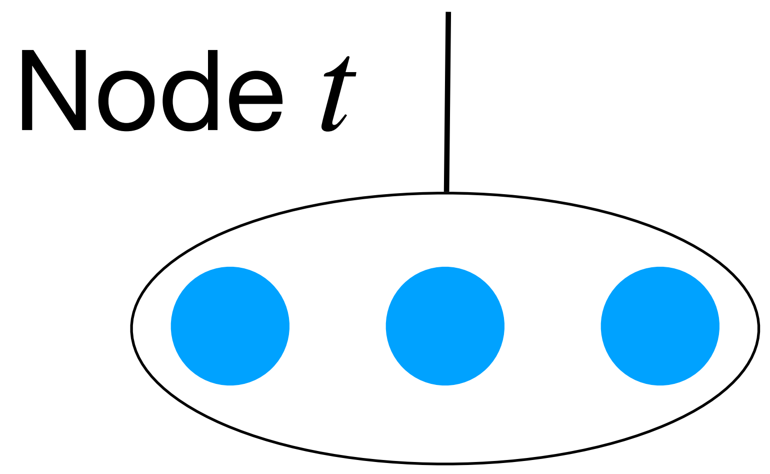


for $S \subseteq \chi(t)$:

if S is an independent set

then $n_t(S) := |S|$

Leaf Nodes



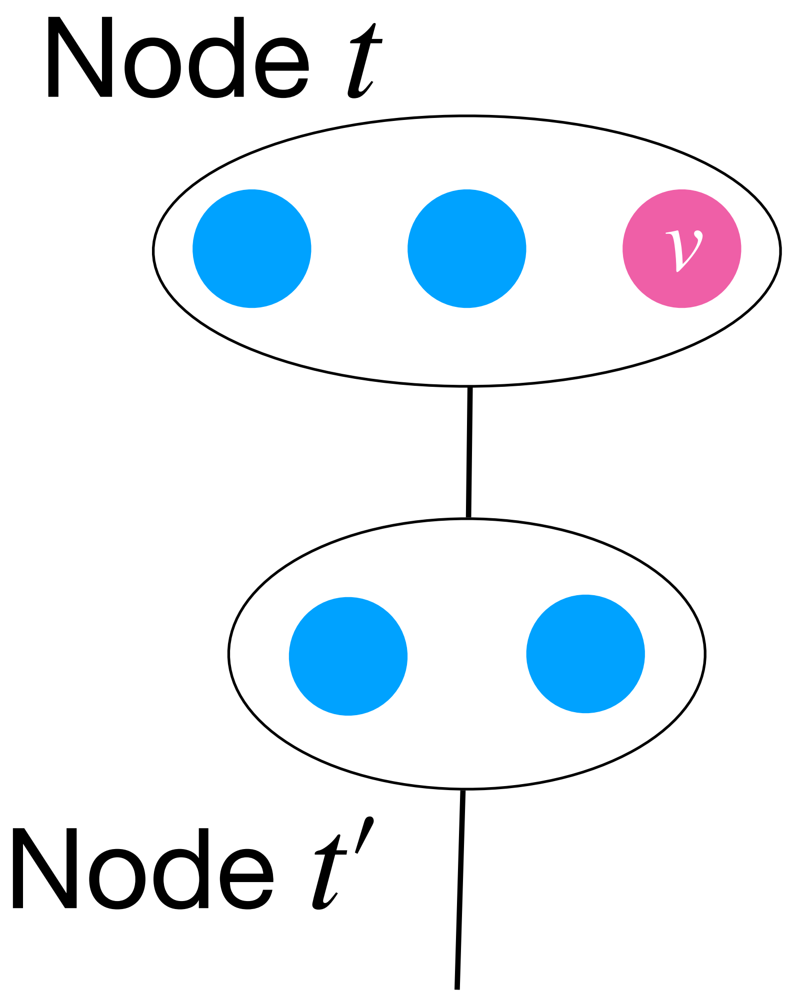
for $S \subseteq \chi(t)$:

  if S is an independent set

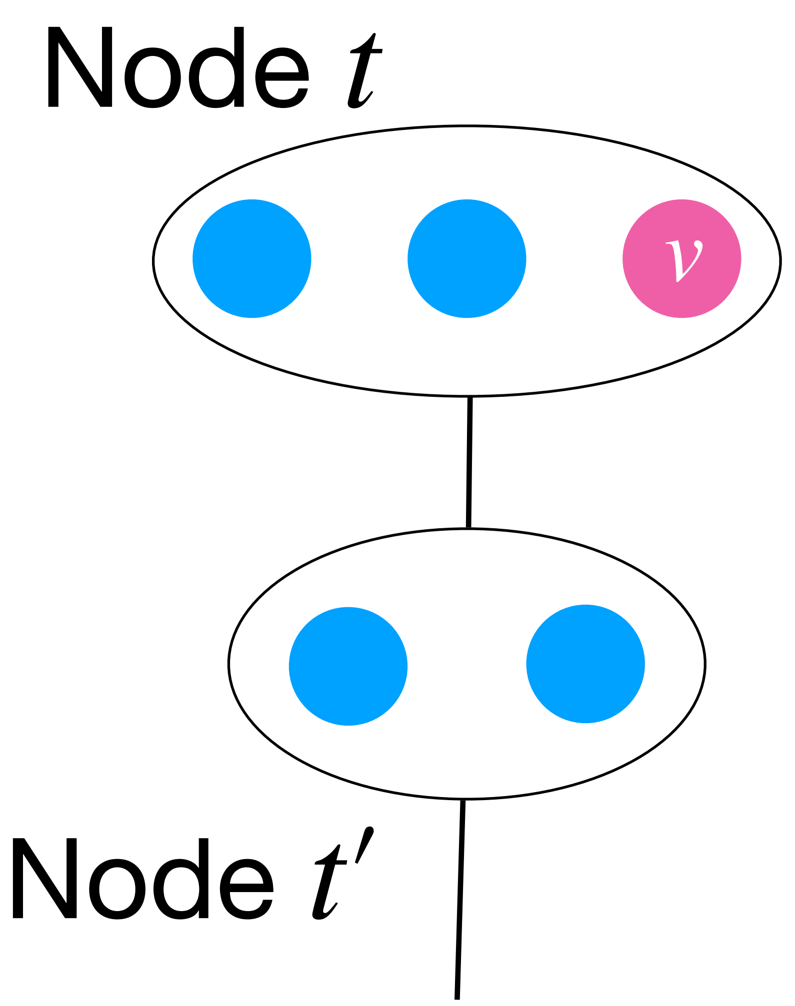
    then $n_t(S) := |S|$

    else $n_t(S) := 0$

Introduce Nodes

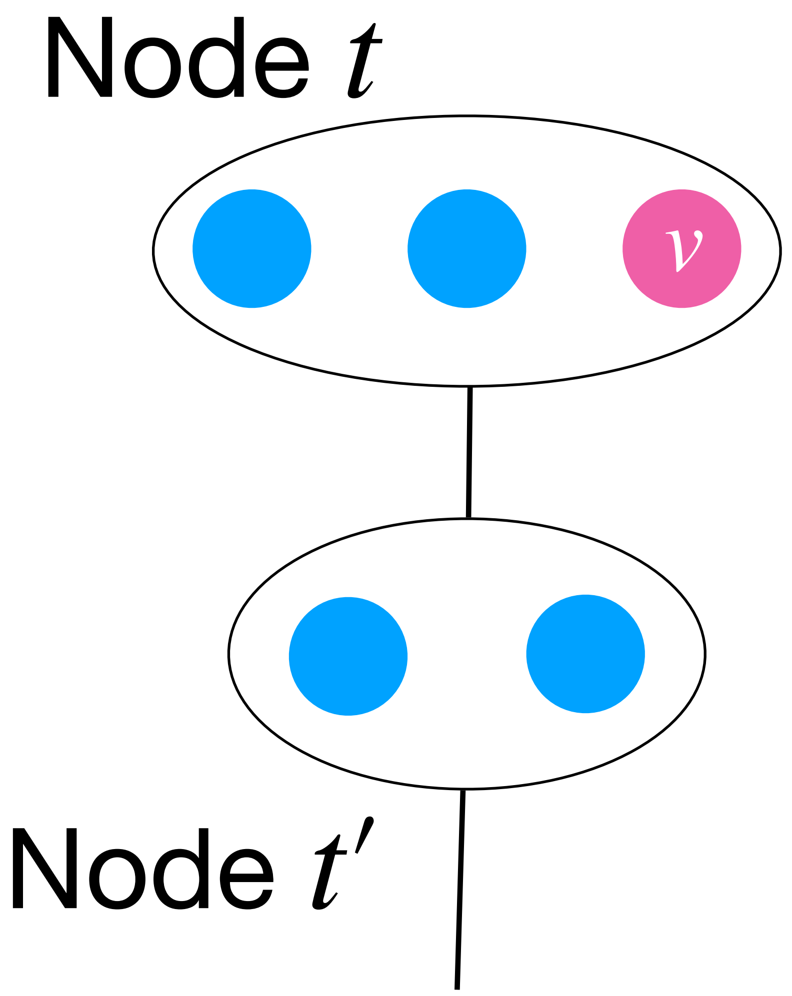


Introduce Nodes



for $S \subseteq \chi(t)$**:**

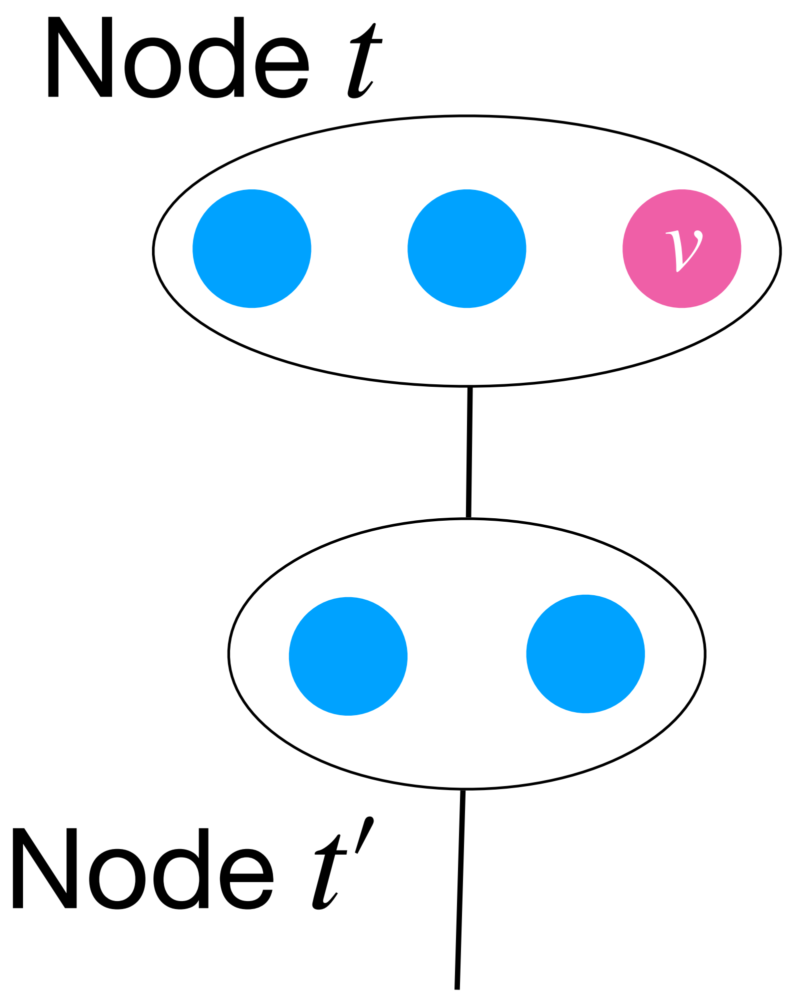
Introduce Nodes



for $S \subseteq \chi(t)$**:**

if $v \notin S$

Introduce Nodes

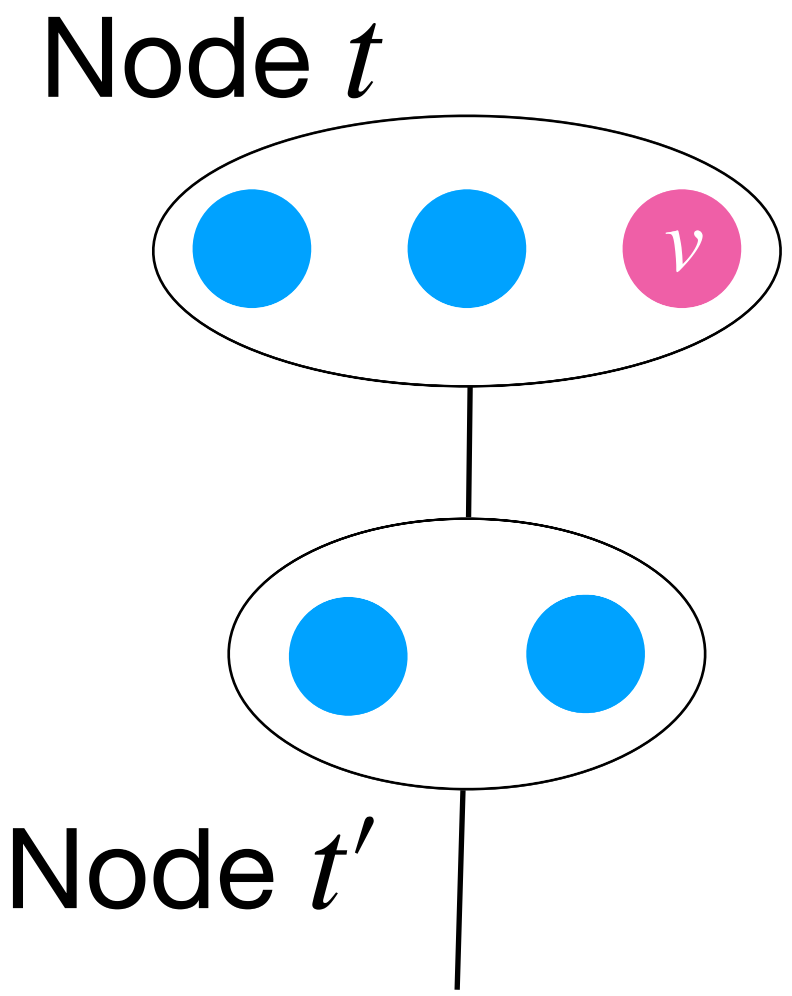


for $S \subseteq \chi(t)$:

if $v \notin S$

then $n_t(S) := n_{t'}(S)$

Introduce Nodes



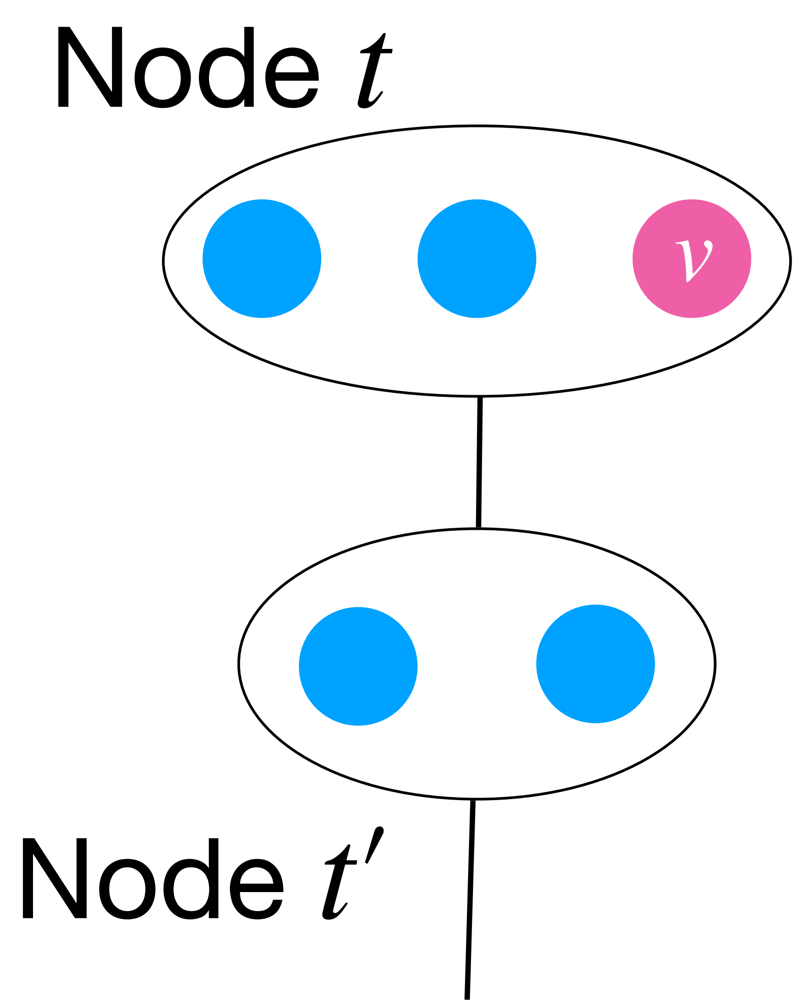
for $S \subseteq \chi(t)$:

if $v \notin S$

then $n_t(S) := n_{t'}(S)$

else if $v \in S$ and S is not an IS

Introduce Nodes



for $S \subseteq \chi(t)$:

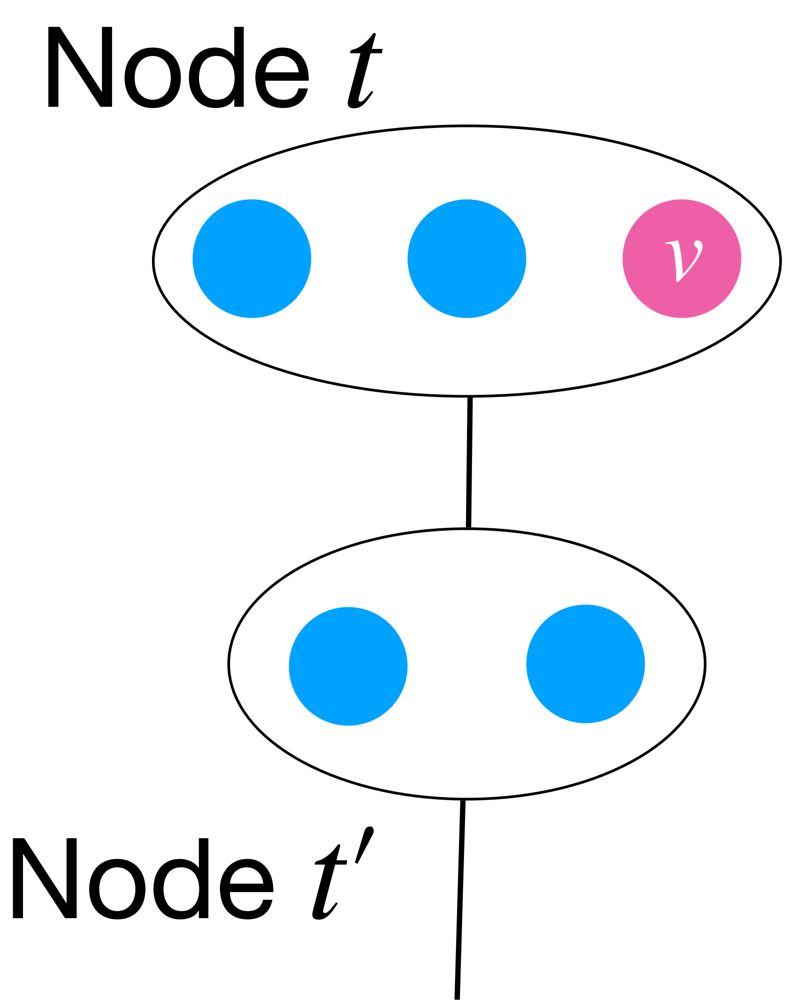
if $v \notin S$

then $n_t(S) := n_{t'}(S)$

else if $v \in S$ and S is not an IS

then $n_t(S) := 0$

Introduce Nodes



for $S \subseteq \chi(t)$:

if $v \notin S$

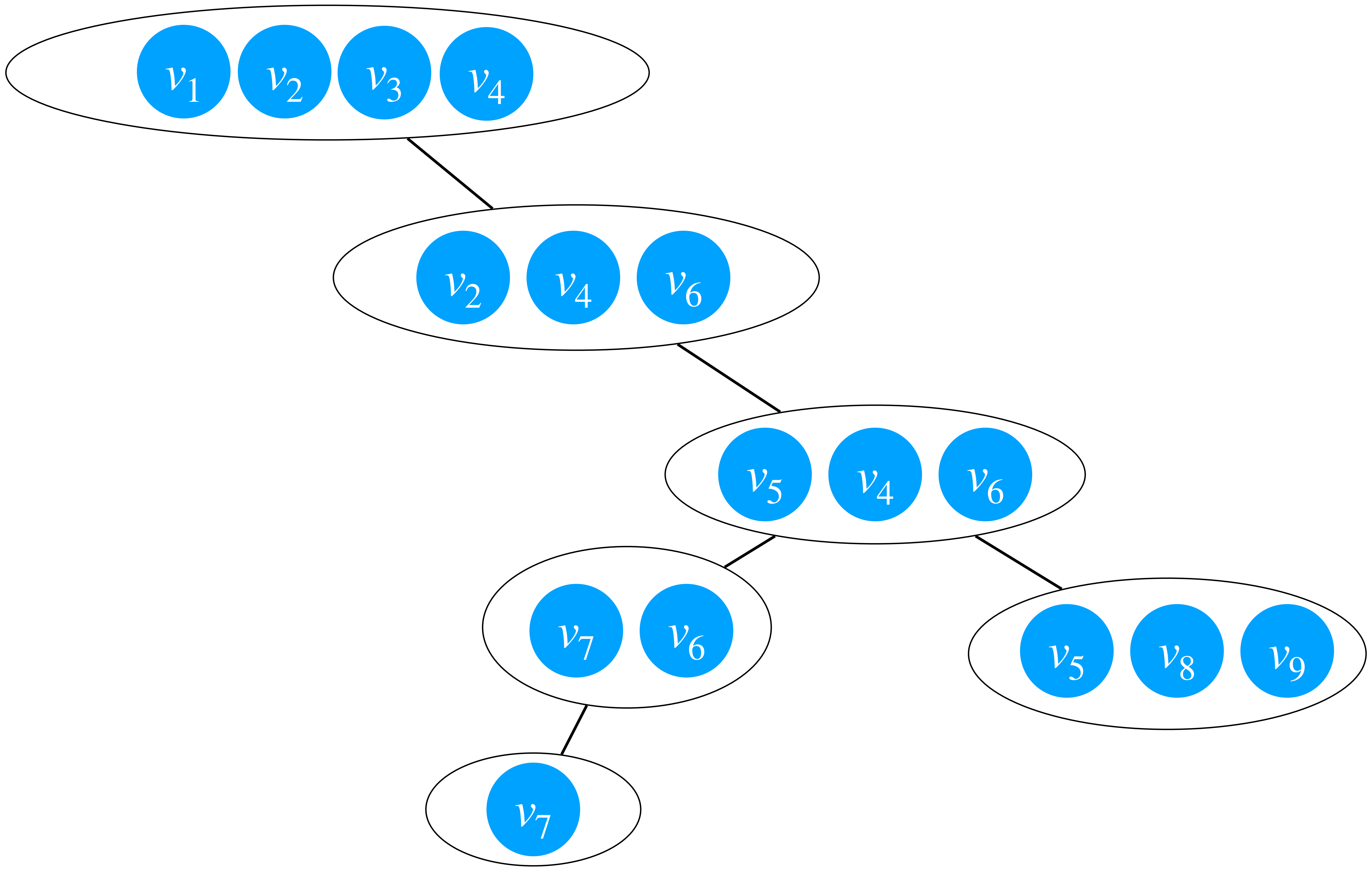
then $n_t(S) := n_{t'}(S)$

else if $v \in S$ **and** S **is not an IS**

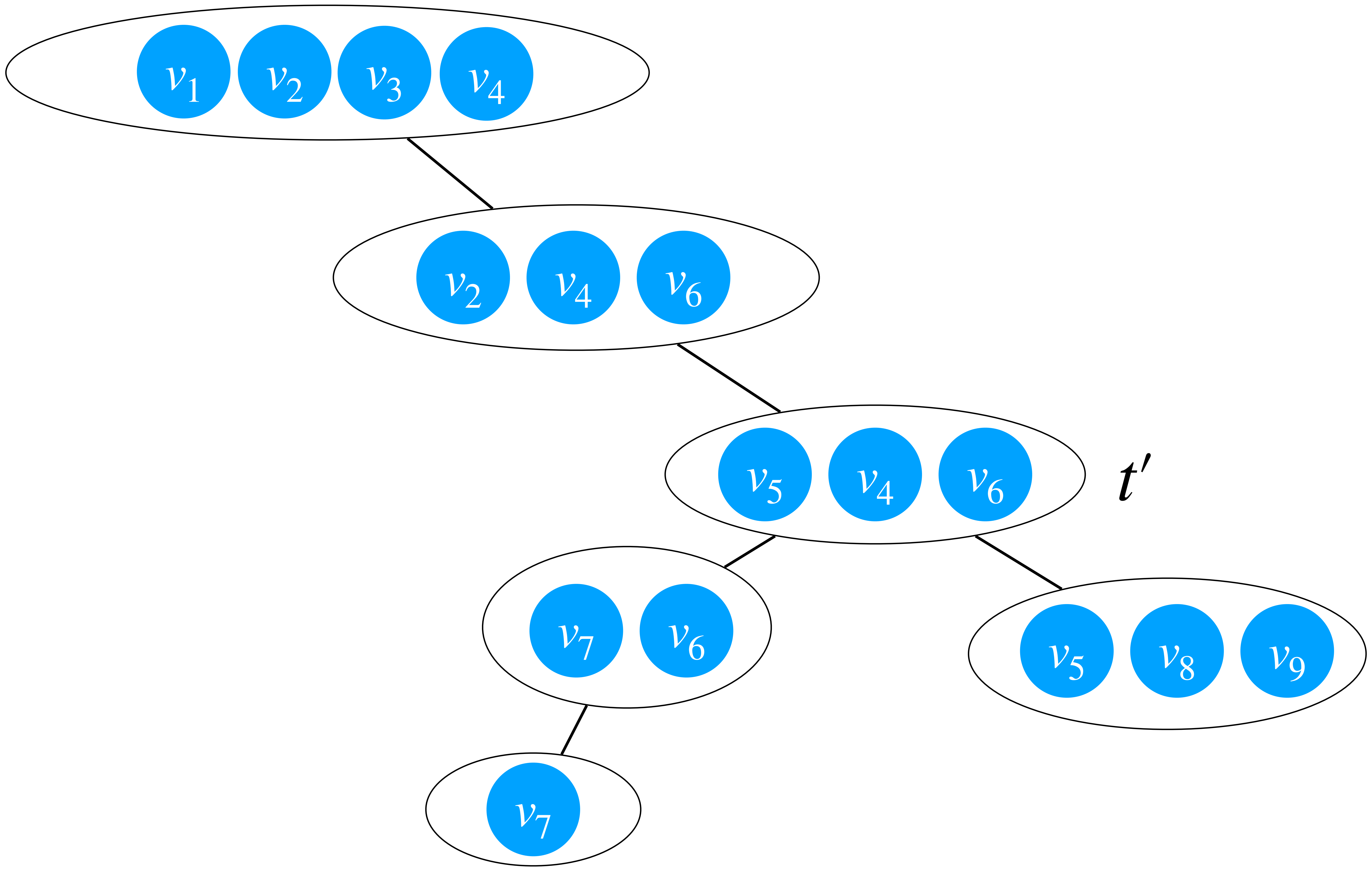
then $n_t(S) := 0$

else $n_t(S) := n_{t'}(S \backslash v) + 1$

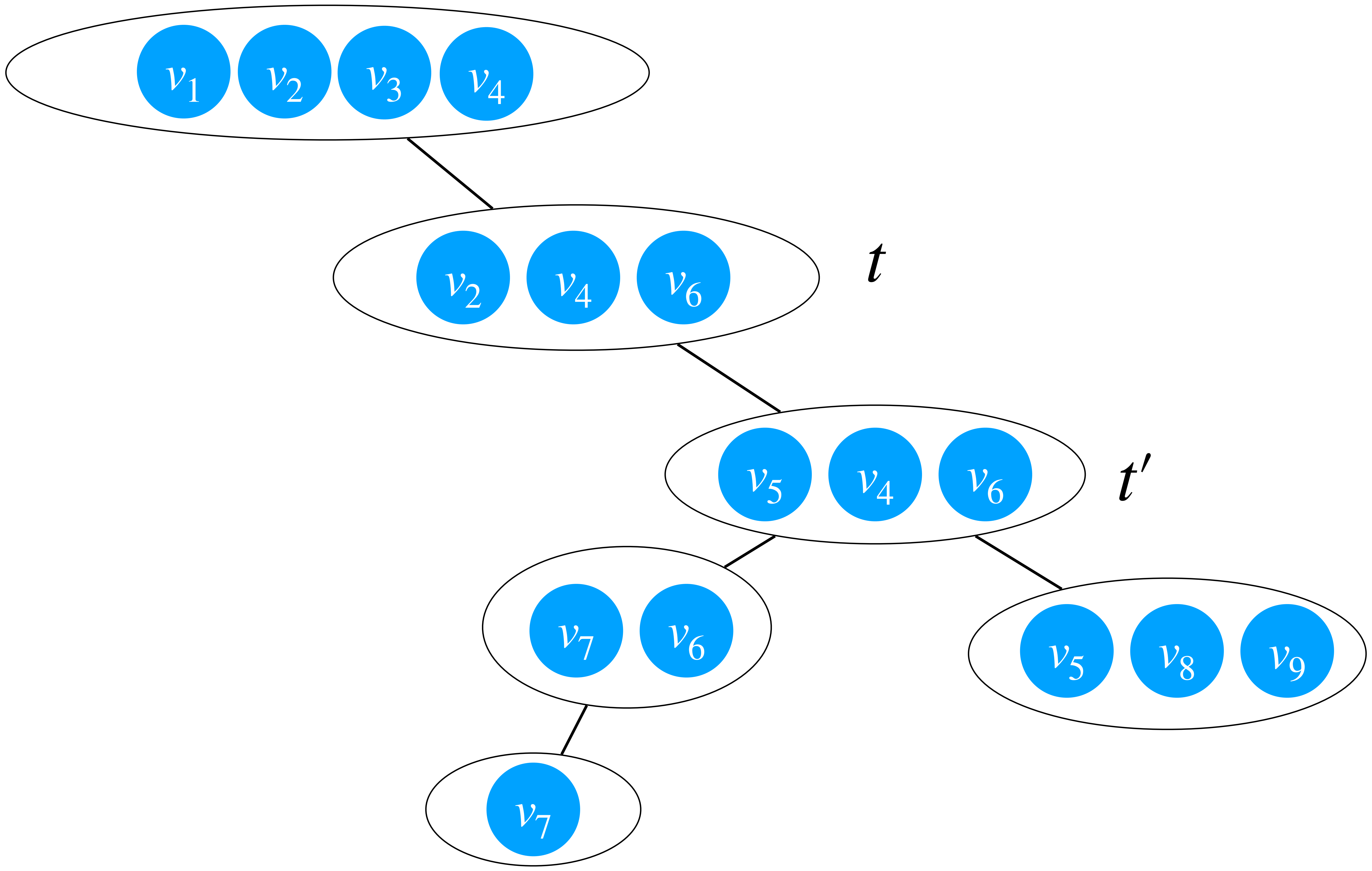
Bags are Separators



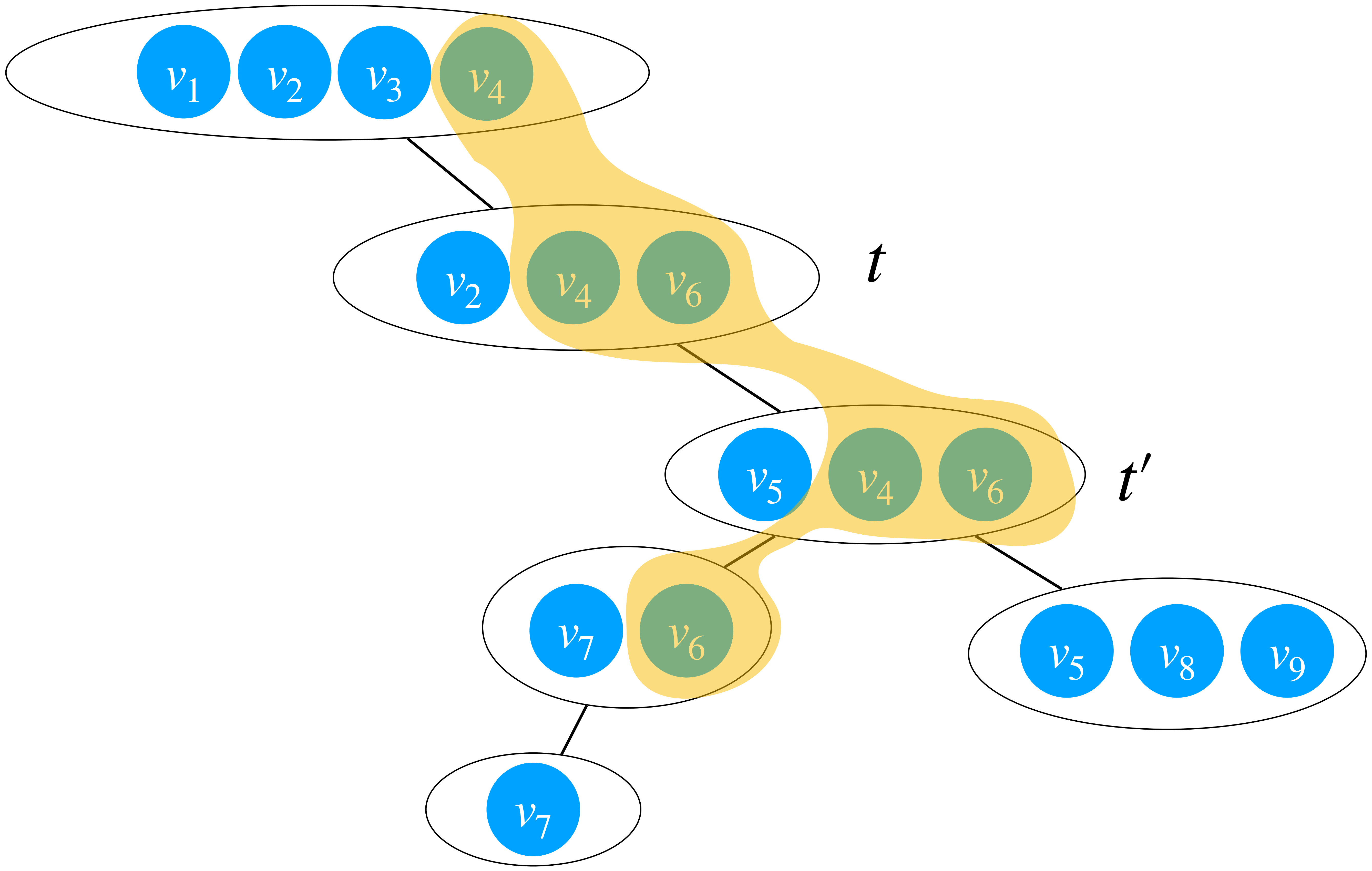
Bags are Separators



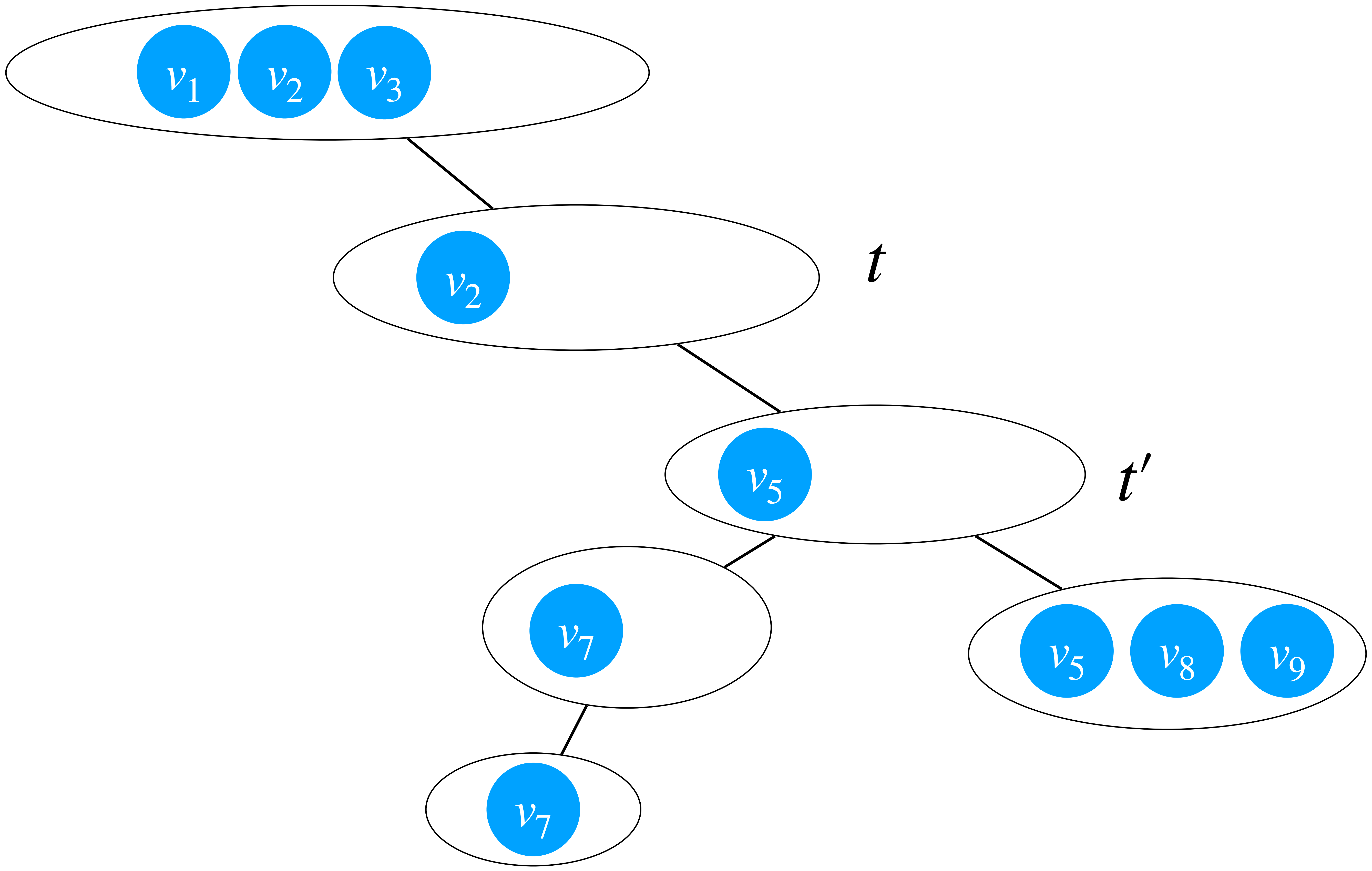
Bags are Separators



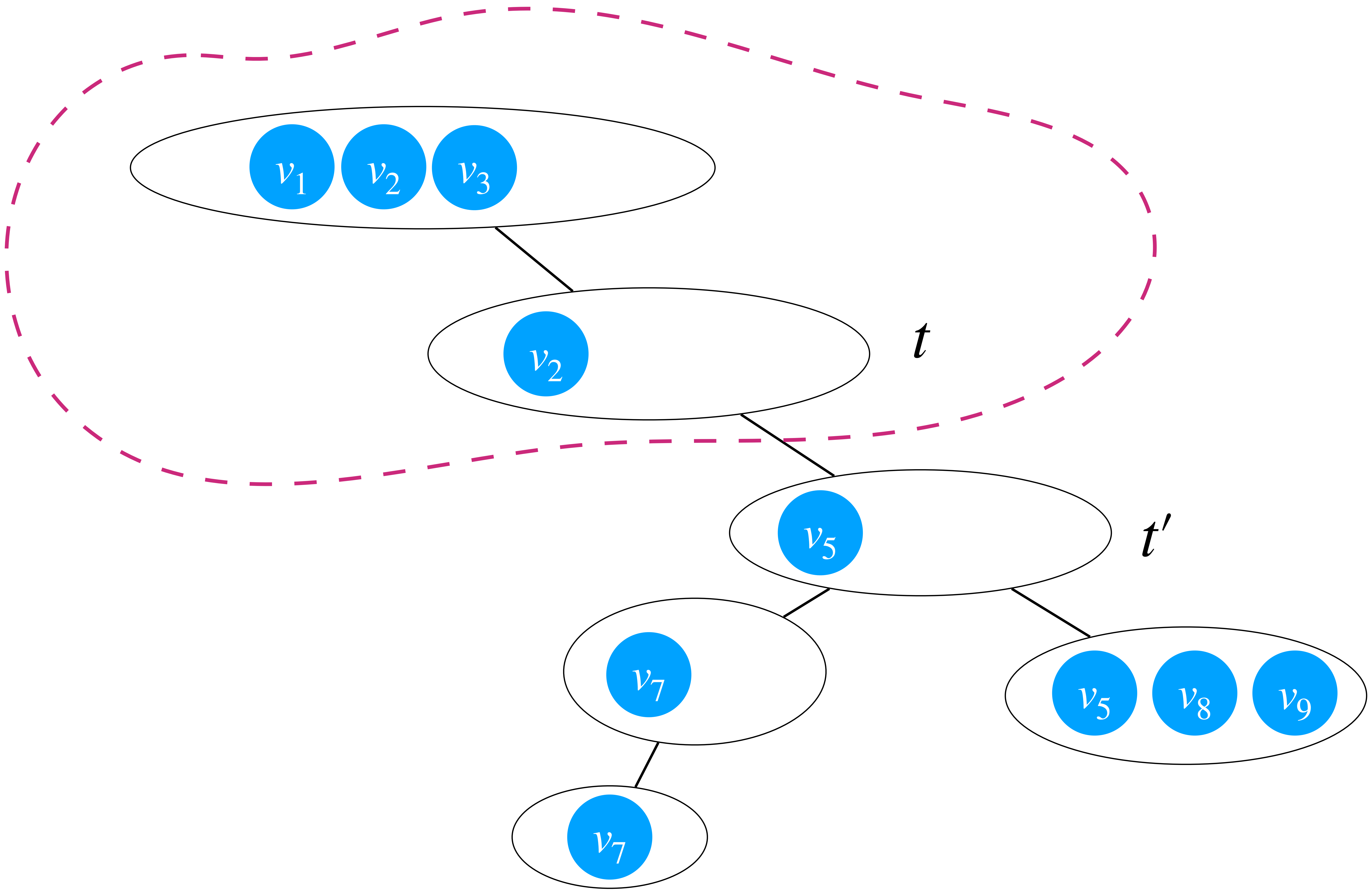
Bags are Separators



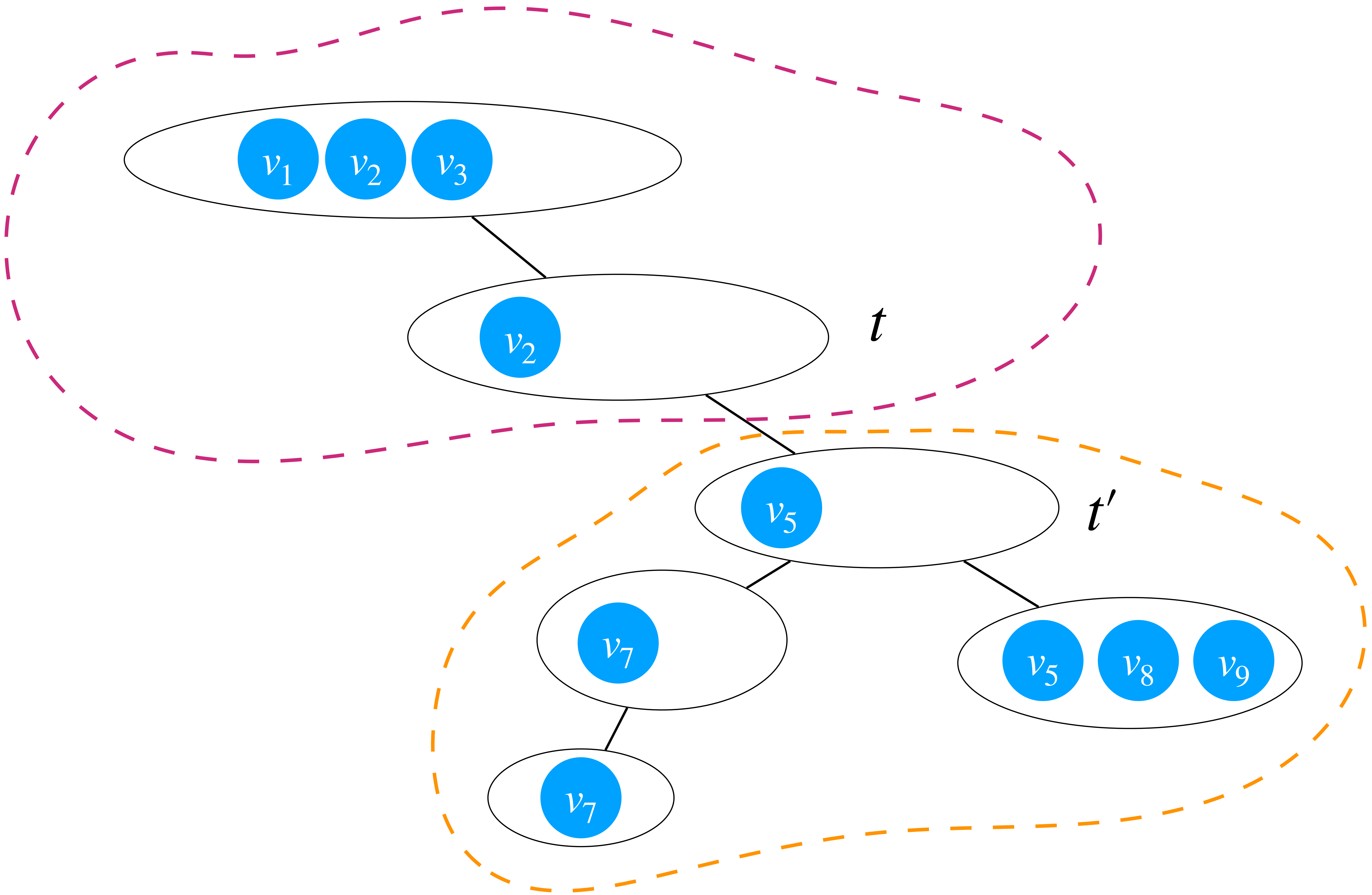
Bags are Separators



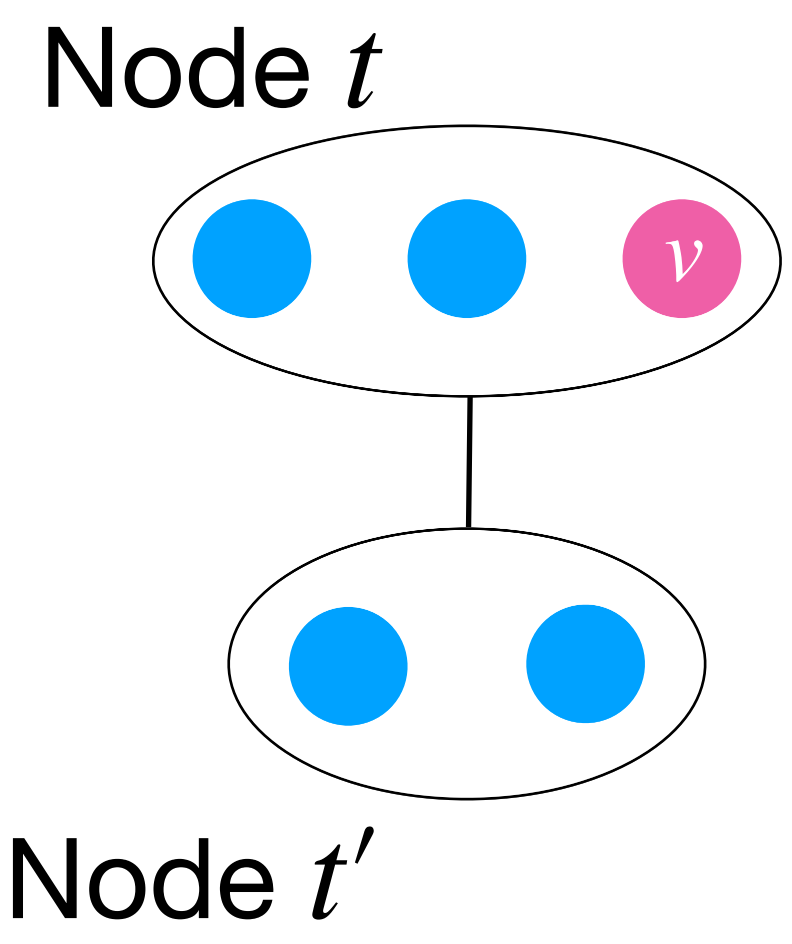
Bags are Separators



Bags are Separators



Introduce Nodes



for $S \subseteq \chi(t)$:

if $v \notin S$

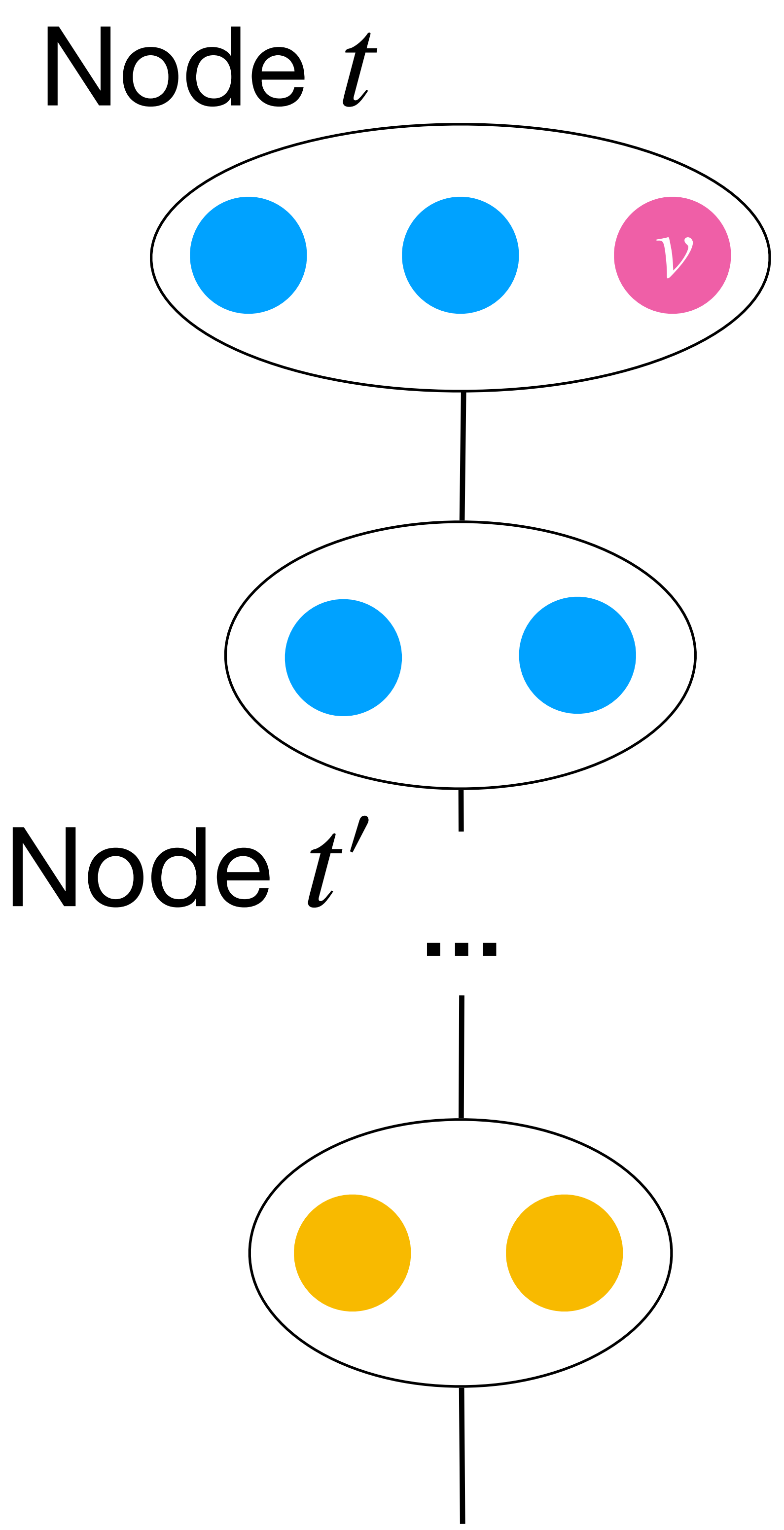
then $n_t(S) := n_{t'}(S)$

else if $v \in S$ and S is not an IS

then $n_t(S) := 0$

else $n_t(S) := n_{t'}(S \backslash v) + 1$

Introduce Nodes



for $S \subseteq \chi(t)$:

if $v \notin S$

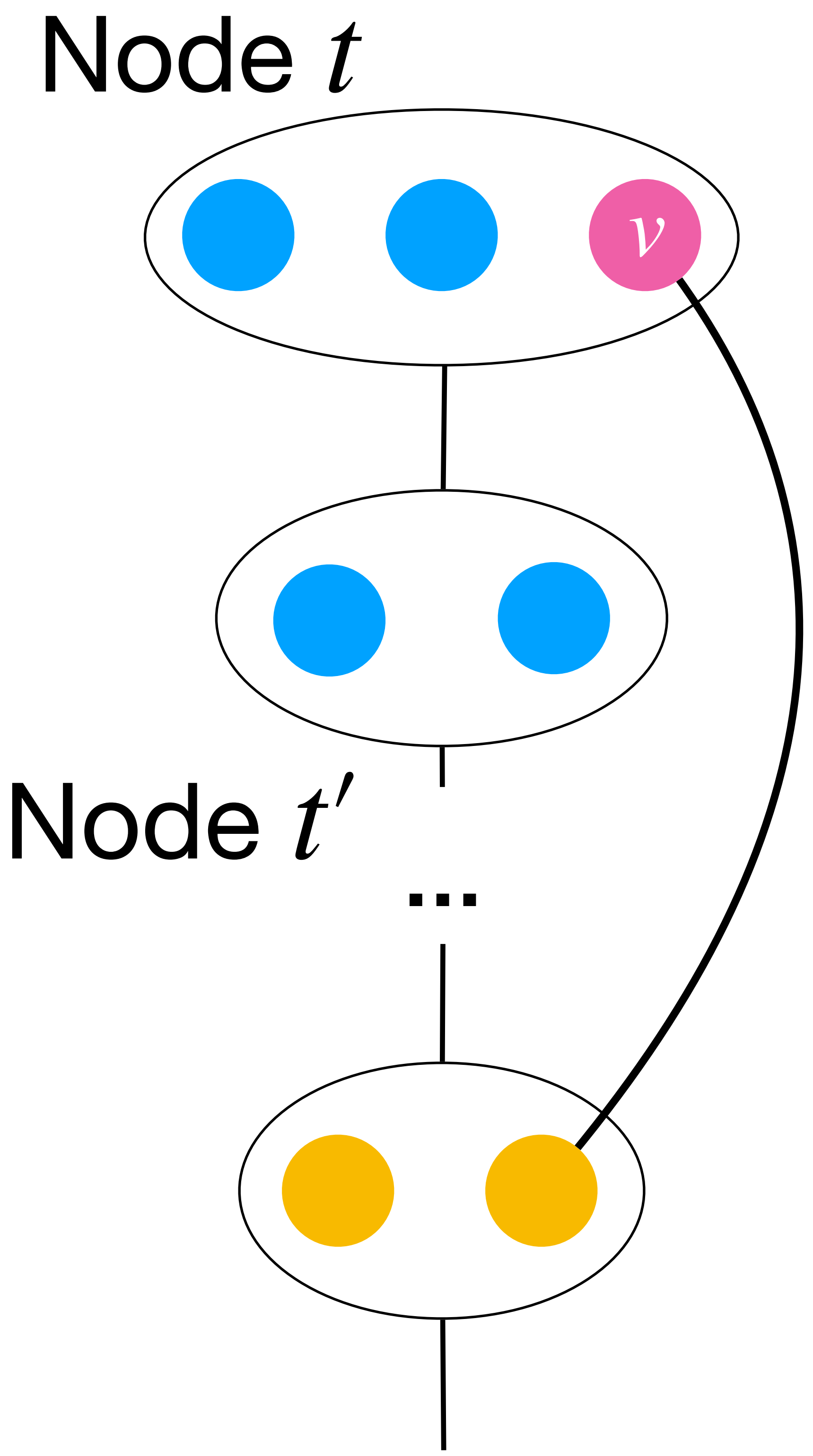
then $n_t(S) := n_{t'}(S)$

else if $v \in S$ **and** S **is not an IS**

then $n_t(S) := 0$

else $n_t(S) := n_{t'}(S \backslash v) + 1$

Introduce Nodes



for $S \subseteq \chi(t)$:

if $v \notin S$

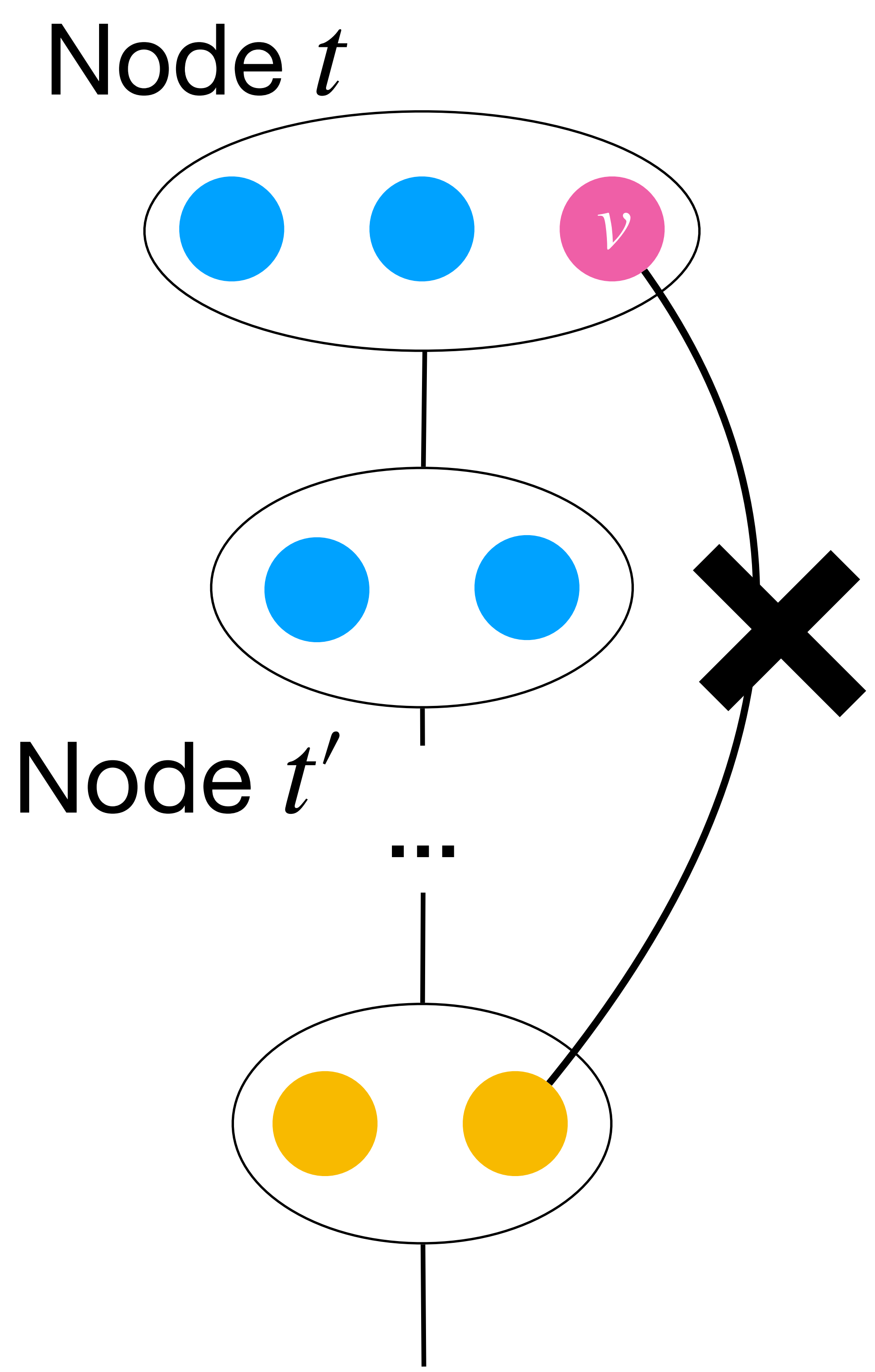
then $n_t(S) := n_{t'}(S)$

else if $v \in S$ **and** S **is not an IS**

then $n_t(S) := 0$

else $n_t(S) := n_{t'}(S \backslash v) + 1$

Introduce Nodes



for $S \subseteq \chi(t)$:

if $v \notin S$

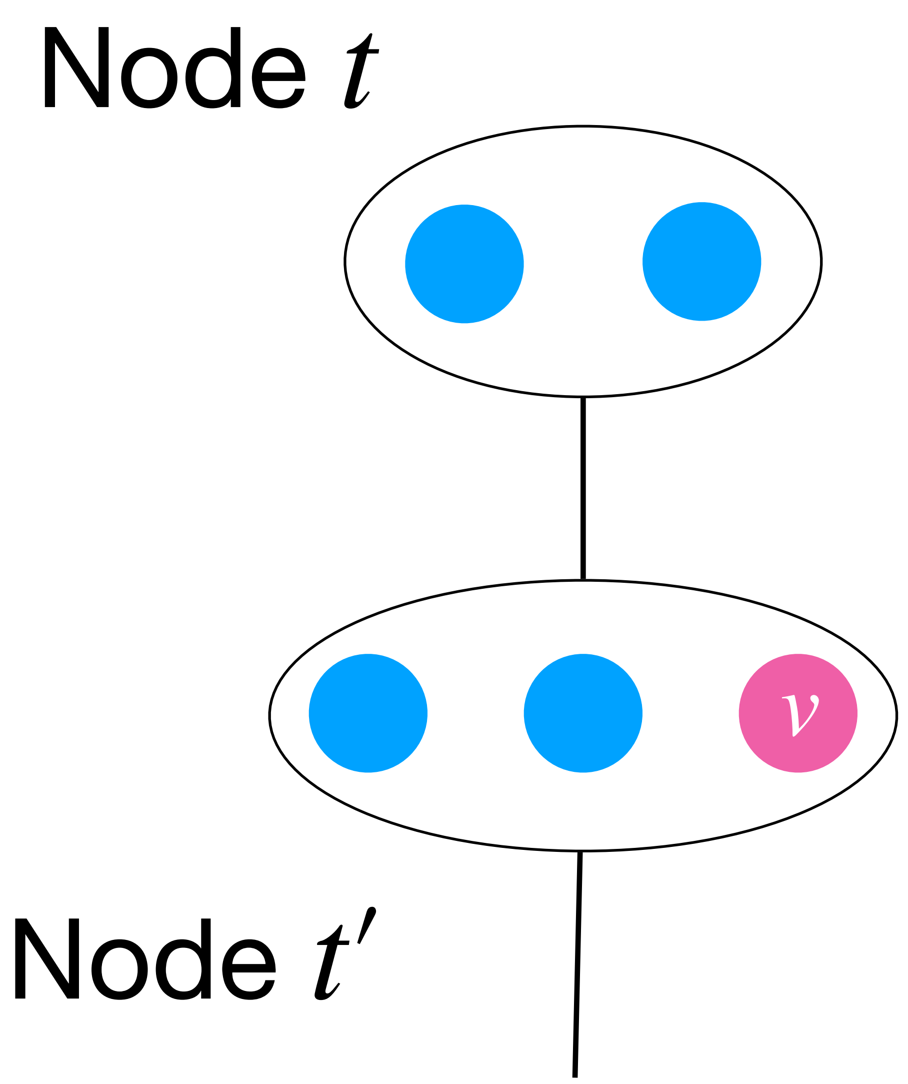
then $n_t(S) := n_{t'}(S)$

else if $v \in S$ **and** S **is not an IS**

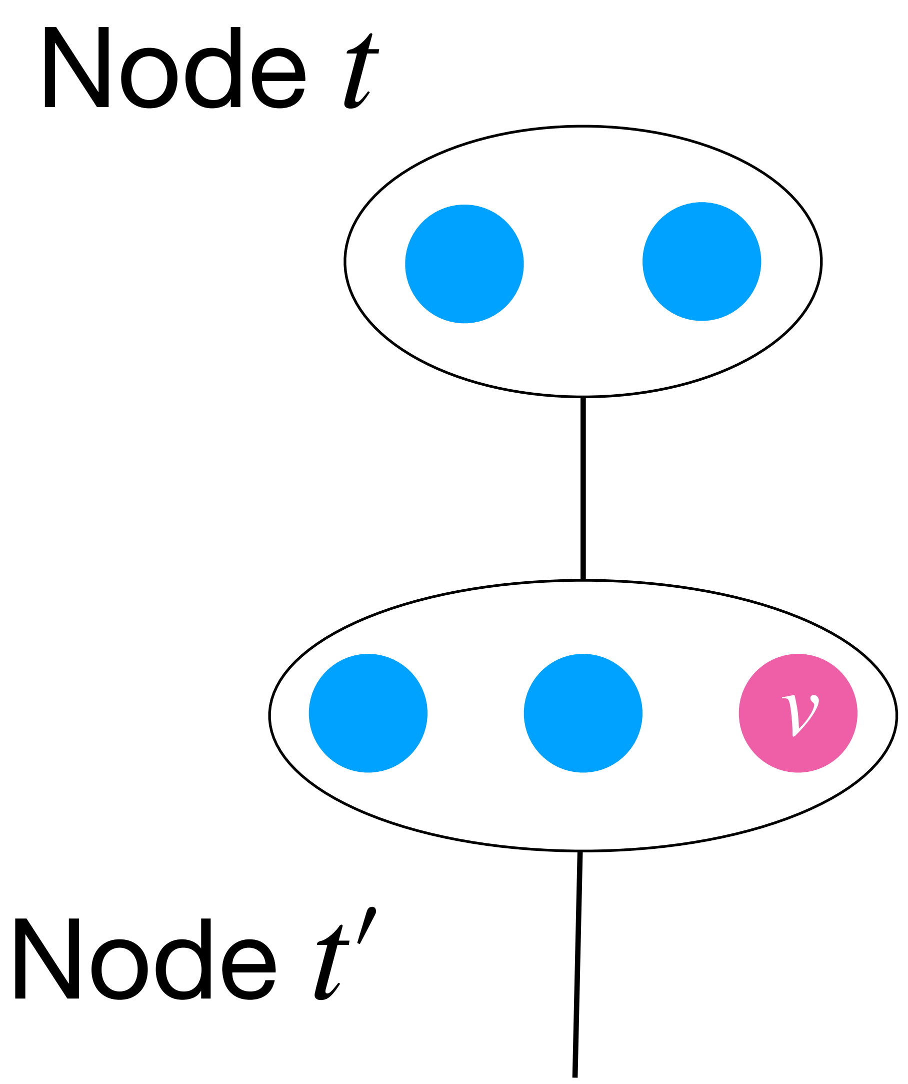
then $n_t(S) := 0$

else $n_t(S) := n_{t'}(S \backslash v) + 1$

Forget Nodes

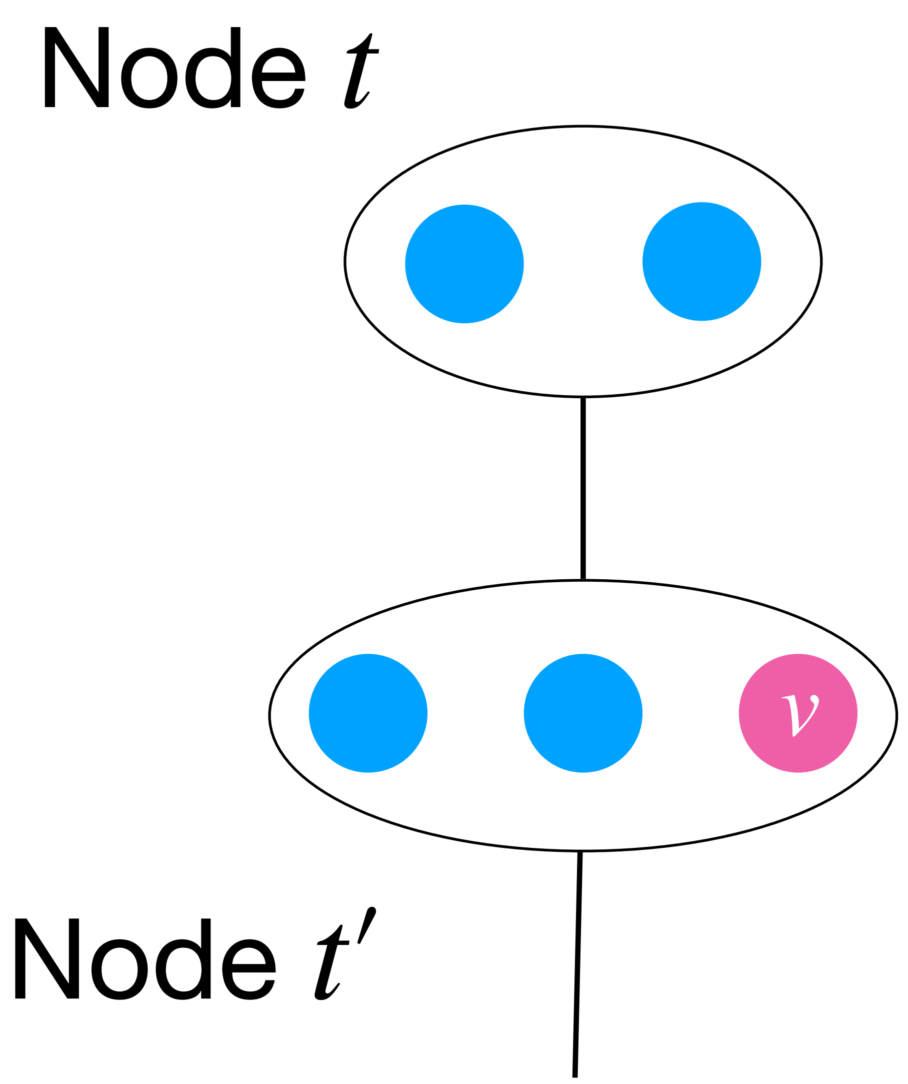


Forget Nodes



for $S \subseteq \chi(t)$

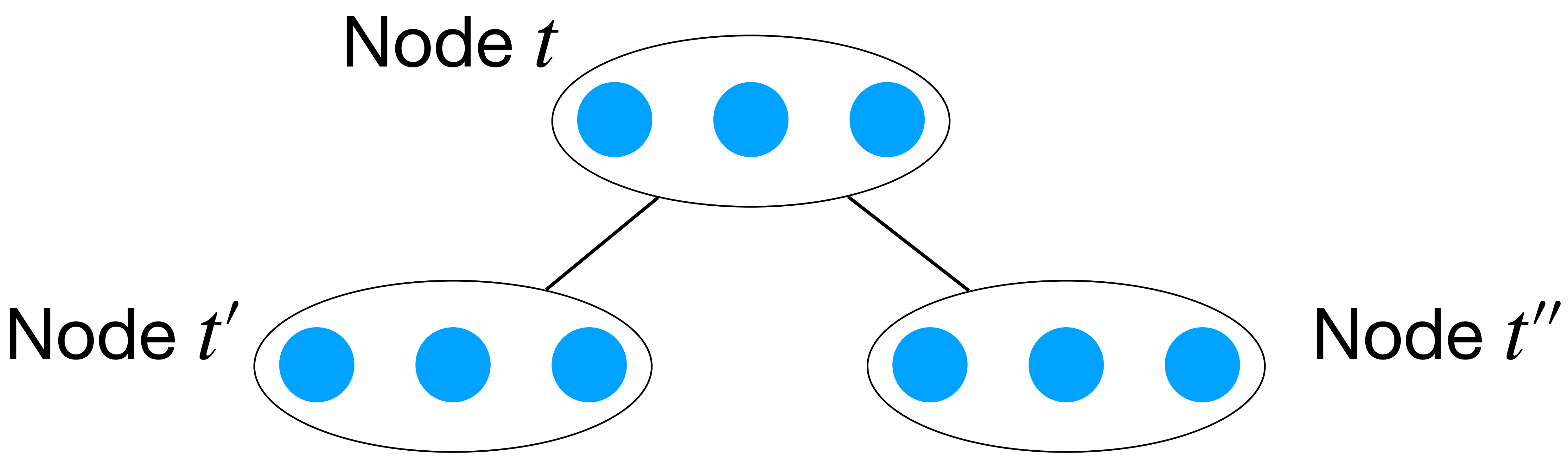
Forget Nodes



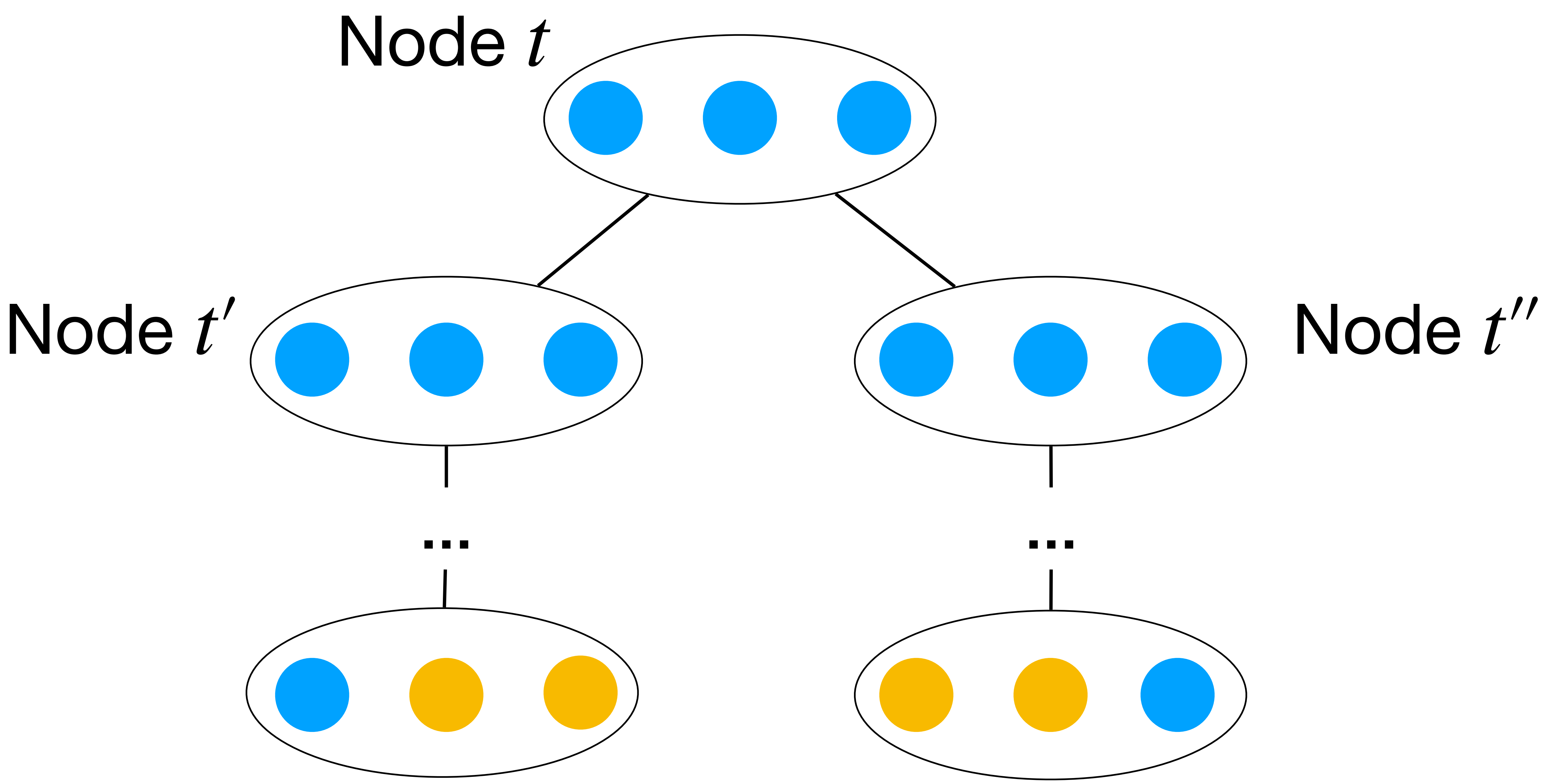
for $S \subseteq \chi(t)$

$$n_t(S) := \max(n_{t'}(S), n_{t'}(S \cup \{v\}))$$

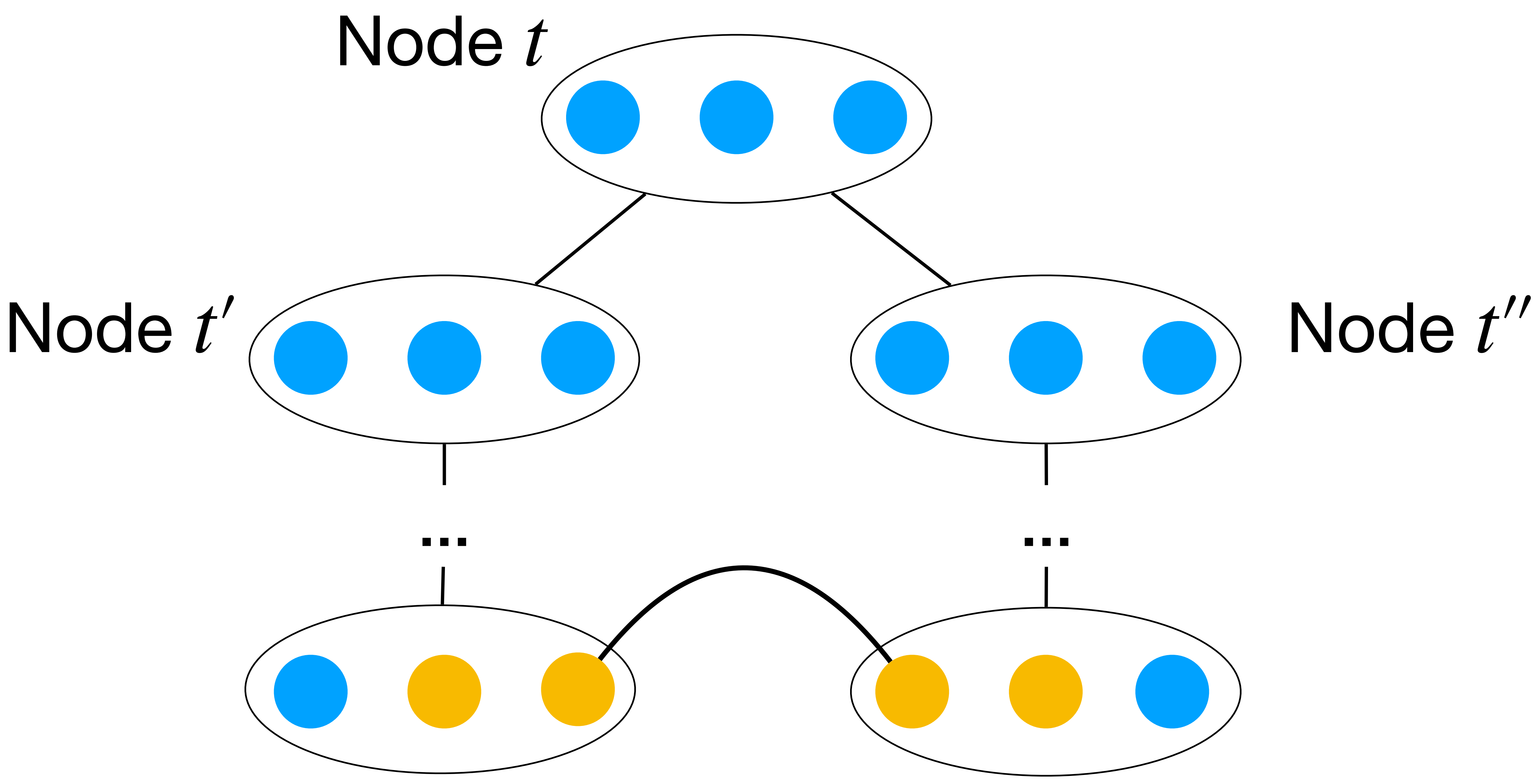
Join Nodes



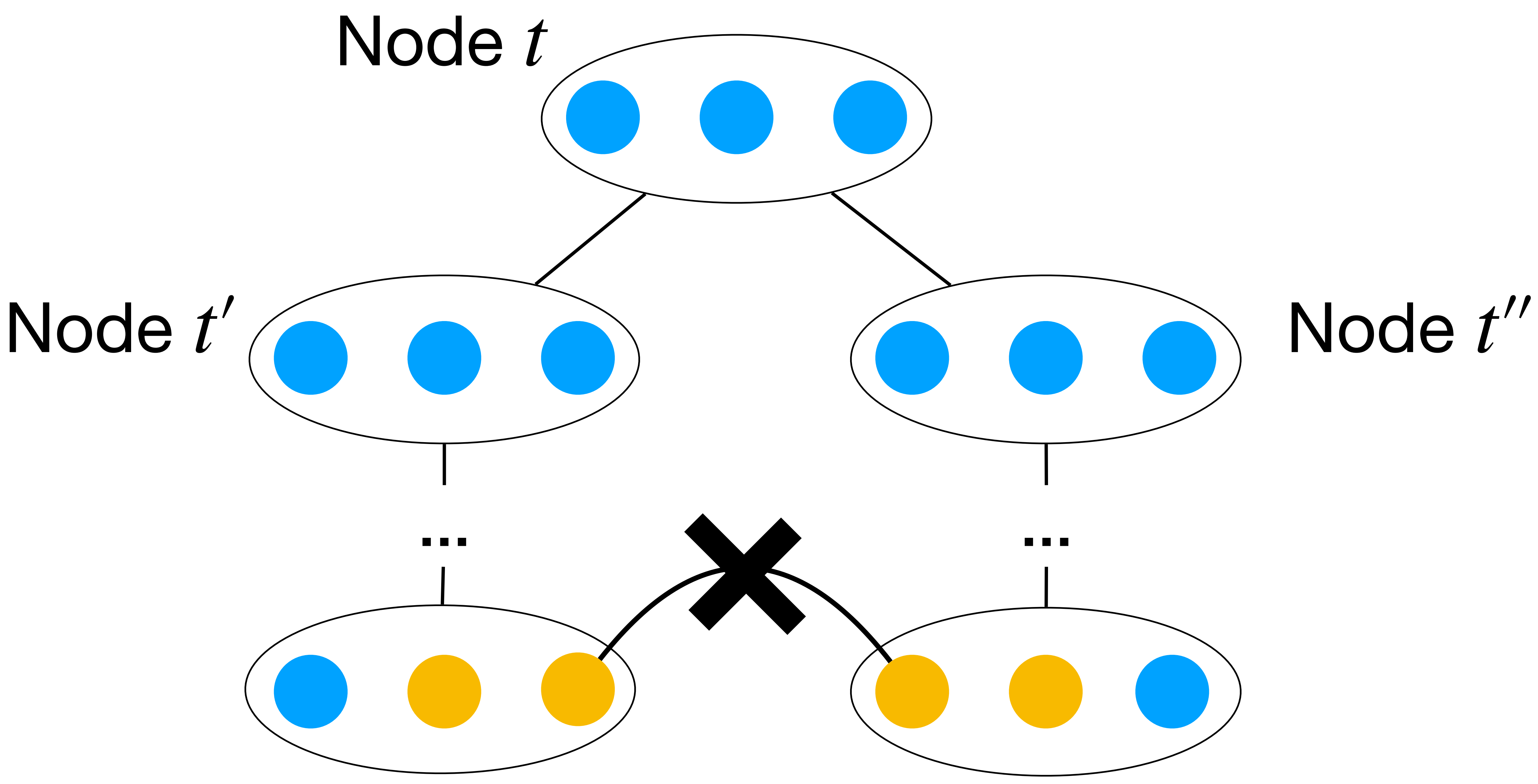
Join Nodes



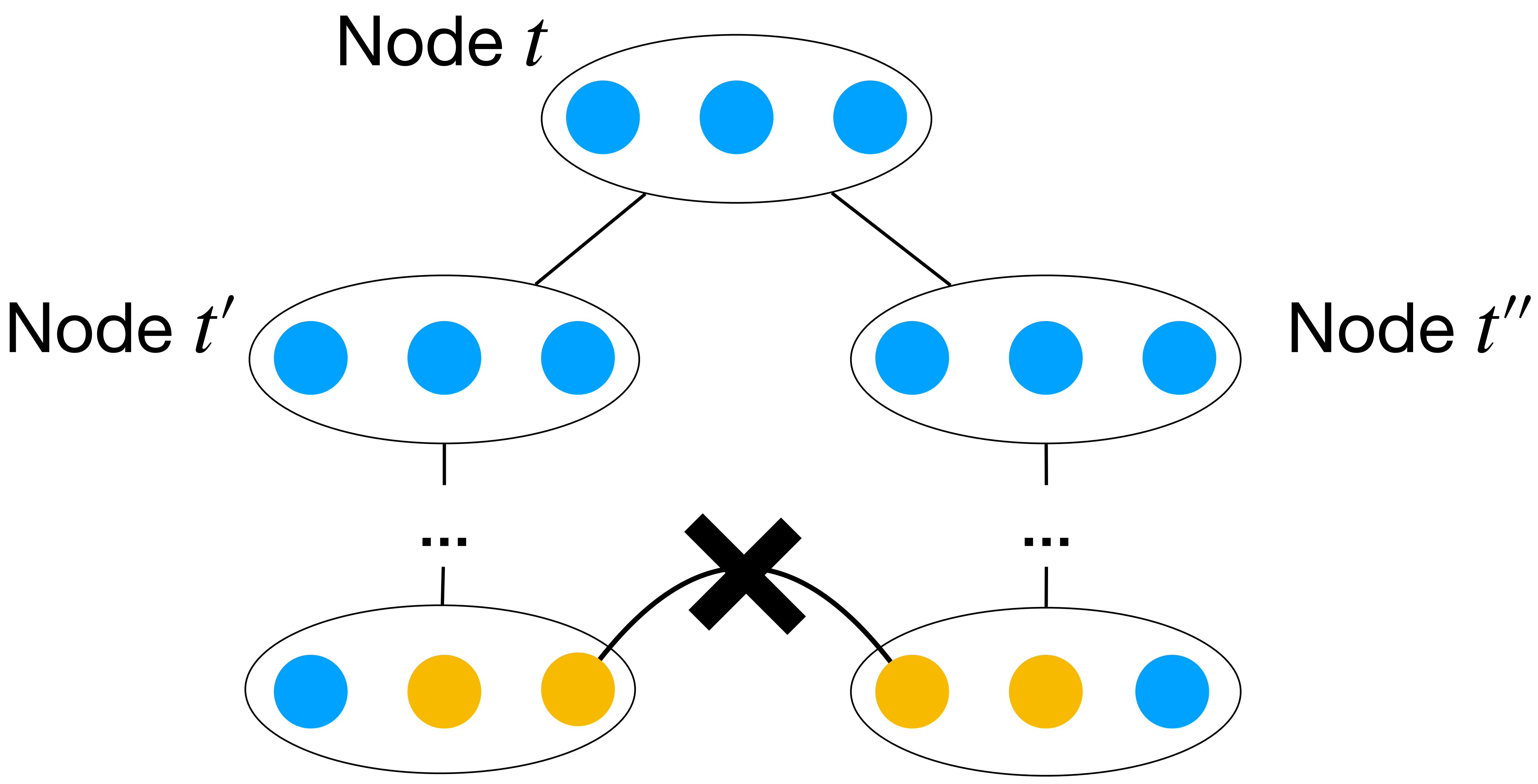
Join Nodes



Join Nodes

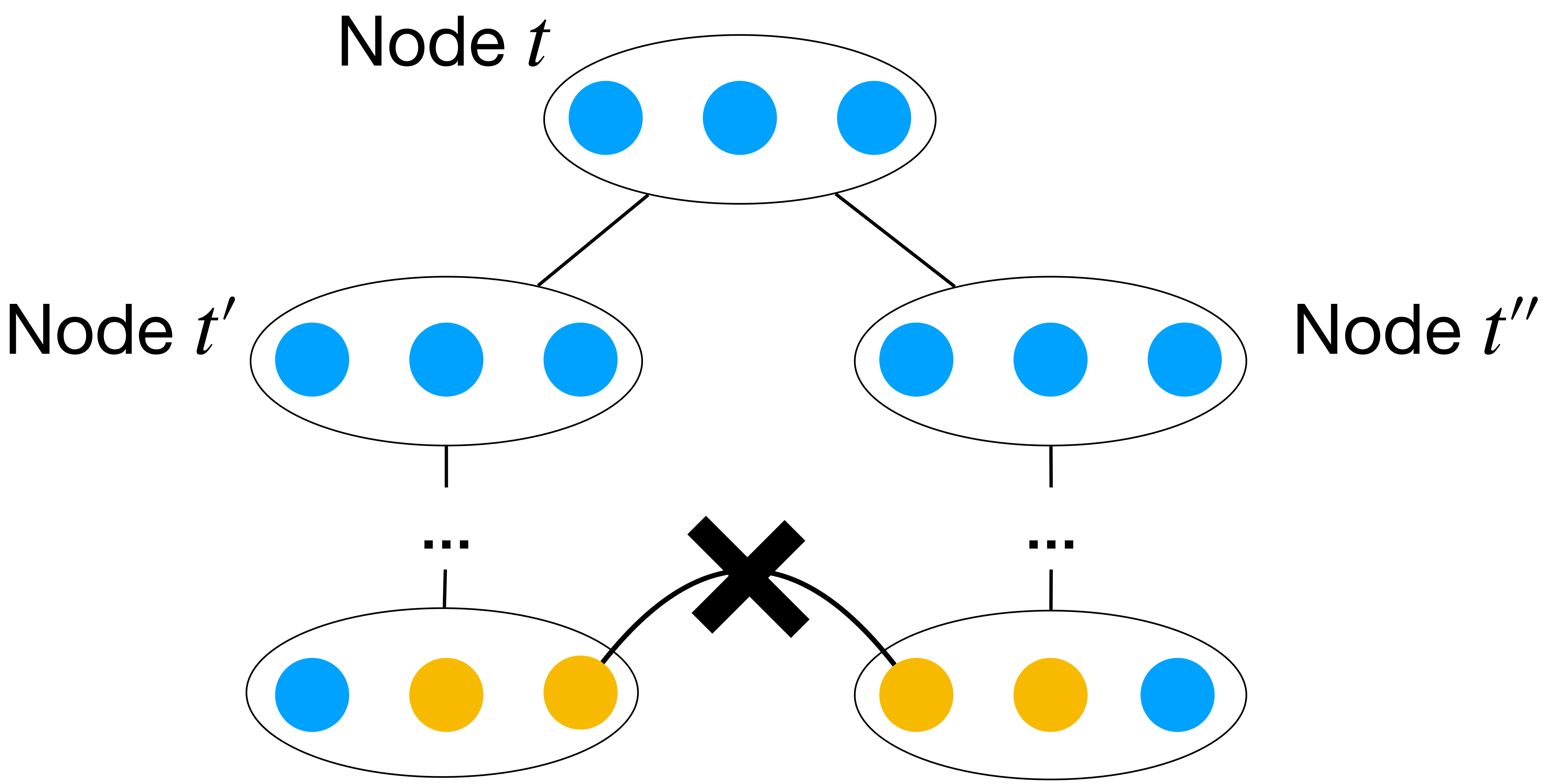


Join Nodes



for $S \subseteq \chi(t)$

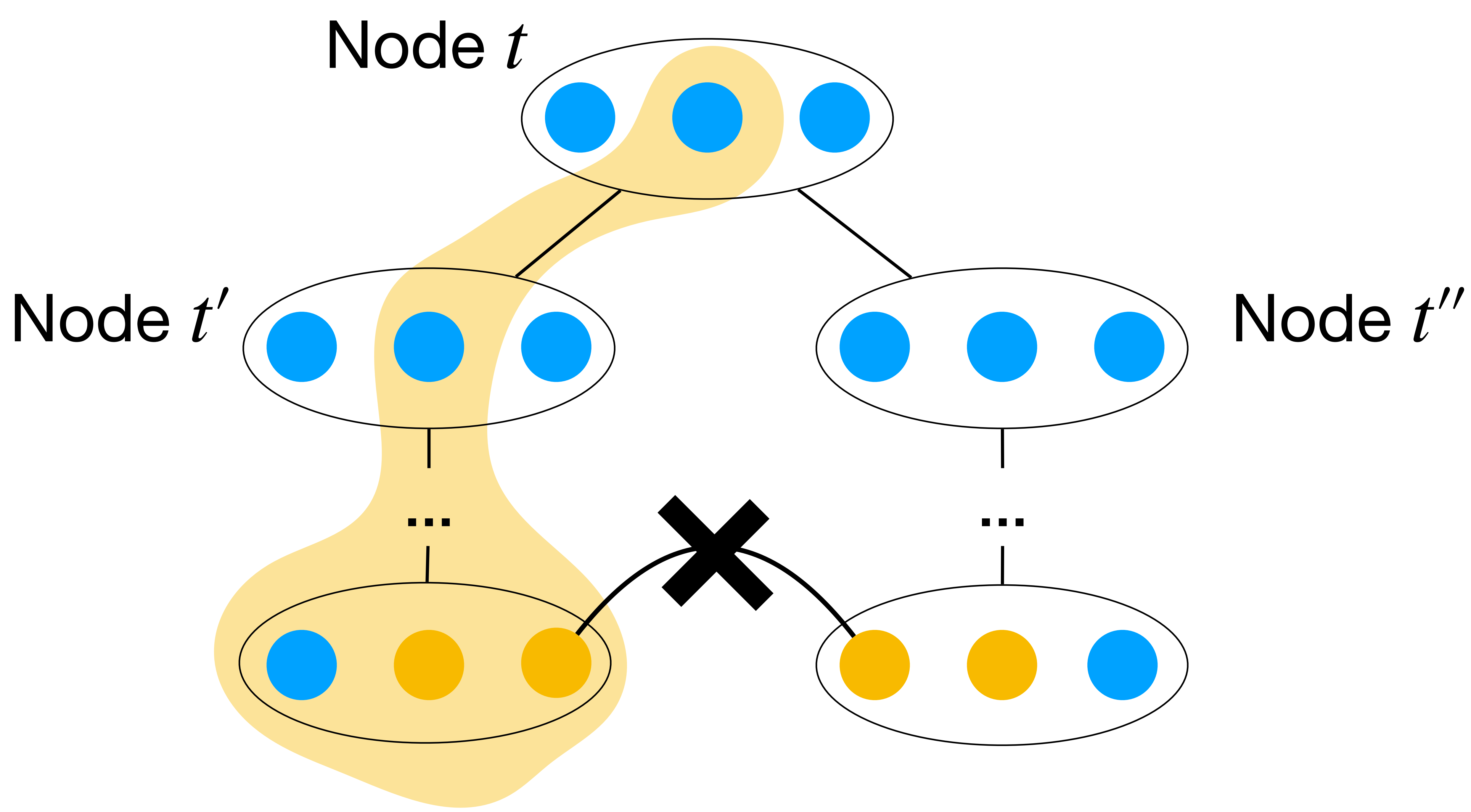
Join Nodes



for $S \subseteq \chi(t)$

$$n_t(S) := n_{t'}(S) + n_{t''}(S)$$

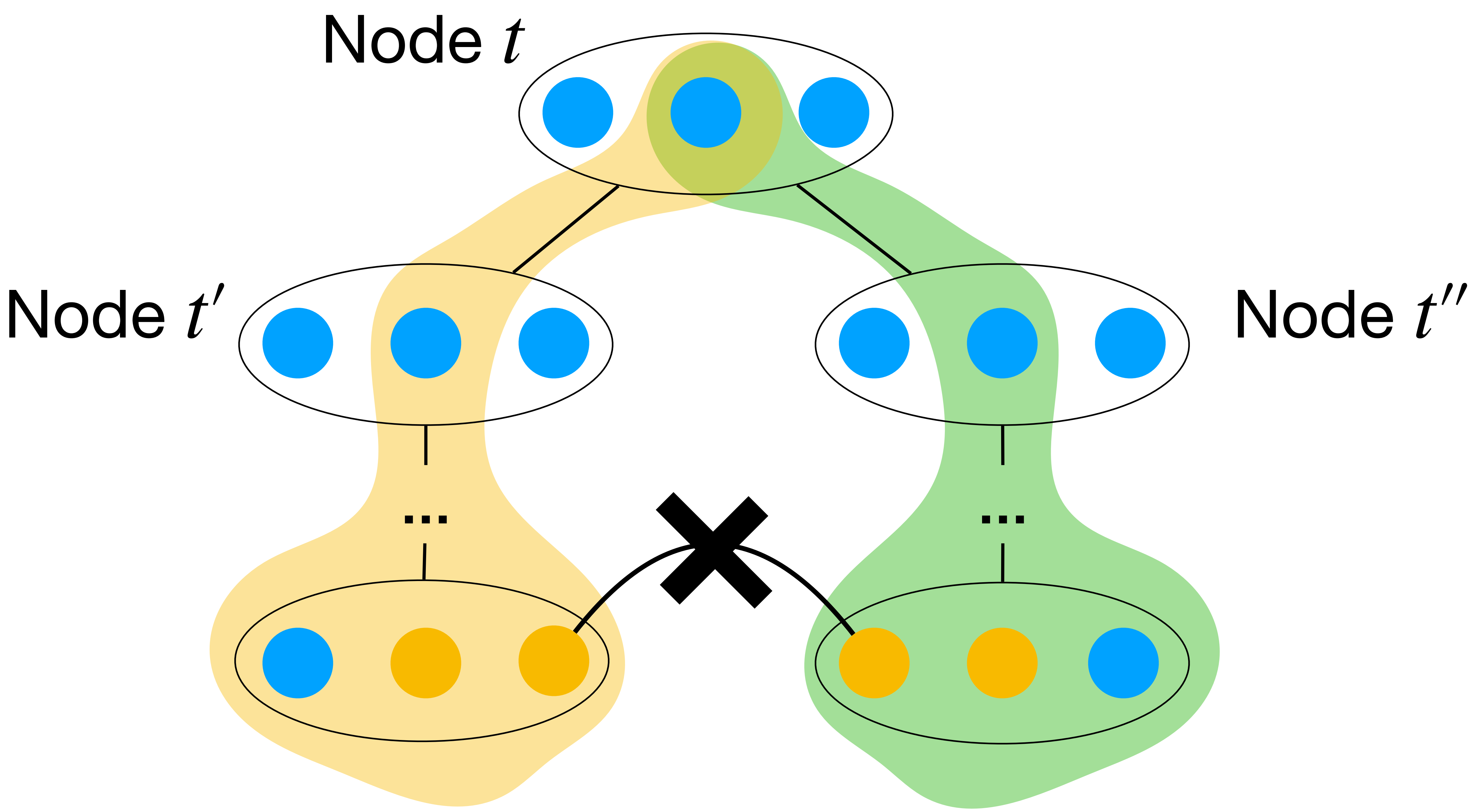
Join Nodes



for $S \subseteq \chi(t)$

$$n_t(S) := n_{t'}(S) + n_{t''}(S)$$

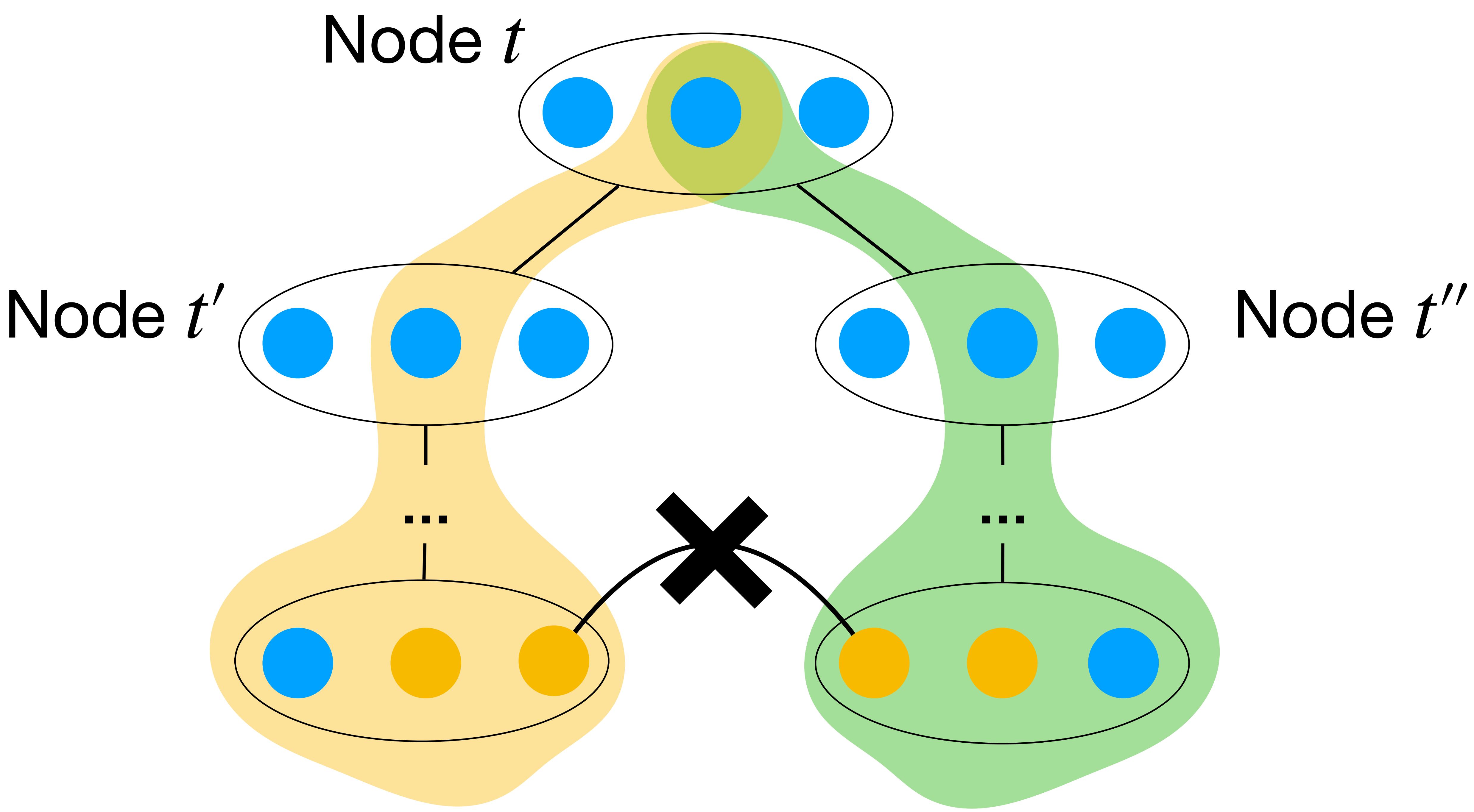
Join Nodes



for $S \subseteq \chi(t)$

$$n_t(S) := n_{t'}(S) + n_{t''}(S)$$

Join Nodes



for $S \subseteq \chi(t)$

$$n_t(S) := n_{t'}(S) + n_{t''}(S) - |S|$$

The Algorithm

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$$n_t(U) := |U|$$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $n_t(U) := n_{t'}(U)$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $n_t(U) := n_{t'}(U)$

if $v \in U$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $n_t(U) := n_{t'}(U)$

if $v \in U$ $n_t(U) := n_{t'}(U \backslash \{v\}) + 1$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $n_t(U) := n_{t'}(U)$

if $v \in U$ $n_t(U) := n_{t'}(U \backslash \{v\}) + 1$ if U is an IS

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $n_t(U) := n_{t'}(U)$

if $v \in U$ $n_t(U) := n_{t'}(U \setminus \{v\}) + 1$ if U is an IS

$n_t(U) := 0$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $n_t(U) := n_{t'}(U)$

if $v \in U$ $n_t(U) := n_{t'}(U \backslash \{v\}) + 1$ if U is an IS

$n_t(U) := 0$ otherwise

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $n_t(U) := n_{t'}(U)$

if $v \in U$ $n_t(U) := n_{t'}(U \backslash \{v\}) + 1$ if U is an IS

$n_t(U) := 0$ otherwise

Forget Nodes

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $n_t(U) := n_{t'}(U)$

if $v \in U$ $n_t(U) := n_{t'}(U \backslash \{v\}) + 1$ if U is an IS

$n_t(U) := 0$ otherwise

Forget Nodes

for each $U \subseteq \chi(t)$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $n_t(U) := n_{t'}(U)$

if $v \in U$ $n_t(U) := n_{t'}(U \backslash \{v\}) + 1$ if U is an IS

$n_t(U) := 0$ otherwise

Forget Nodes

for each $U \subseteq \chi(t)$

$$n_t(U) := \max(n_{t'}(U), n_{t'}(U \cup \{v\}))$$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $\quad n_t(U) := n_{t'}(U)$

if $v \in U$ $\quad n_t(U) := n_{t'}(U \backslash \{v\}) + 1$ if U is an IS

$n_t(U) := 0$ otherwise

Forget Nodes

for each $U \subseteq \chi(t)$

$$n_t(U) := \max(n_{t'}(U), n_{t'}(U \cup \{v\}))$$

Join Nodes

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $n_t(U) := n_{t'}(U)$

if $v \in U$ $n_t(U) := n_{t'}(U \backslash \{v\}) + 1$ if U is an IS

$n_t(U) := 0$ otherwise

Forget Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := \max(n_{t'}(U), n_{t'}(U \cup \{v\}))$

Join Nodes

for each $U \subseteq \chi(t)$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $n_t(U) := n_{t'}(U)$

if $v \in U$ $n_t(U) := n_{t'}(U \backslash \{v\}) + 1$ if U is an IS

$n_t(U) := 0$ otherwise

Forget Nodes

for each $U \subseteq \chi(t)$

$$n_t(U) := \max(n_{t'}(U), n_{t'}(U \cup \{v\}))$$

Join Nodes

for each $U \subseteq \chi(t)$

$$n_t(U) = n_{t'}(U) + n_{t''}(U) - |U|$$

The Algorithm

1. Compute a nice tree decomposition (T, χ) of G
2. For each $t \in T$ and $U \subseteq \chi(t)$, initialize $n_t(U)$
3. Update $n_t(U)$ by dynamic programming …
4. Output $\max\{n_r(U) \mid U \subseteq \chi(r)\}$

Leaf Nodes

for each $U \subseteq \chi(t)$

$n_t(U) := |U|$ if U is an IS

$n_t(U) := 0$ otherwise

Introduce Nodes

for each $U \subseteq \chi(t)$

if $v \notin U$ $n_t(U) := n_{t'}(U)$

if $v \in U$ $n_t(U) := n_{t'}(U \backslash \{v\}) + 1$ if U is an IS

$n_t(U) := 0$ otherwise

Forget Nodes

for each $U \subseteq \chi(t)$

$$n_t(U) := \max(n_{t'}(U), n_{t'}(U \cup \{v\}))$$

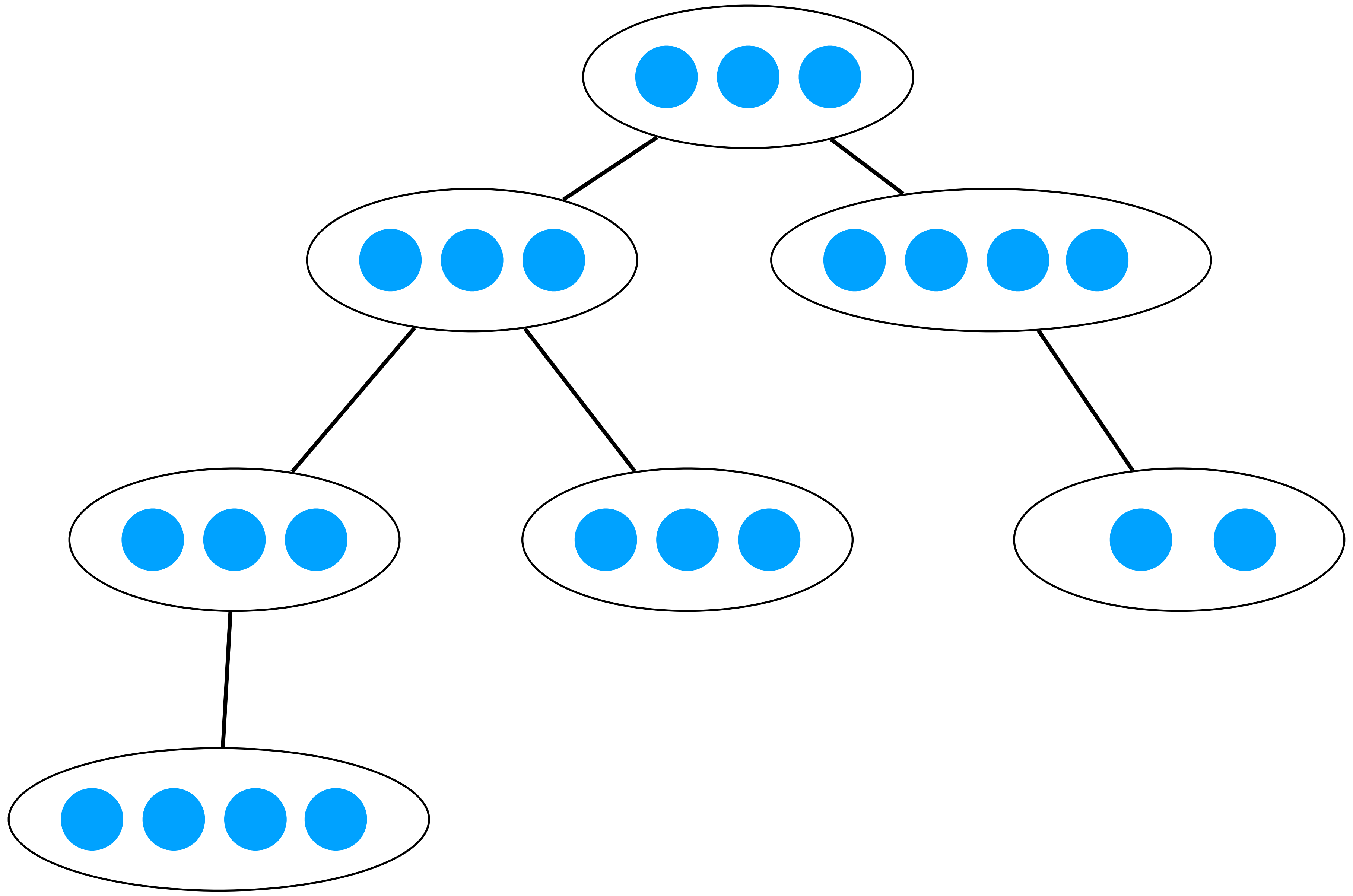
Join Nodes

for each $U \subseteq \chi(t)$

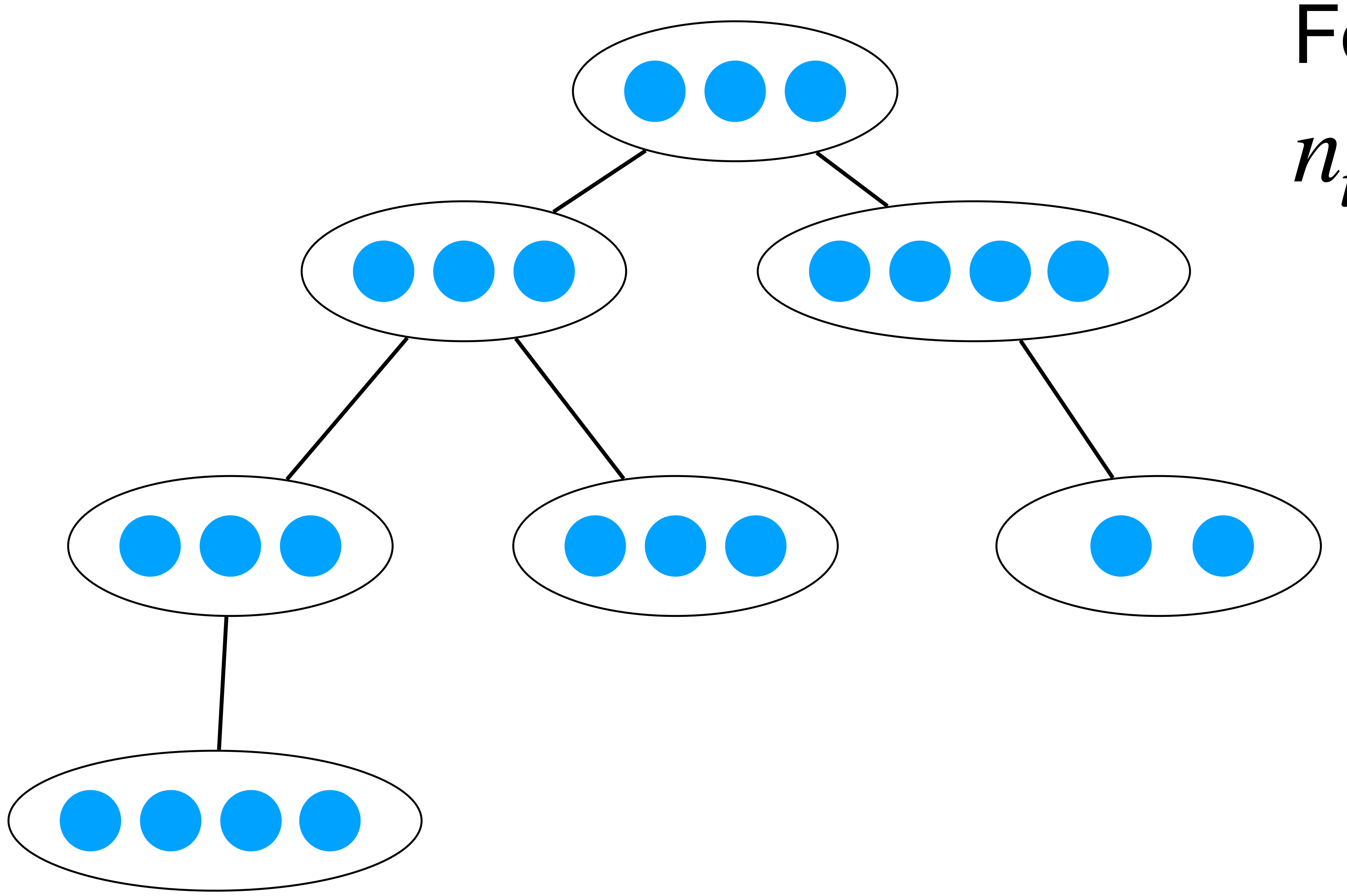
$$n_t(U) = n_{t'}(U) + n_{t''}(U) - |U|$$

Independent Set Parameterized by Treewidth

Independent Set Parameterized by Treewidth

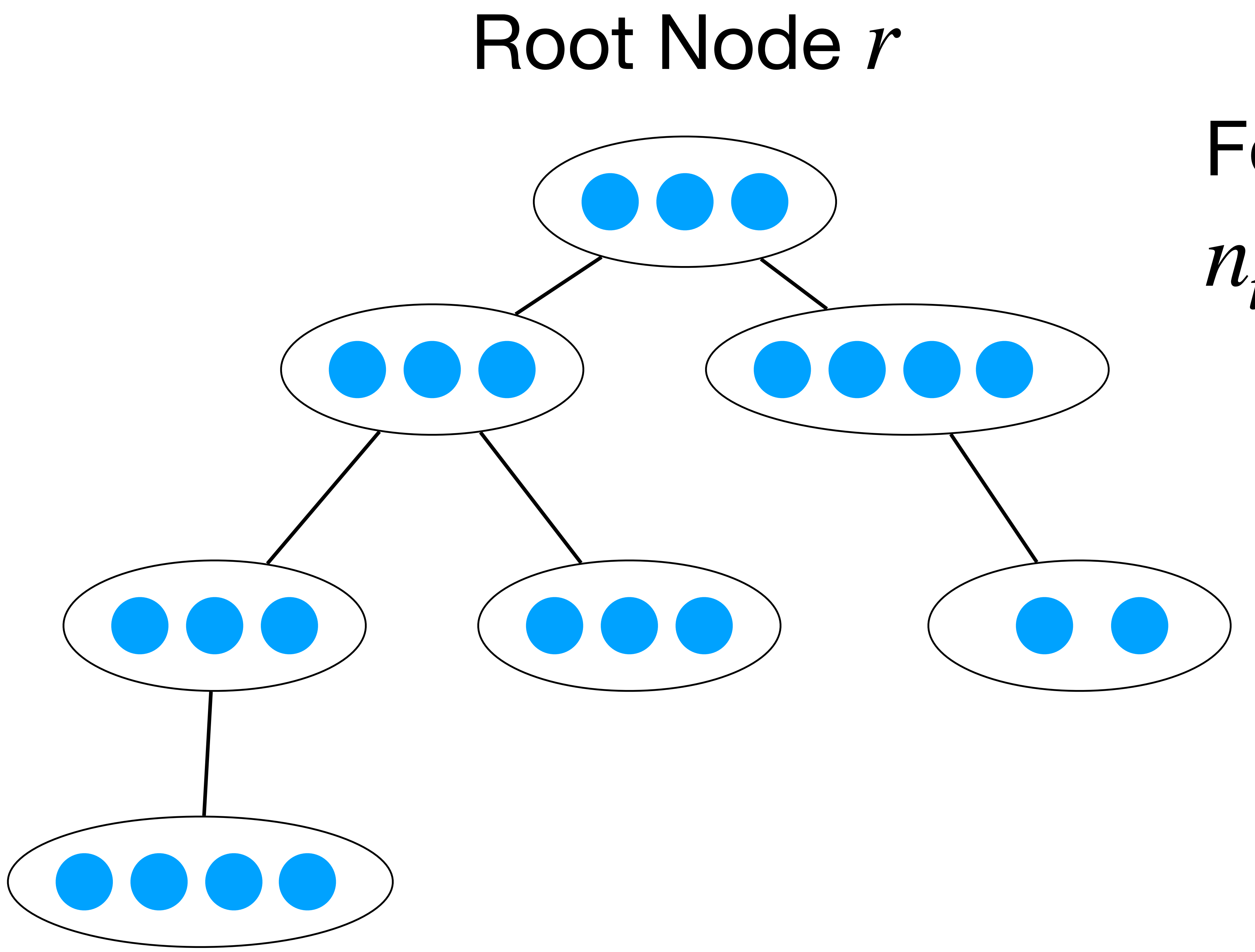


Independent Set Parameterized by Treewidth



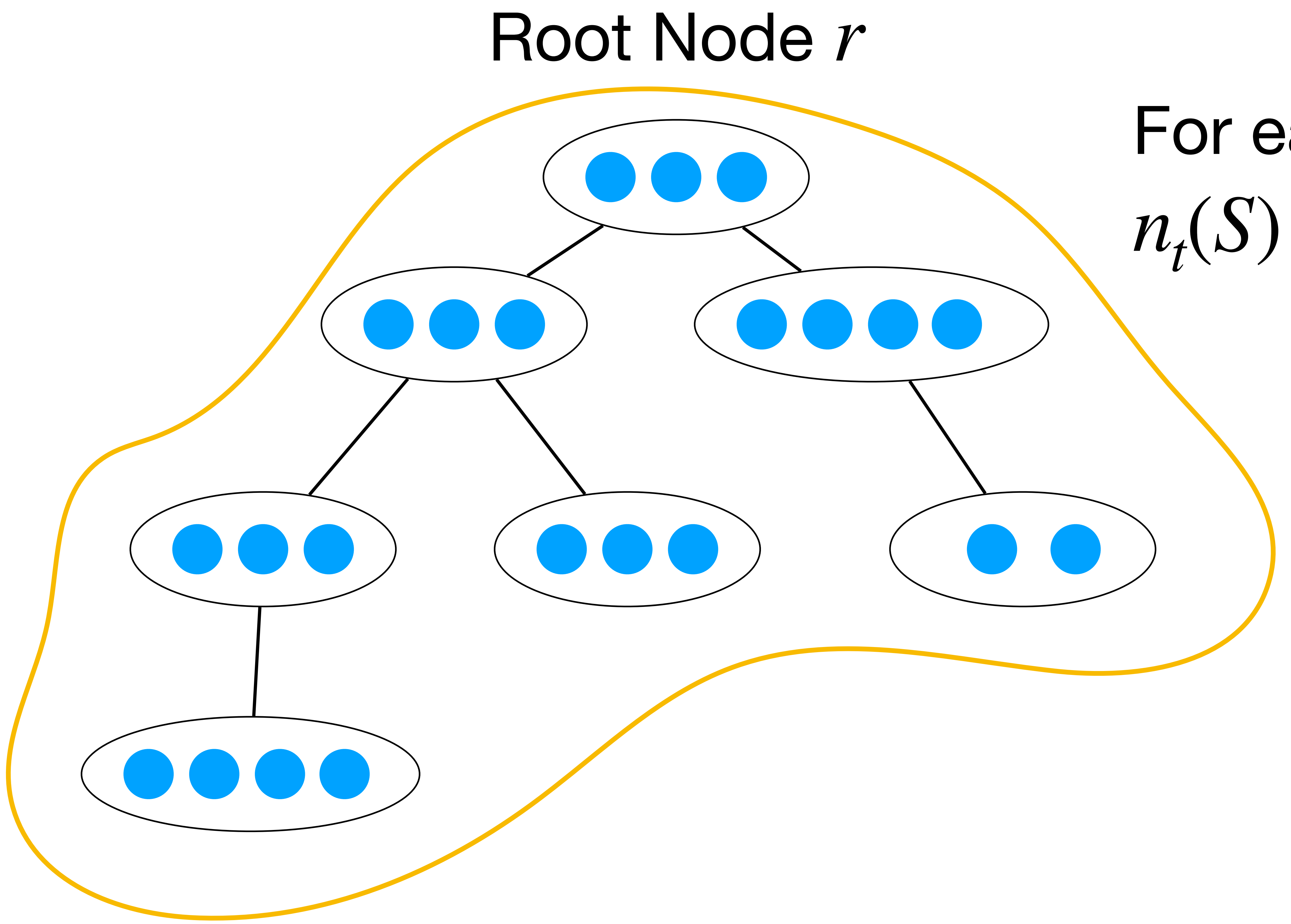
For each subset $S \subseteq \chi(t)$ of bag vertices,
$n_t(S)$ = | max. **IS** of G_t containing S |.

Independent Set Parameterized by Treewidth



For each subset $S \subseteq \chi(t)$ of bag vertices,
$n_t(S)$ = | max. **IS** of G_t containing S |.

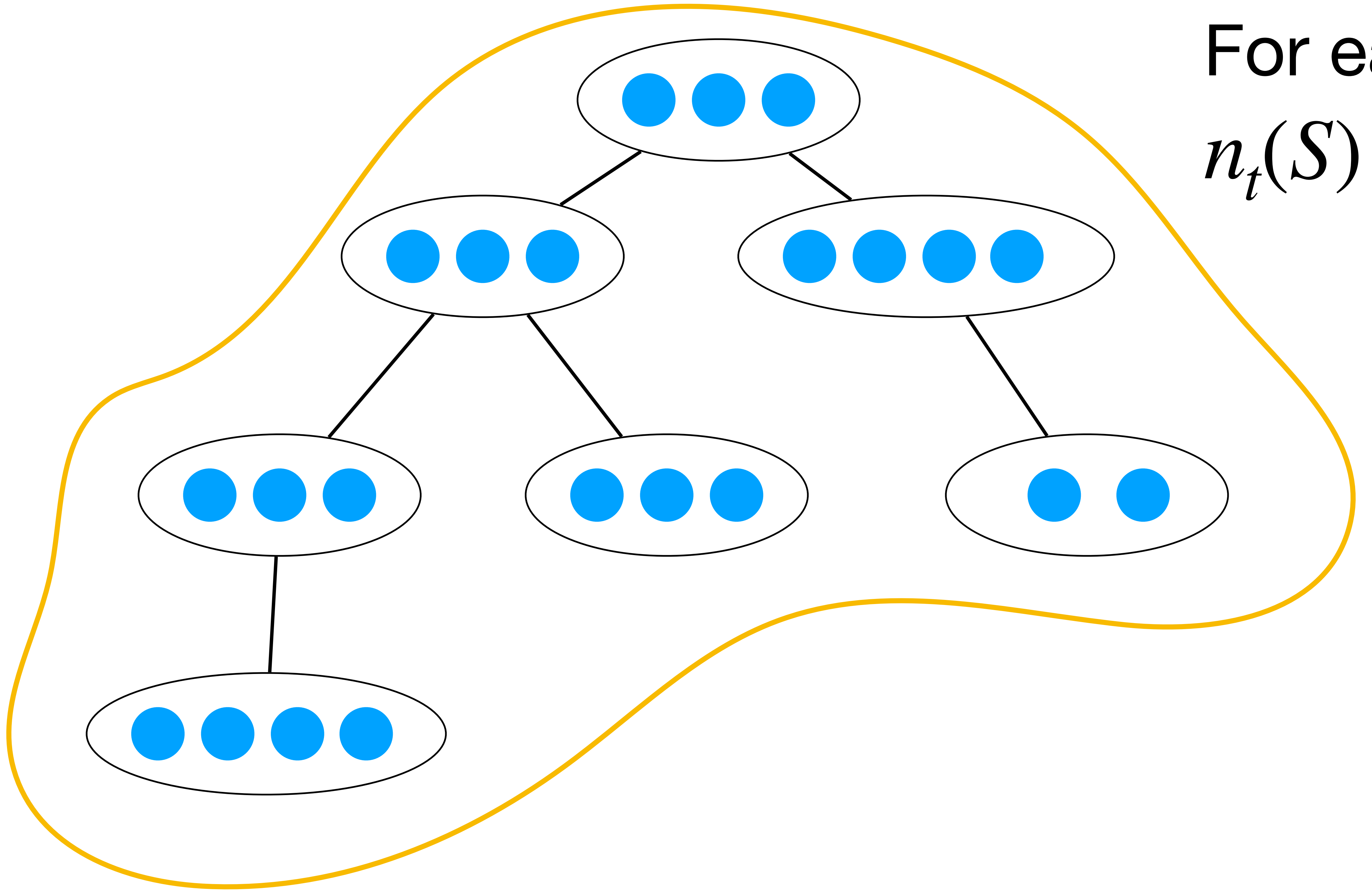
Independent Set Parameterized by Treewidth



For each subset $S \subseteq \chi(t)$ of bag vertices,
$n_t(S)$ = | max. **IS** of G_t containing S |.

Independent Set Parameterized by Treewidth

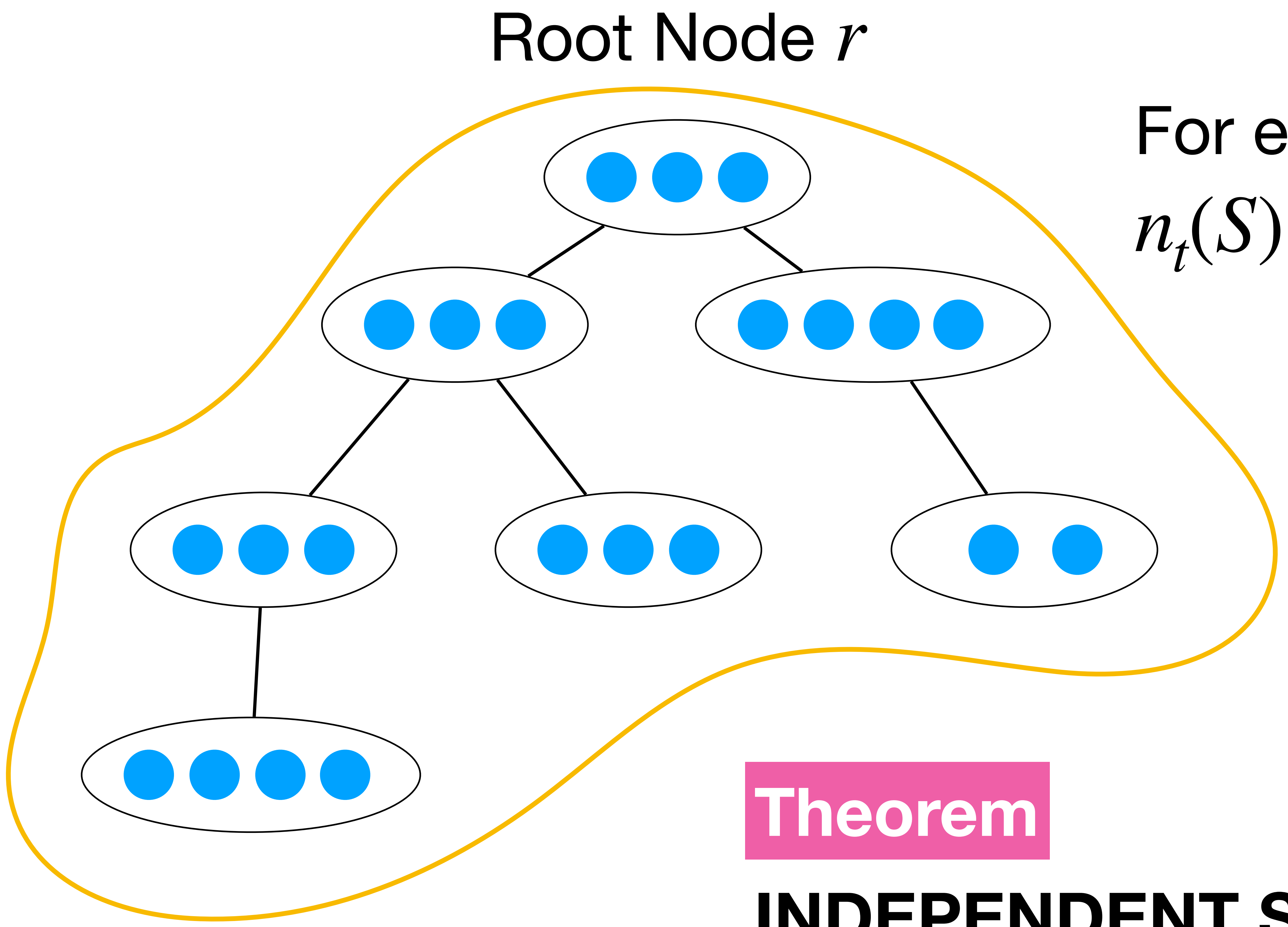
Root Node r



For each subset $S \subseteq \chi(t)$ of bag vertices,
$n_t(S)$ = | max. **IS** of G_t containing S |.

$$G_r = G$$

Independent Set Parameterized by Treewidth



For each subset $S \subseteq \chi(t)$ of bag vertices,
$n_t(S)$ = | max. **IS** of G_t containing S |.

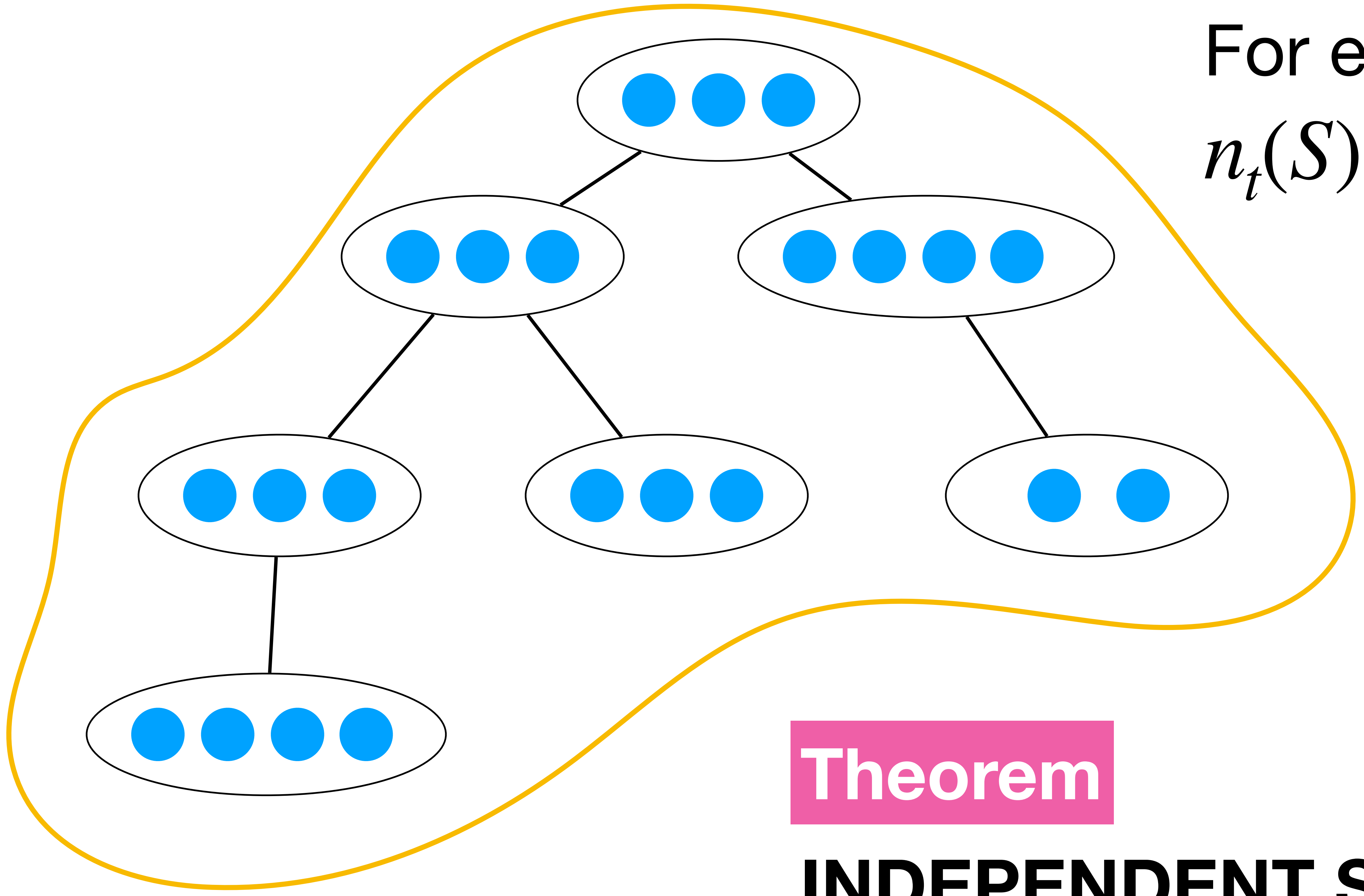
$$G_r = G$$

Theorem

INDEPENDENT SET is **FPT** parameterized by the treewidth of the input graph.*

Independent Set Parameterized by Treewidth

Root Node r



For each subset $S \subseteq \chi(t)$ of bag vertices,
$n_t(S)$ = | max. **IS** of G_t containing S |.

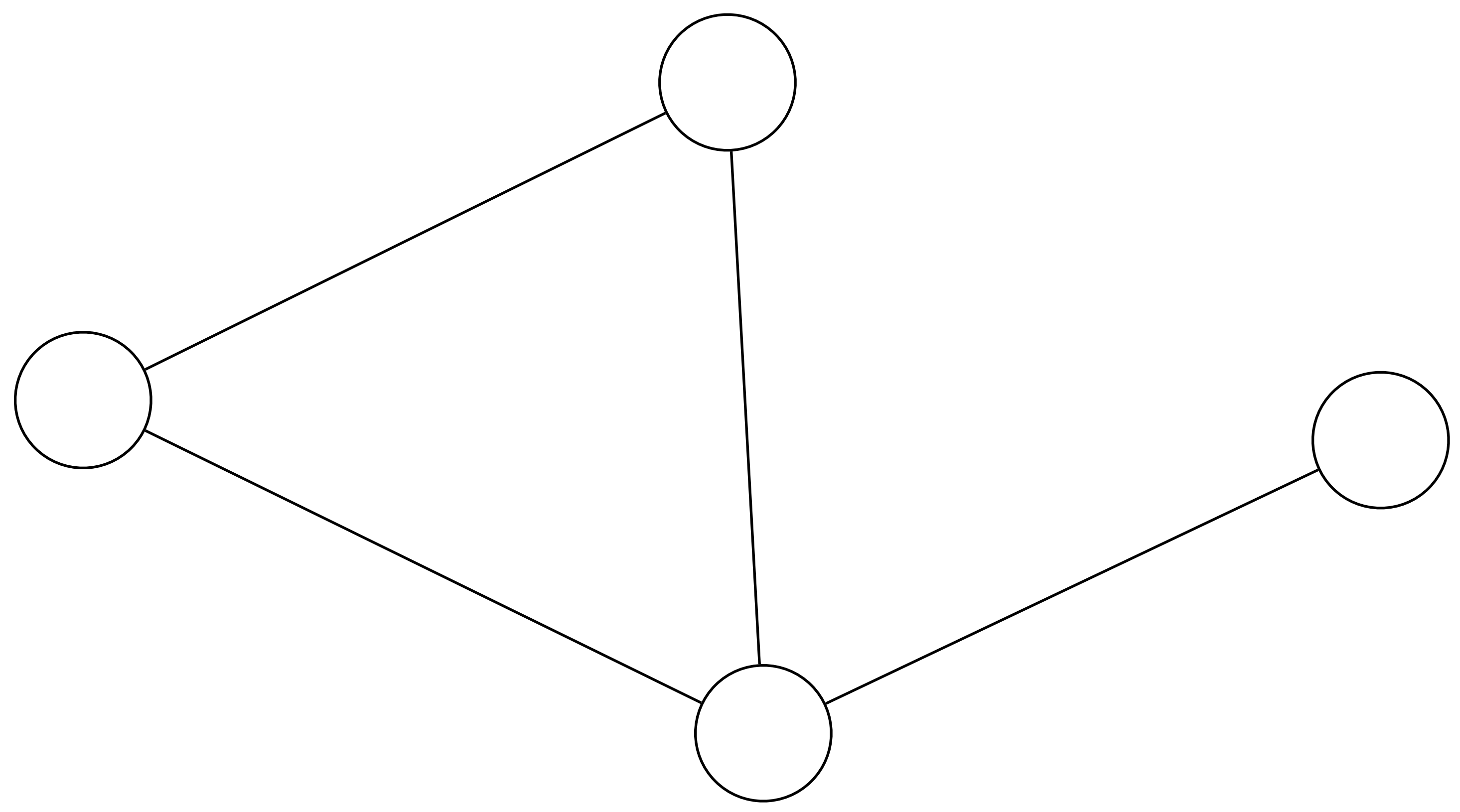
$$G_r = G$$

Theorem

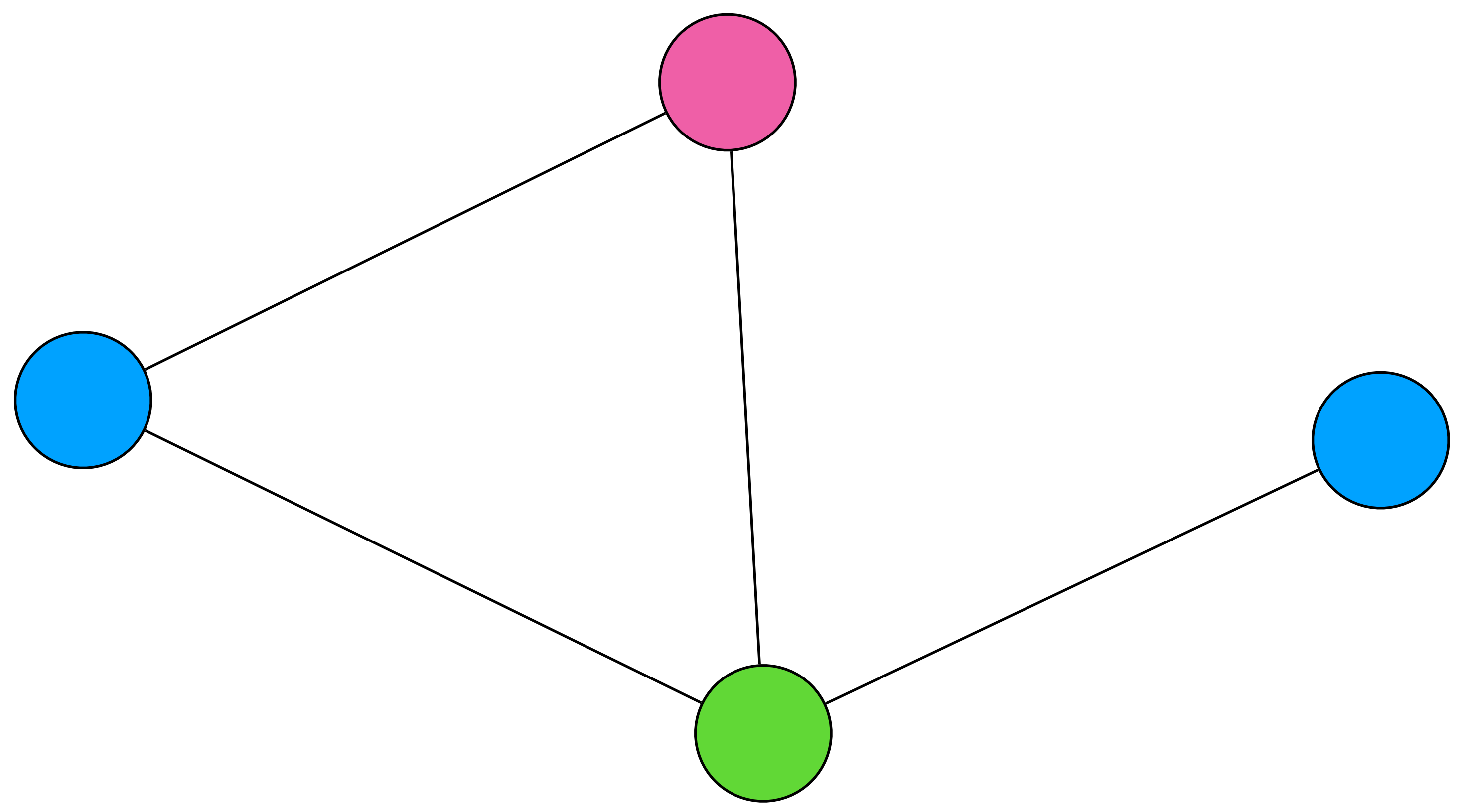
INDEPENDENT SET is **FPT** parameterized by the treewidth of the input graph.*

*If we can compute a nice tree decomposition of width $g(k)$ in FPT time.

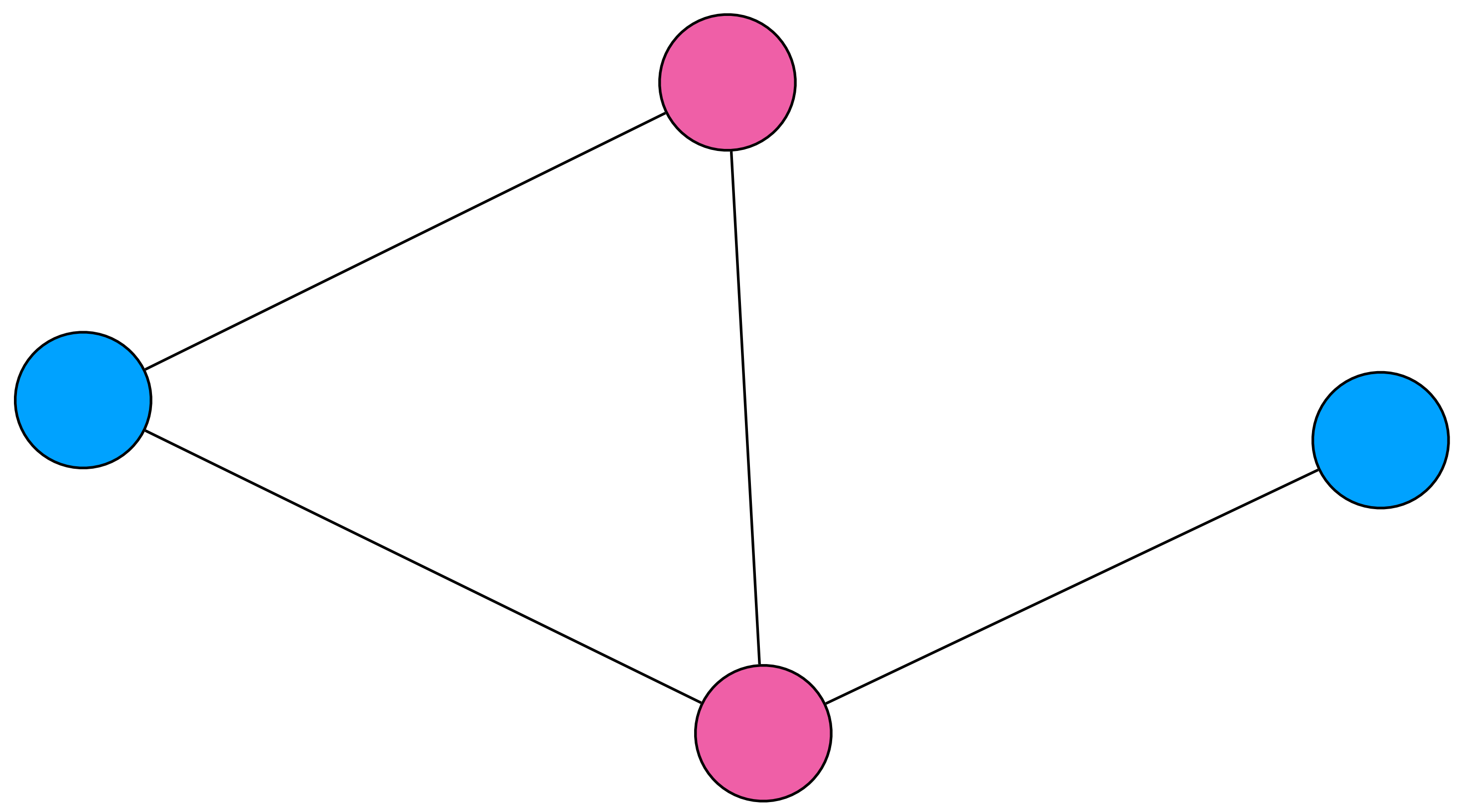
3-Colorability



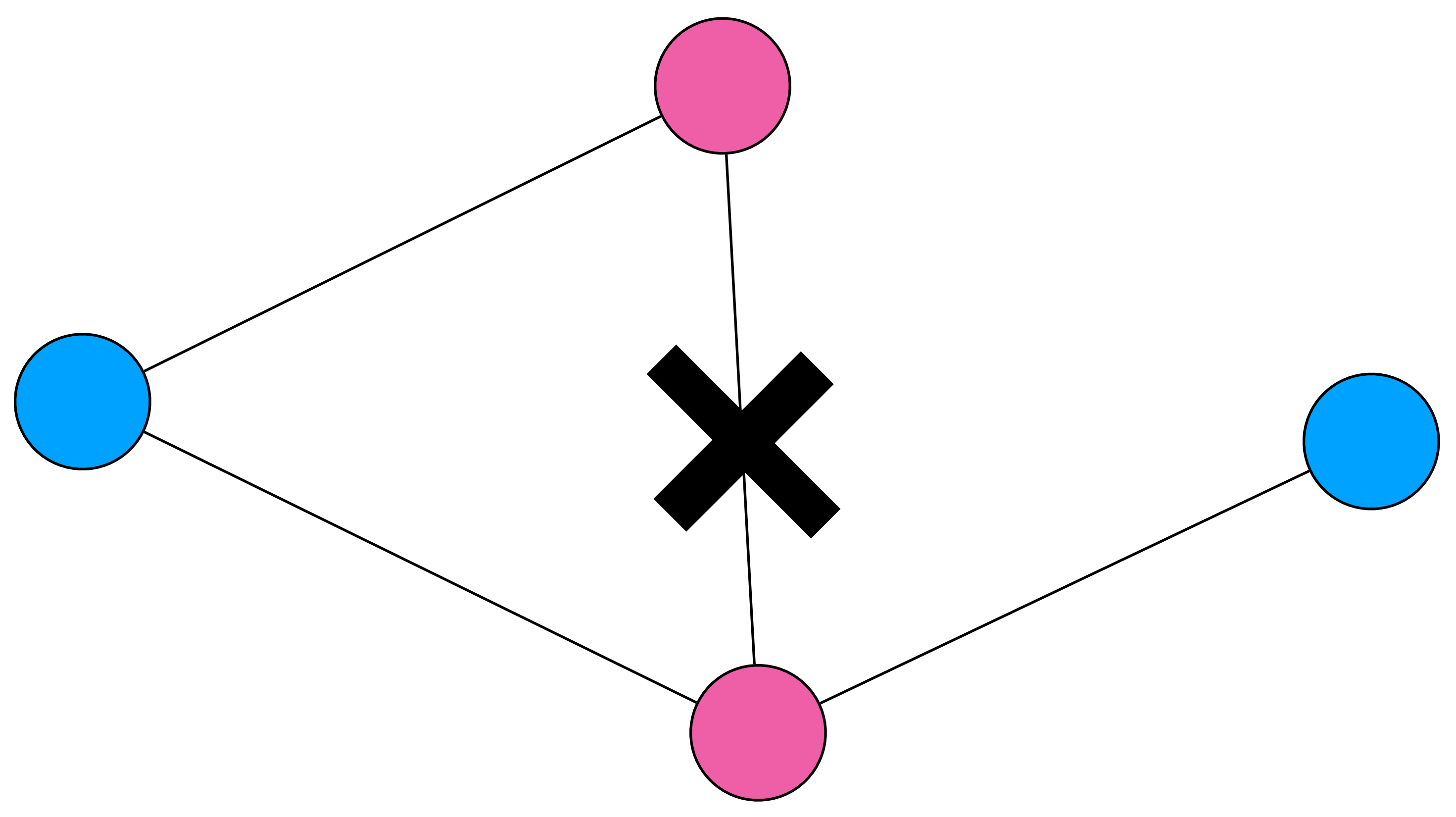
3-Colorability



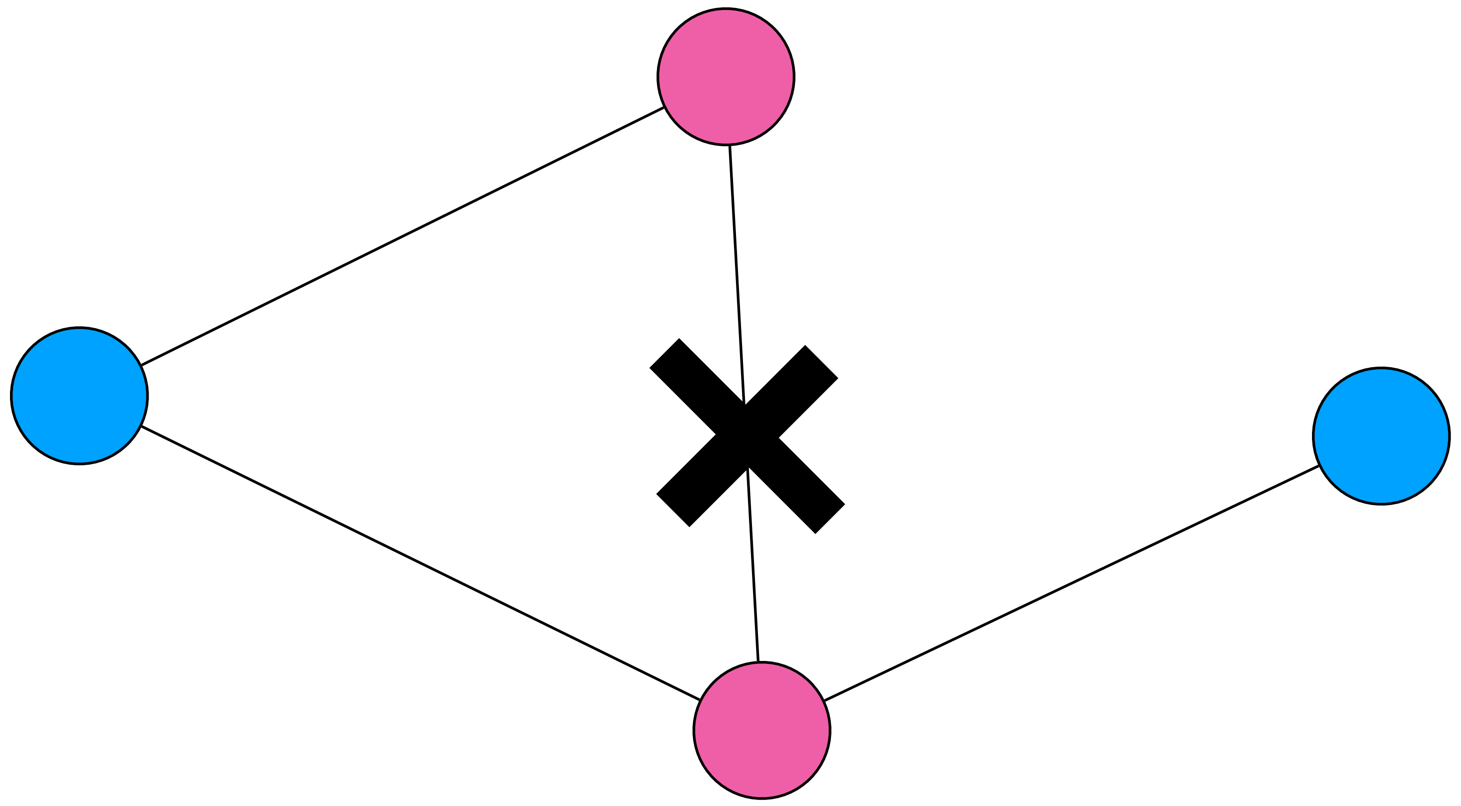
3-Colorability



3-Colorability



3-Colorability

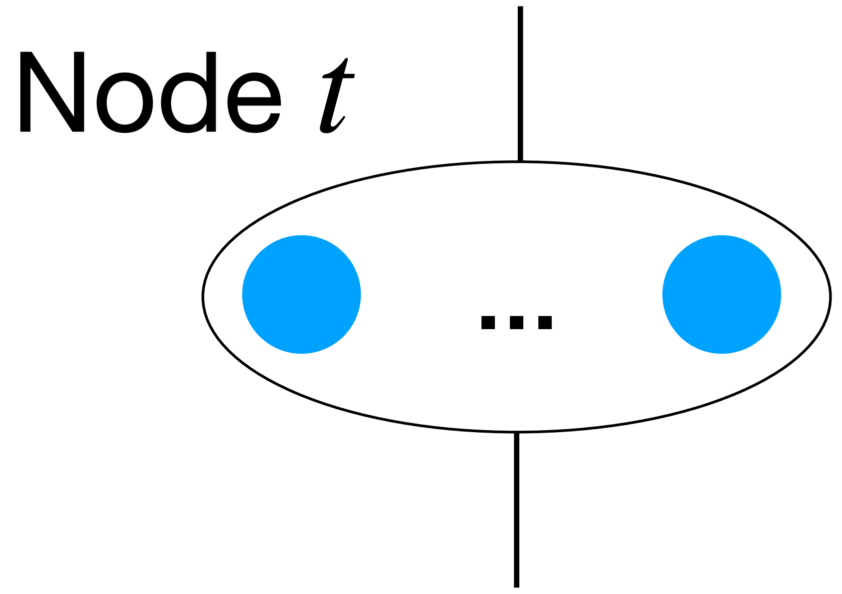


3-COLORABILITY

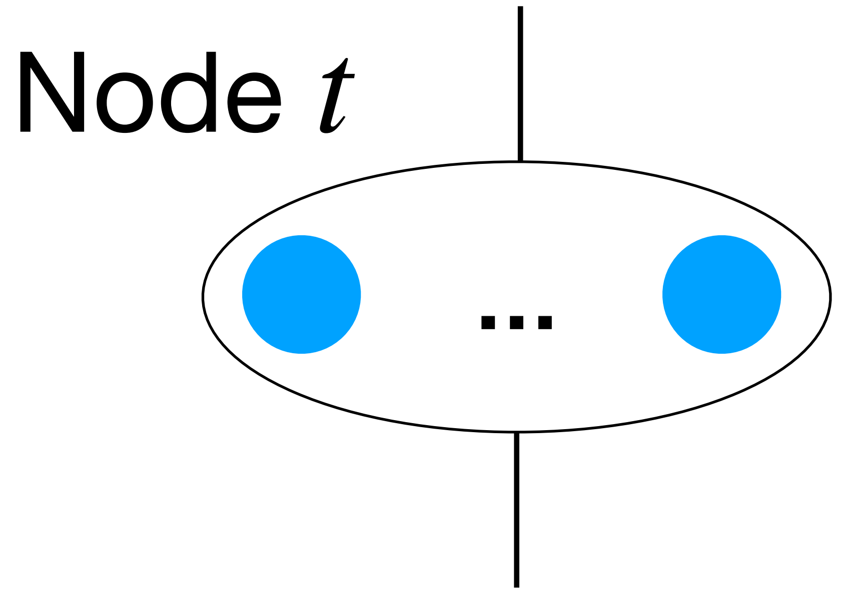
Input: A graph G.

Question: Does G have a vertex-coloring with 3 colors?

3-Colorability on Tree Decompositions

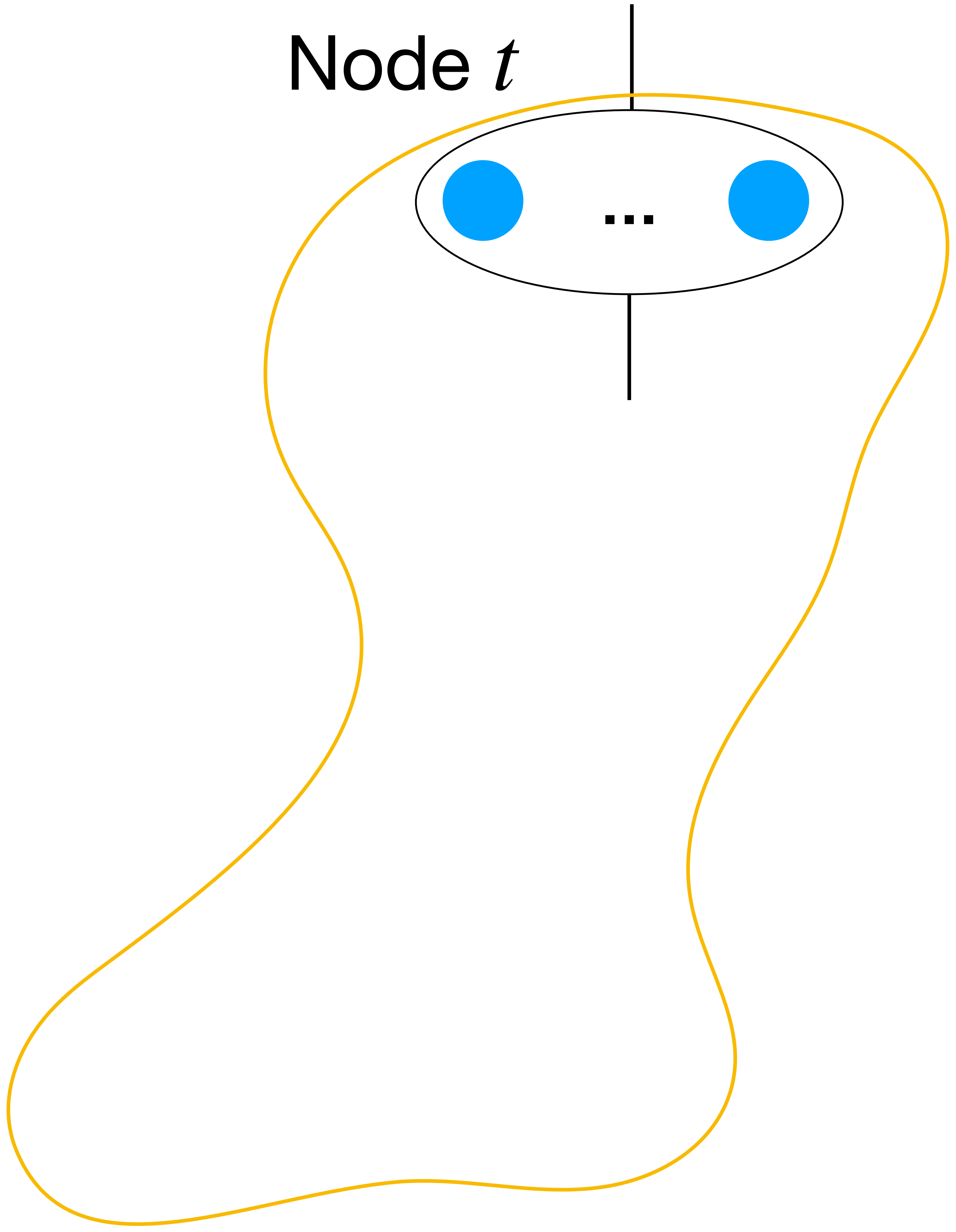


3-Colorability on Tree Decompositions



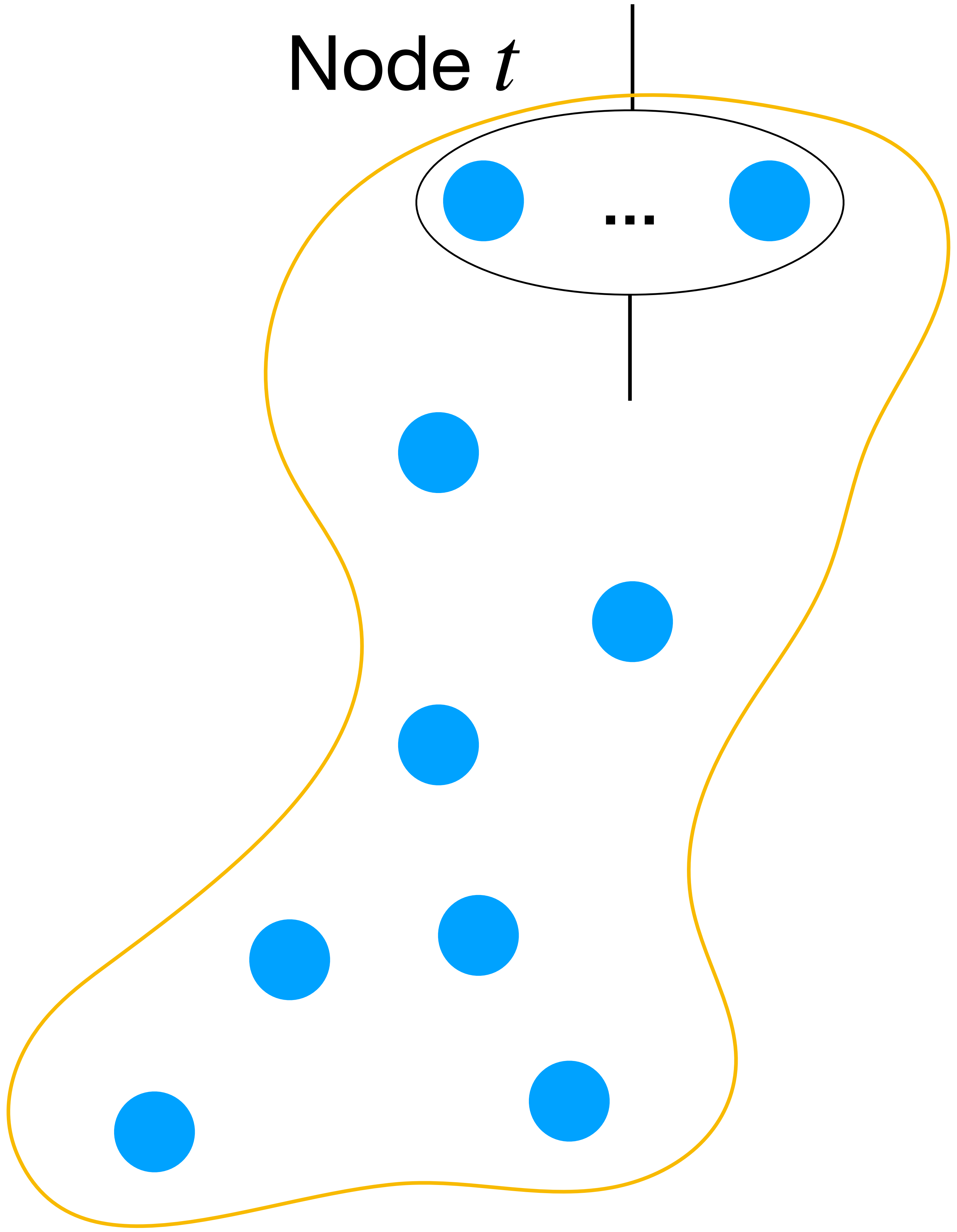
for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

3-Colorability on Tree Decompositions



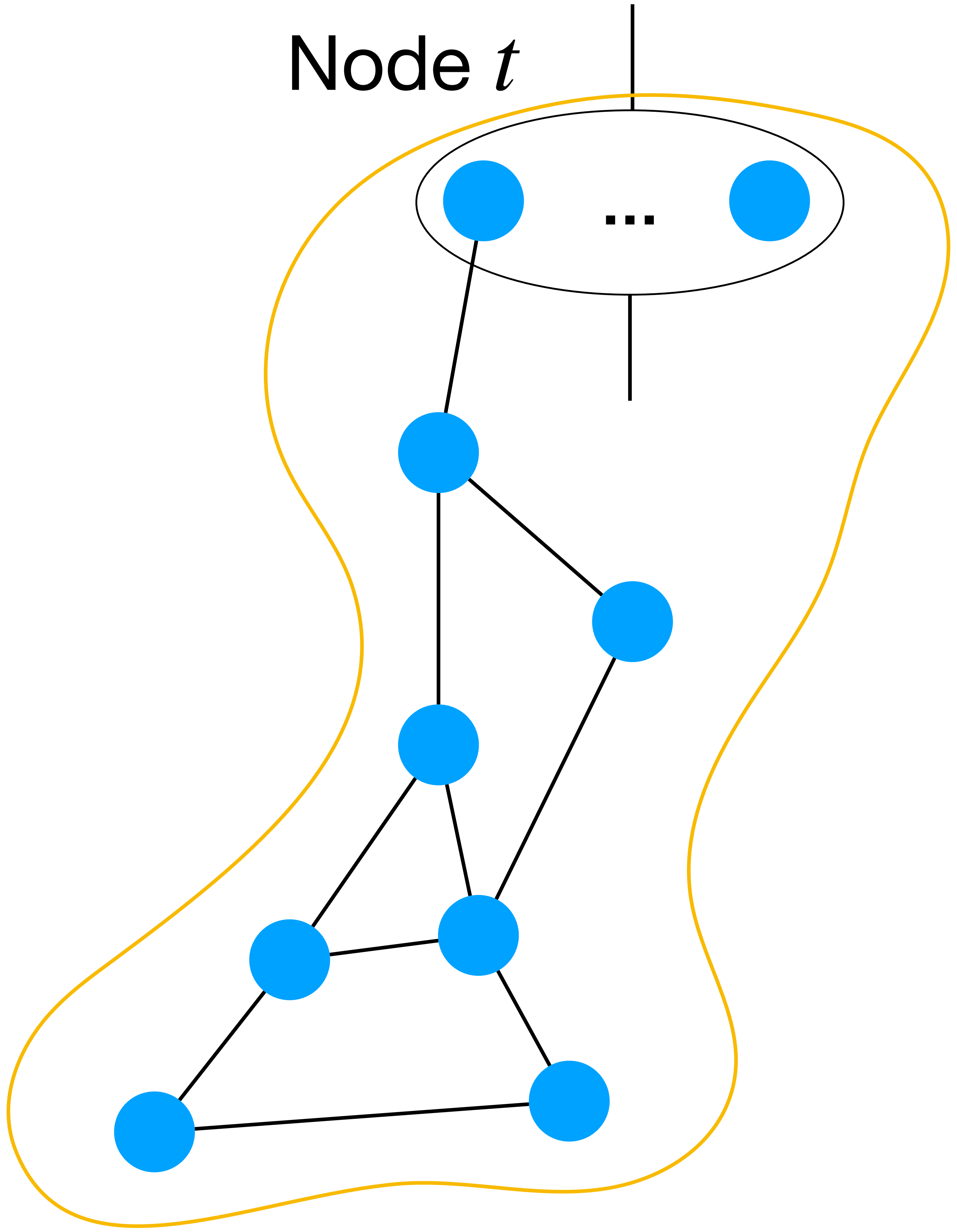
for $\sigma : \chi(t) \to \{red, green, blue\}$

3-Colorability on Tree Decompositions



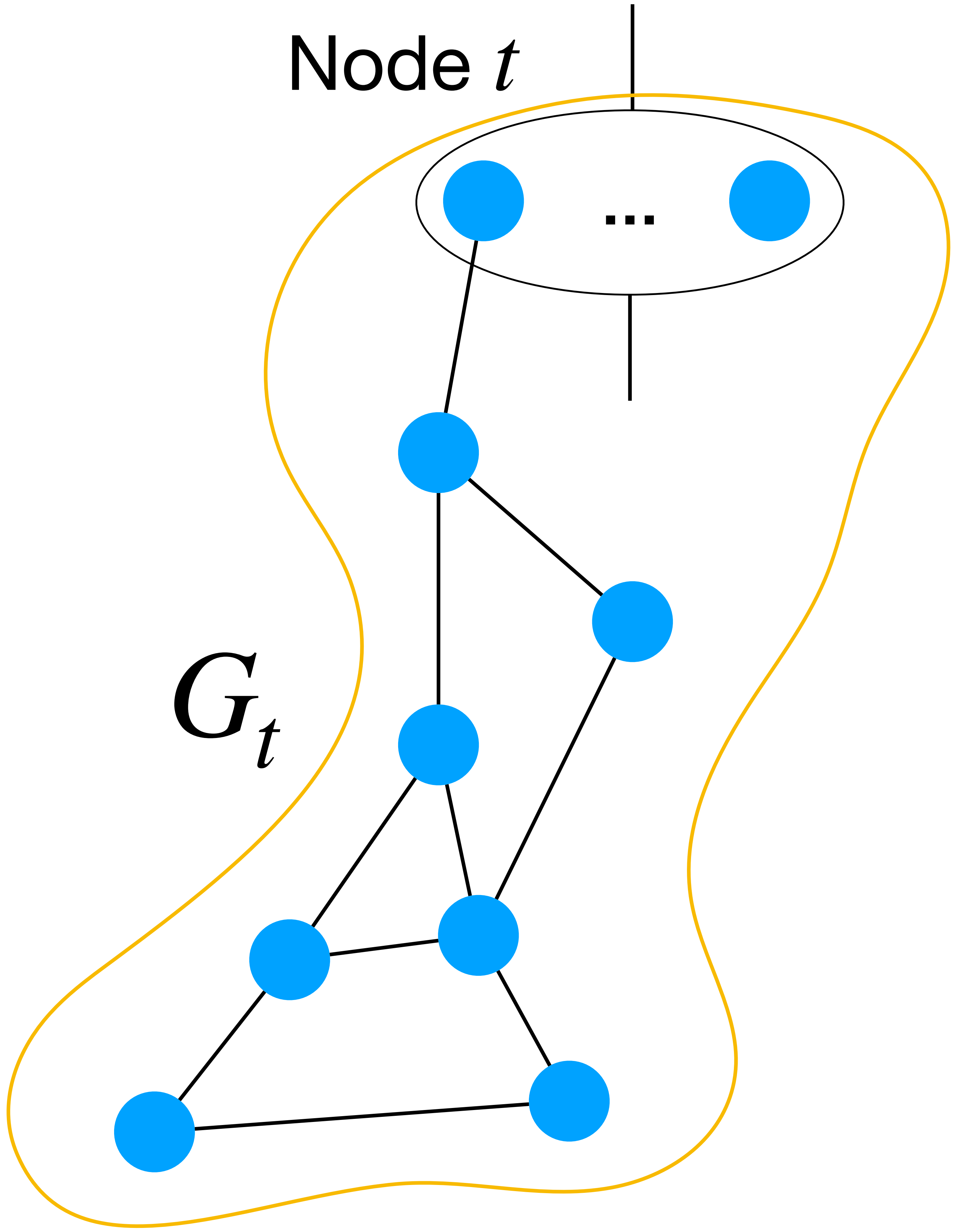
for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

3-Colorability on Tree Decompositions



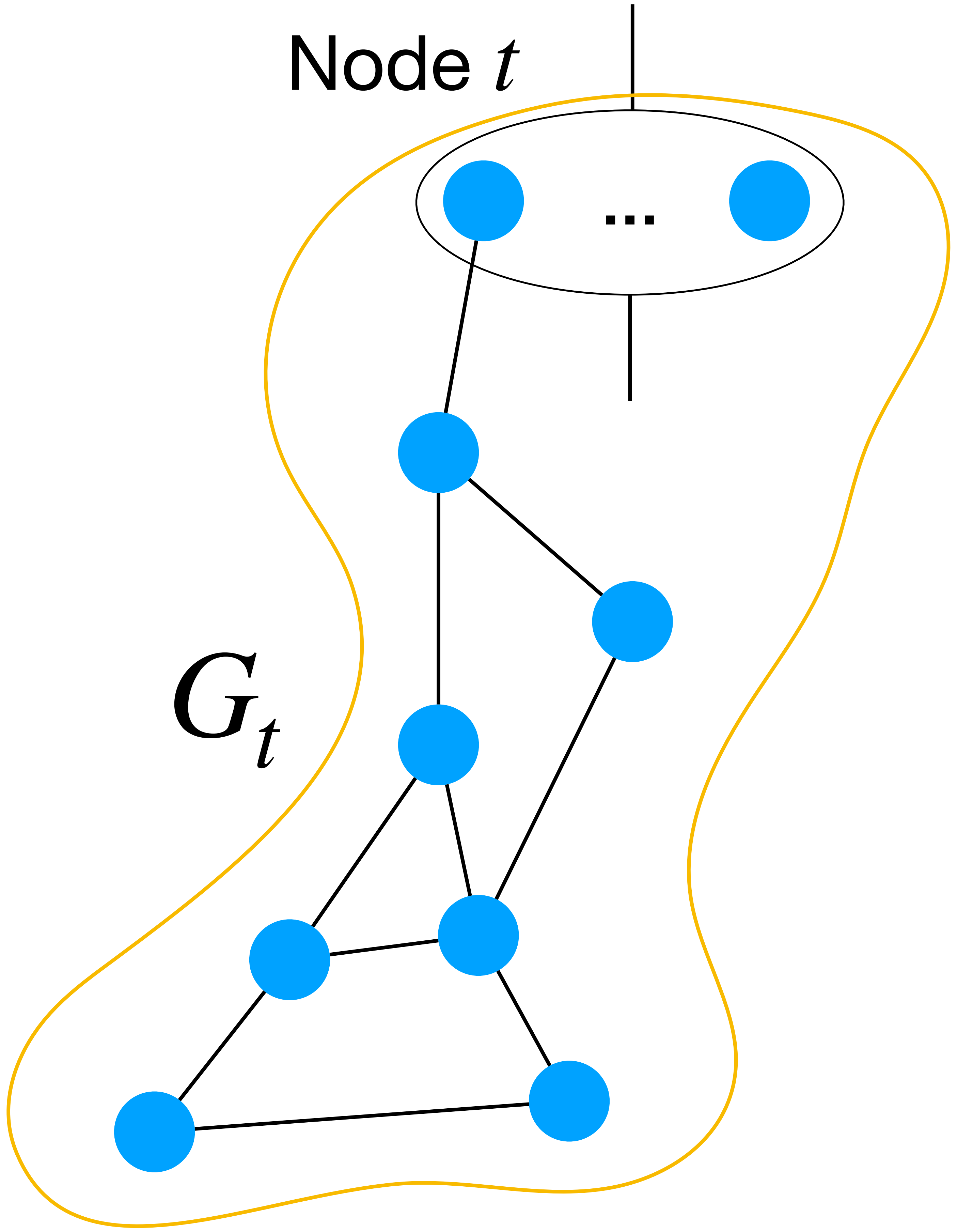
for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

3-Colorability on Tree Decompositions



for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

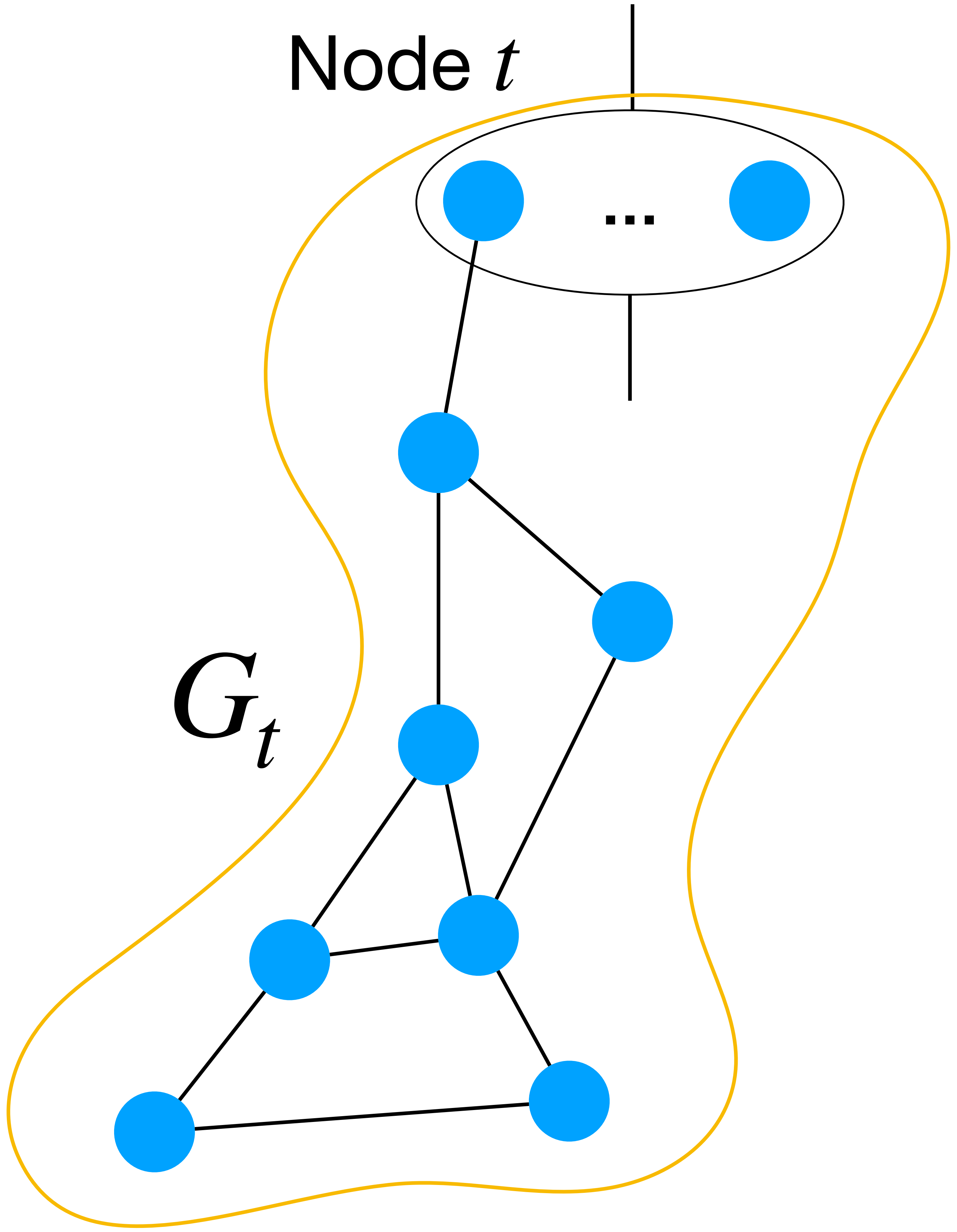
3-Colorability on Tree Decompositions



for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

$$n_t(\sigma) =$$

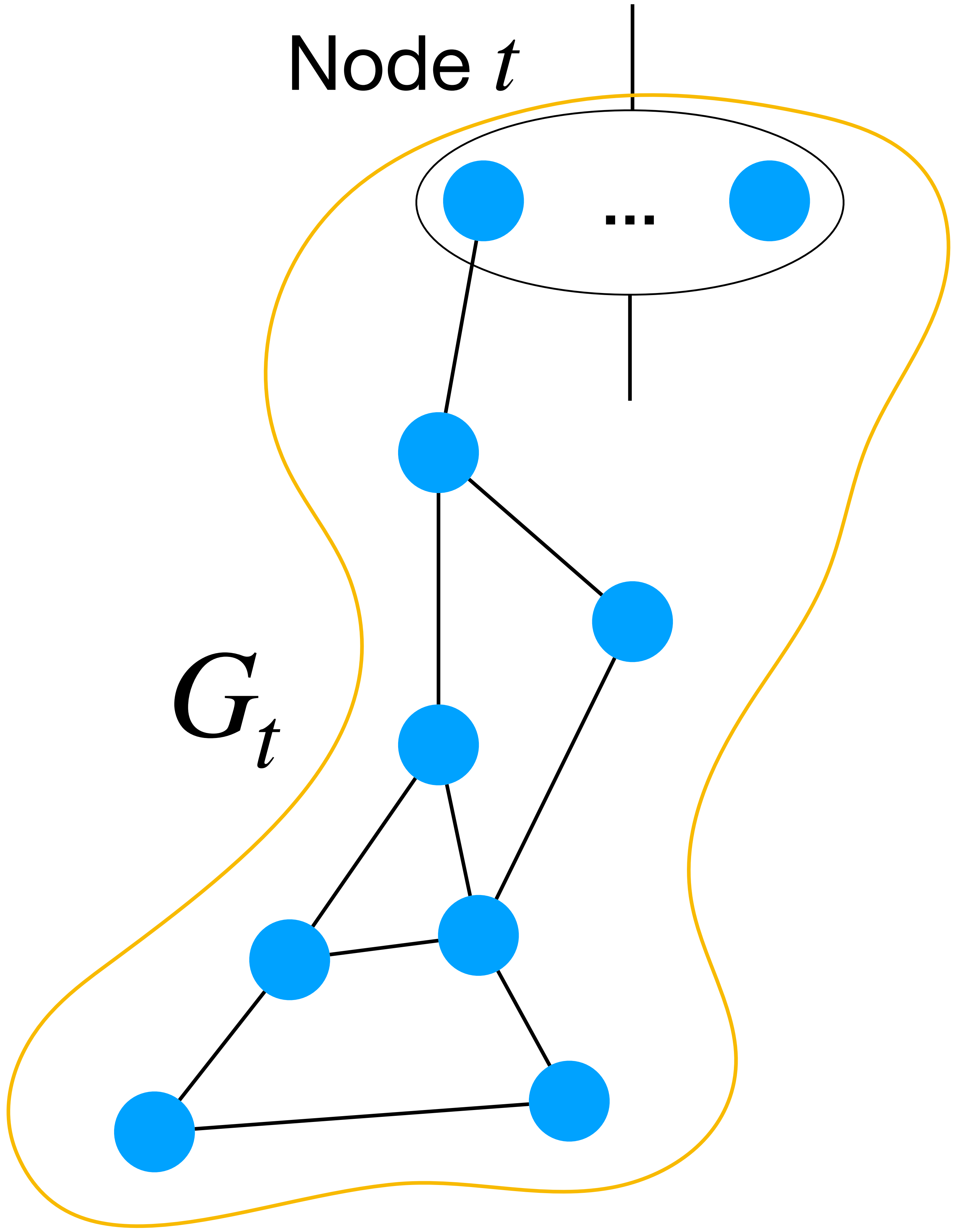
3-Colorability on Tree Decompositions



for $\sigma : \chi(t) \to \{red, green, blue\}$

$n_t(\sigma) =$ **true**, if σ can be extended to G_t

3-Colorability on Tree Decompositions

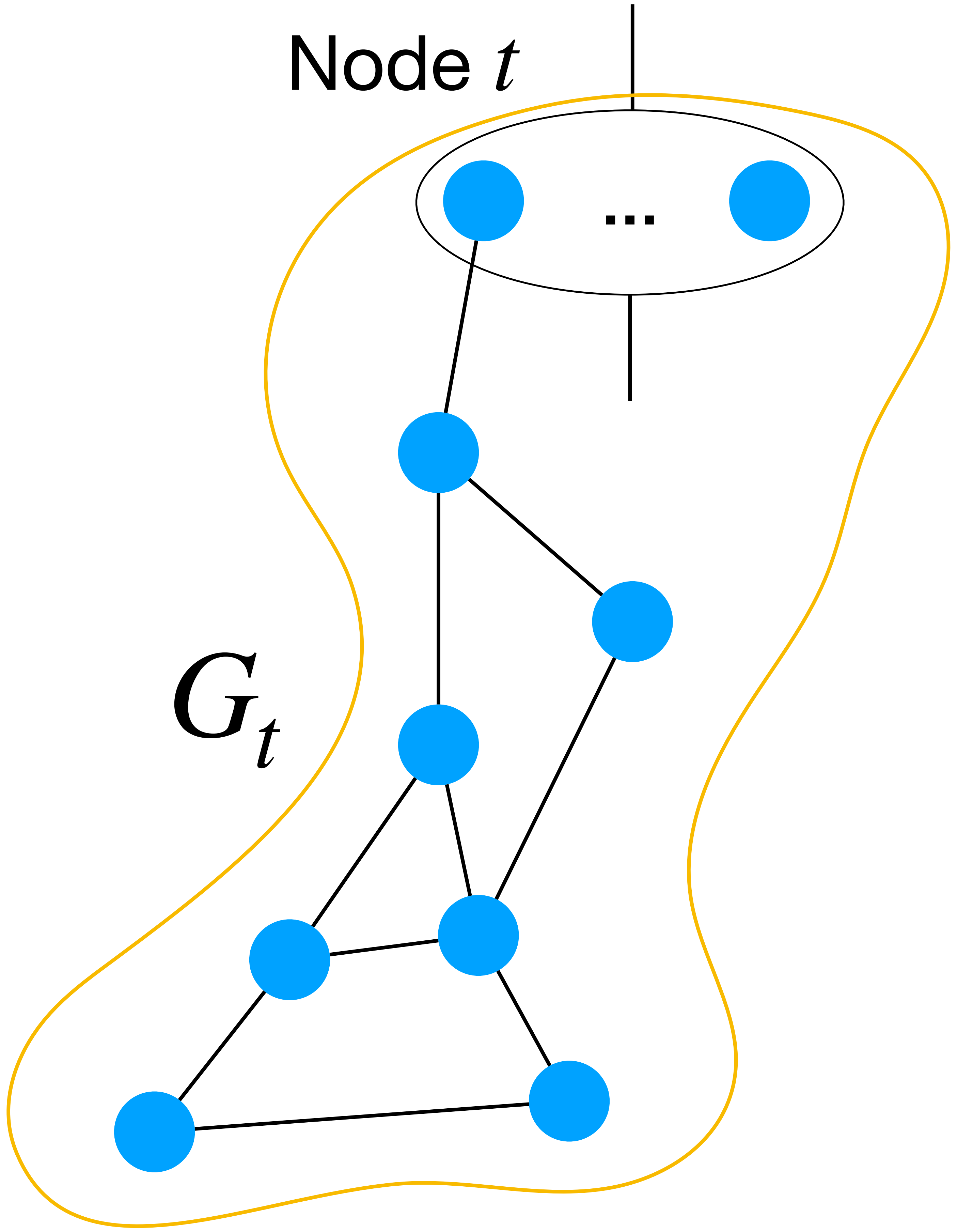


for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

$n_t(\sigma) =$ **true**, if σ can be extended to G_t

false, otherwise

3-Colorability on Tree Decompositions

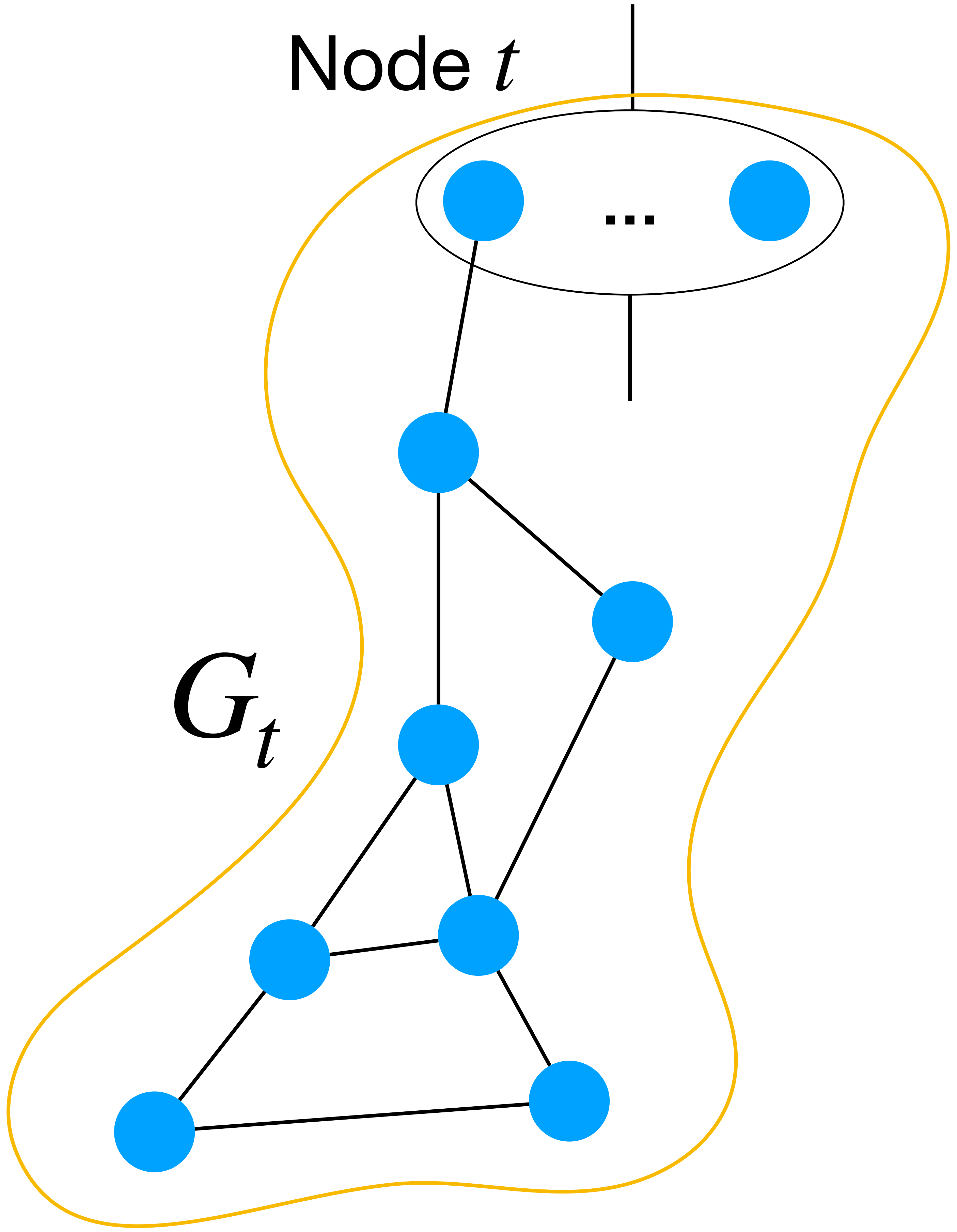


for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

$$n_t(\sigma) = \begin{cases} \textbf{true}\text{, if } \sigma \text{ can be extended to } G_t \\ \textbf{false}\text{, otherwise} \end{cases}$$

width k

3-Colorability on Tree Decompositions



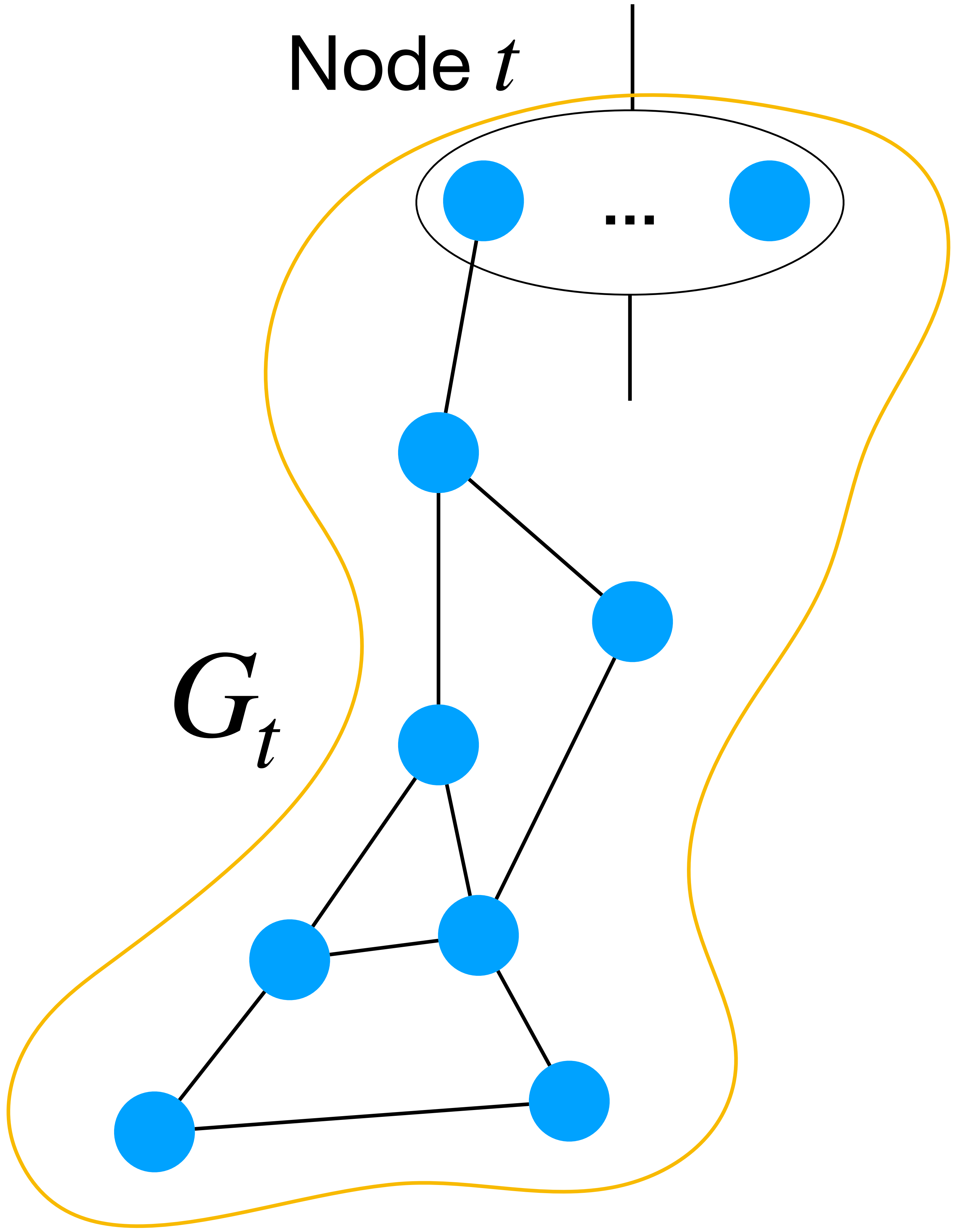
for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

$n_t(\sigma) =$ **true**, if σ can be extended to G_t

false, otherwise

width k ➡

3-Colorability on Tree Decompositions

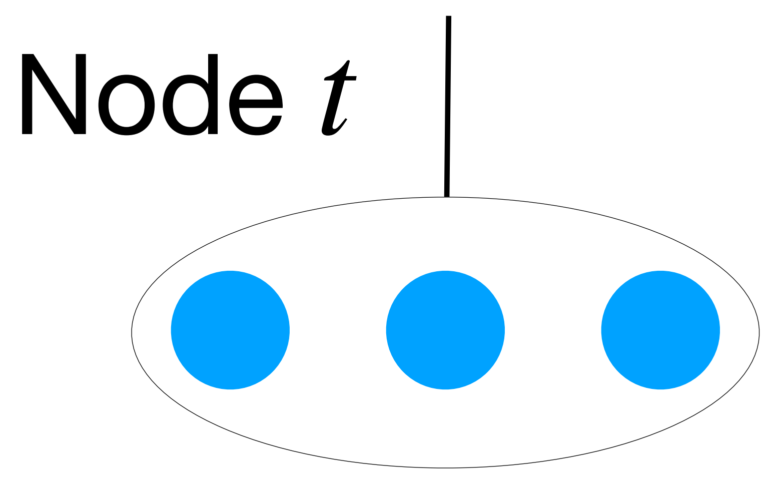


for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

$n_t(\sigma) =$ **true**, if σ can be extended to G_t
false, otherwise

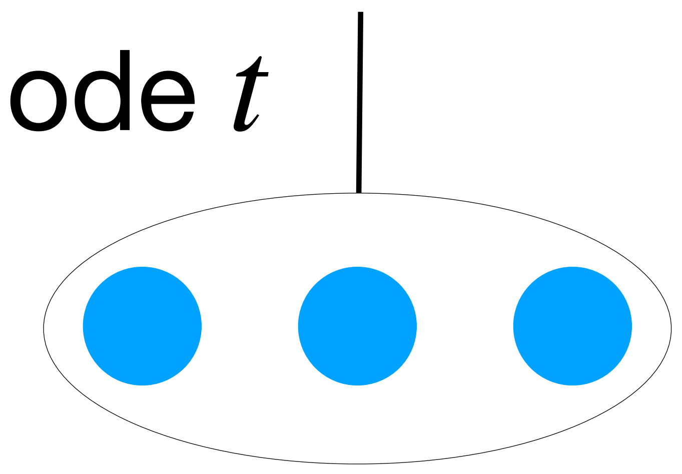
width k ➡ $\leq 3^{k+1}$ colorings

Leaf Nodes



Leaf Nodes

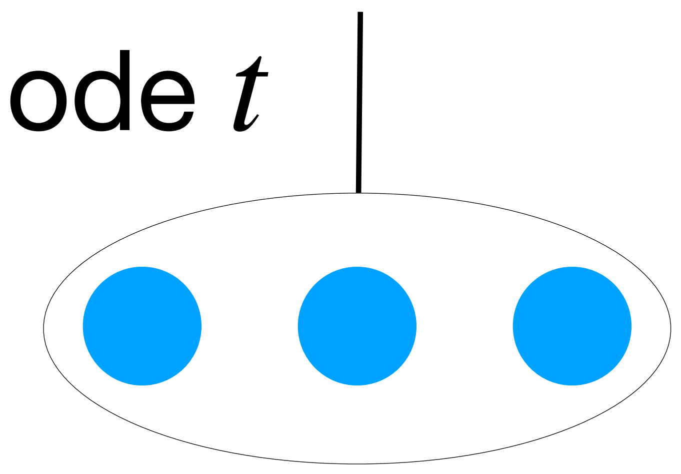
Node t



for $\sigma : \chi(t) \to \{red, green, blue\}$

Leaf Nodes

Node t

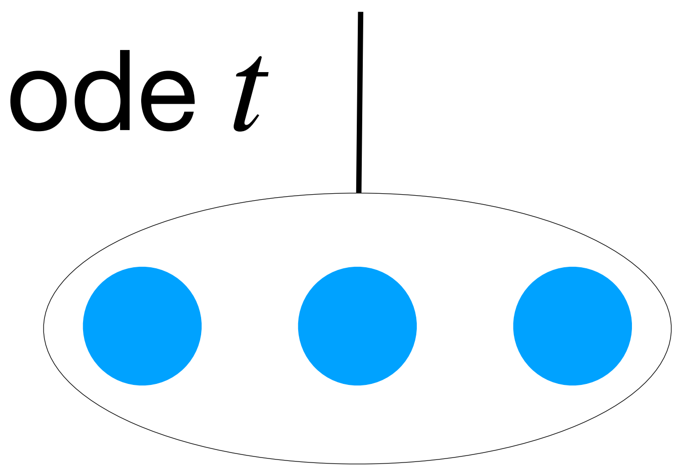


for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

if σ is a proper coloring of G_t

Leaf Nodes

Node t

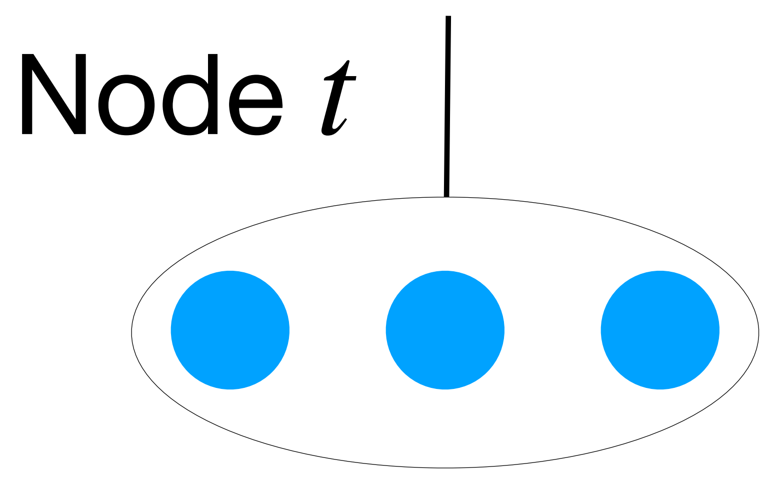


for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

if σ is a proper coloring of G_t

then $n_t(\sigma)$:= **true**

Leaf Nodes



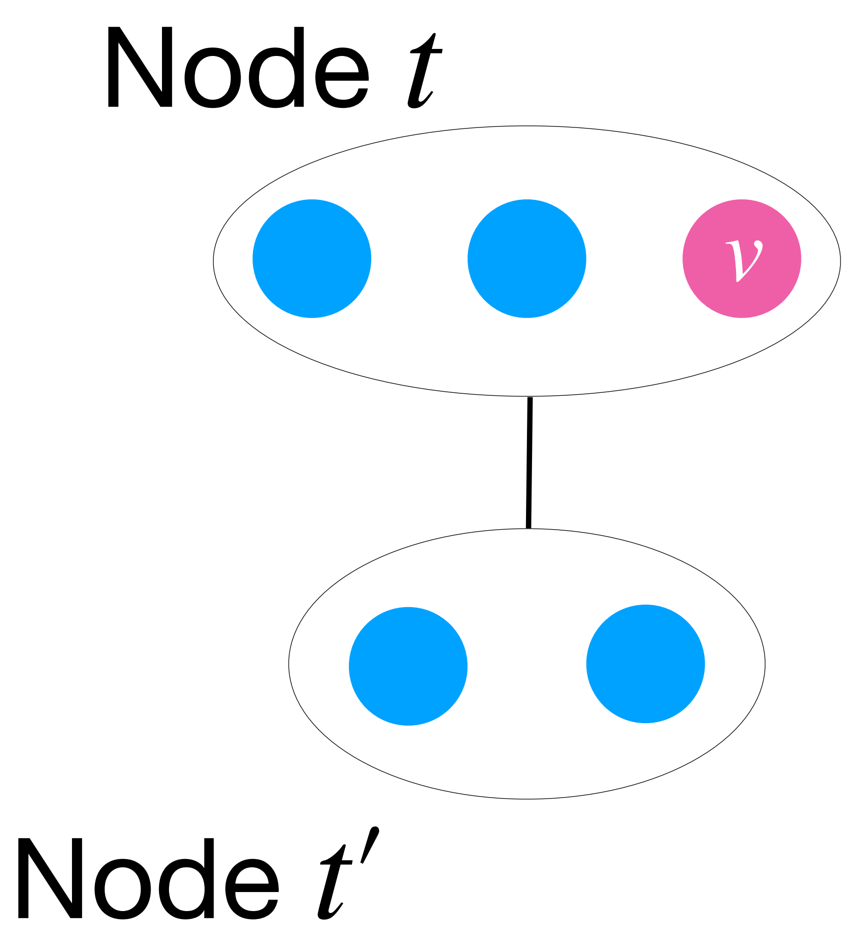
for $\sigma : \chi(t) \to \{red, green, blue\}$

if σ is a proper coloring of G_t

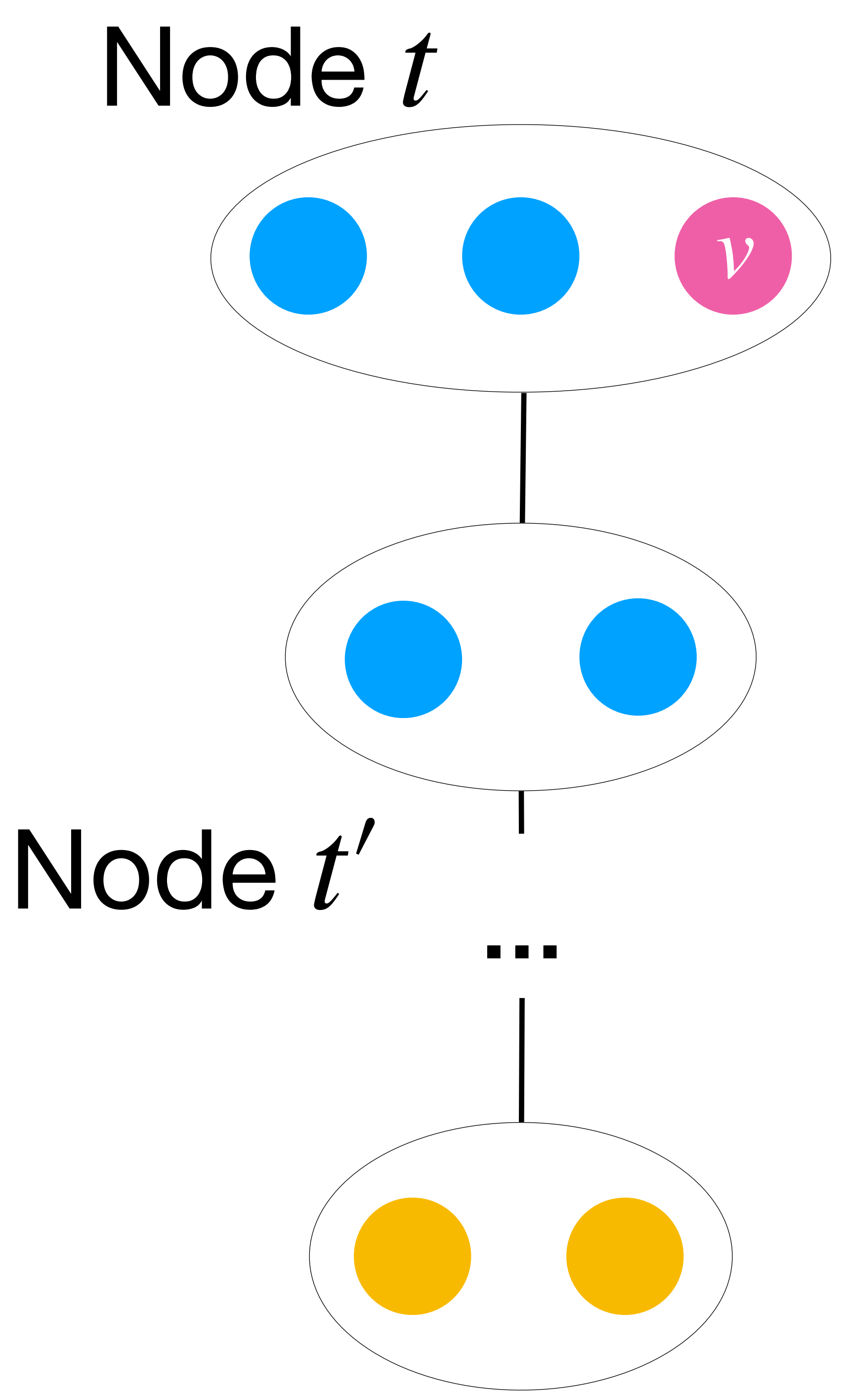
then $n_t(\sigma) :=$ **true**

else $n_t(\sigma) :=$ **false**

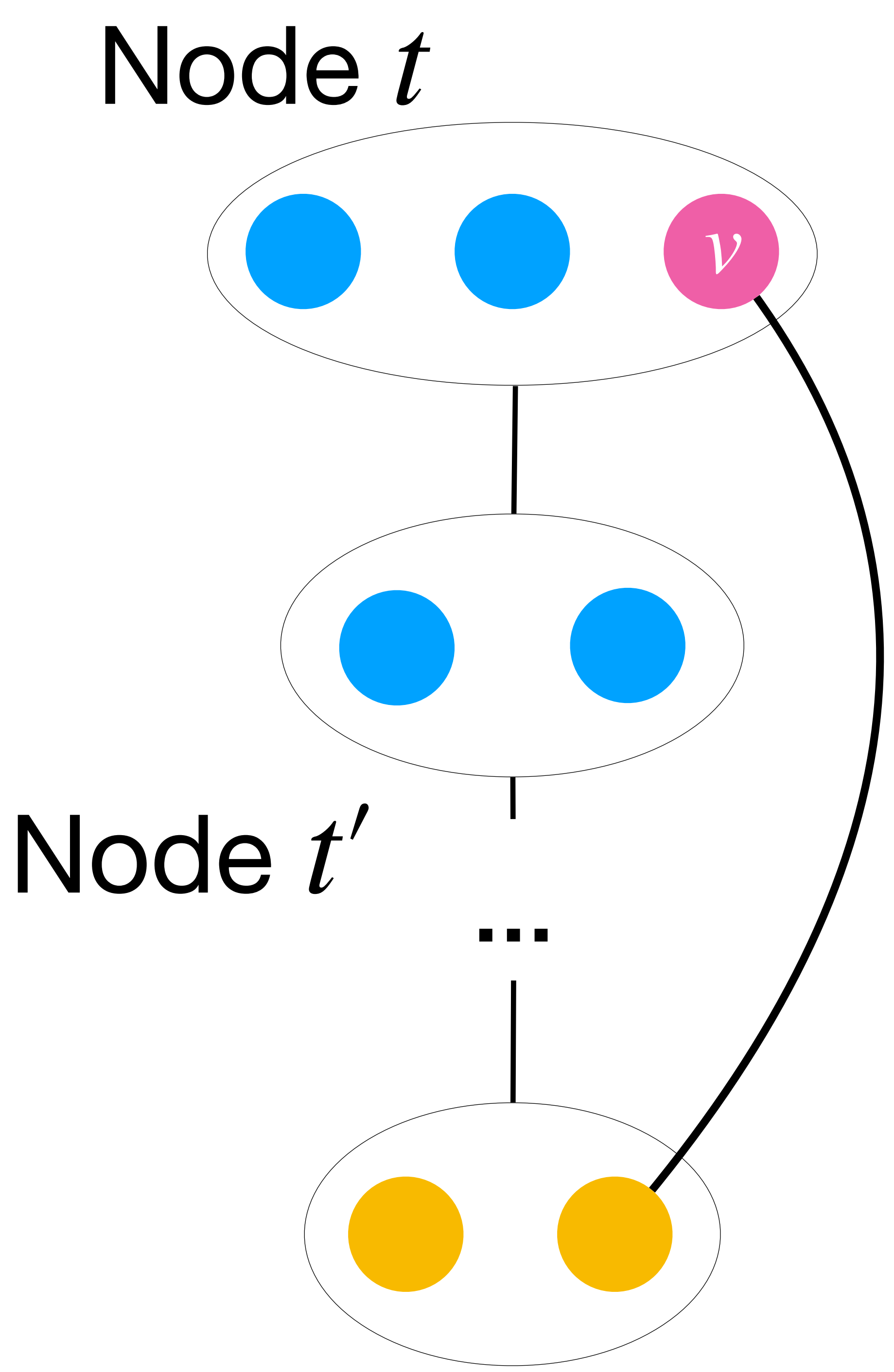
Introduce Nodes



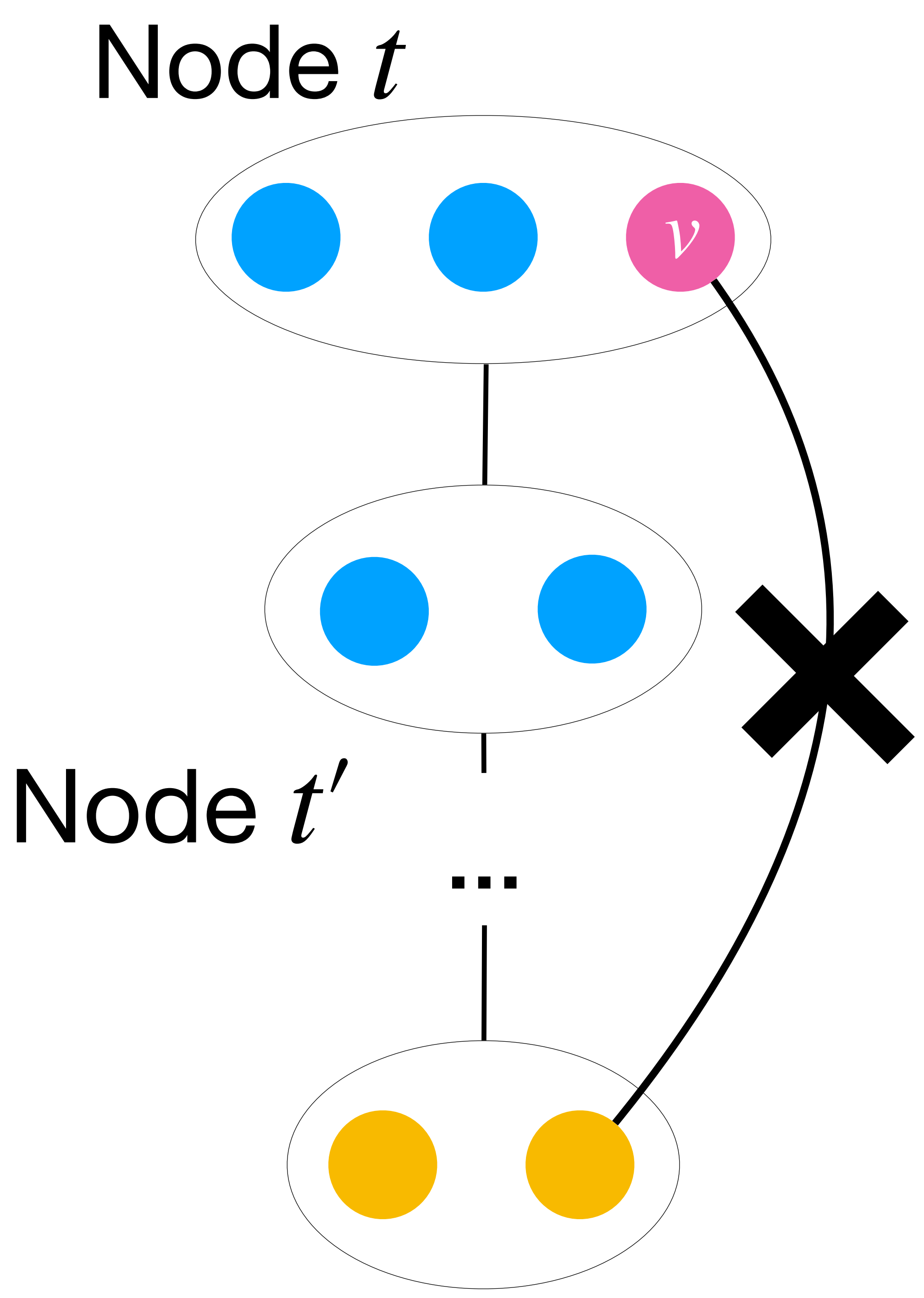
Introduce Nodes



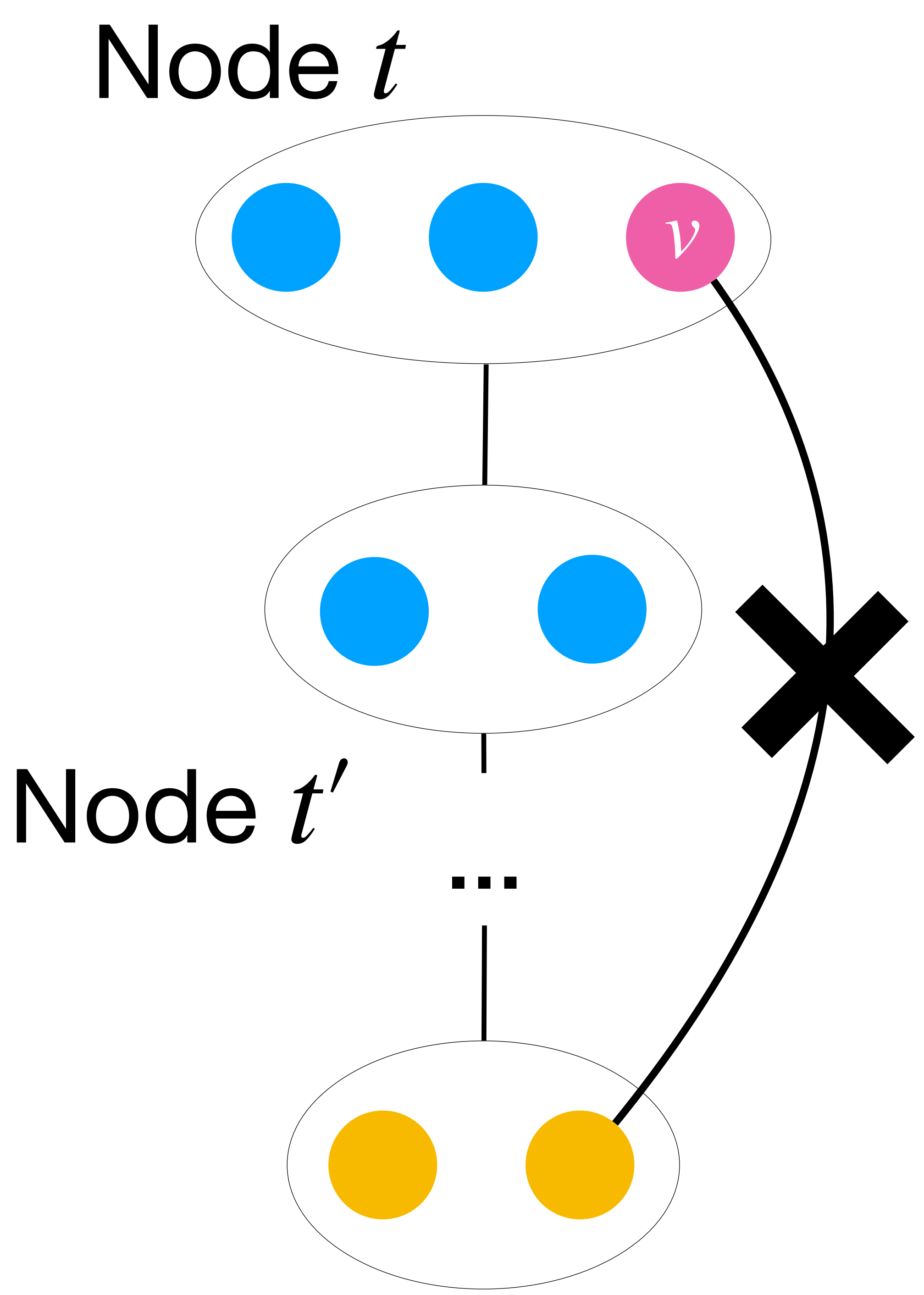
Introduce Nodes



Introduce Nodes

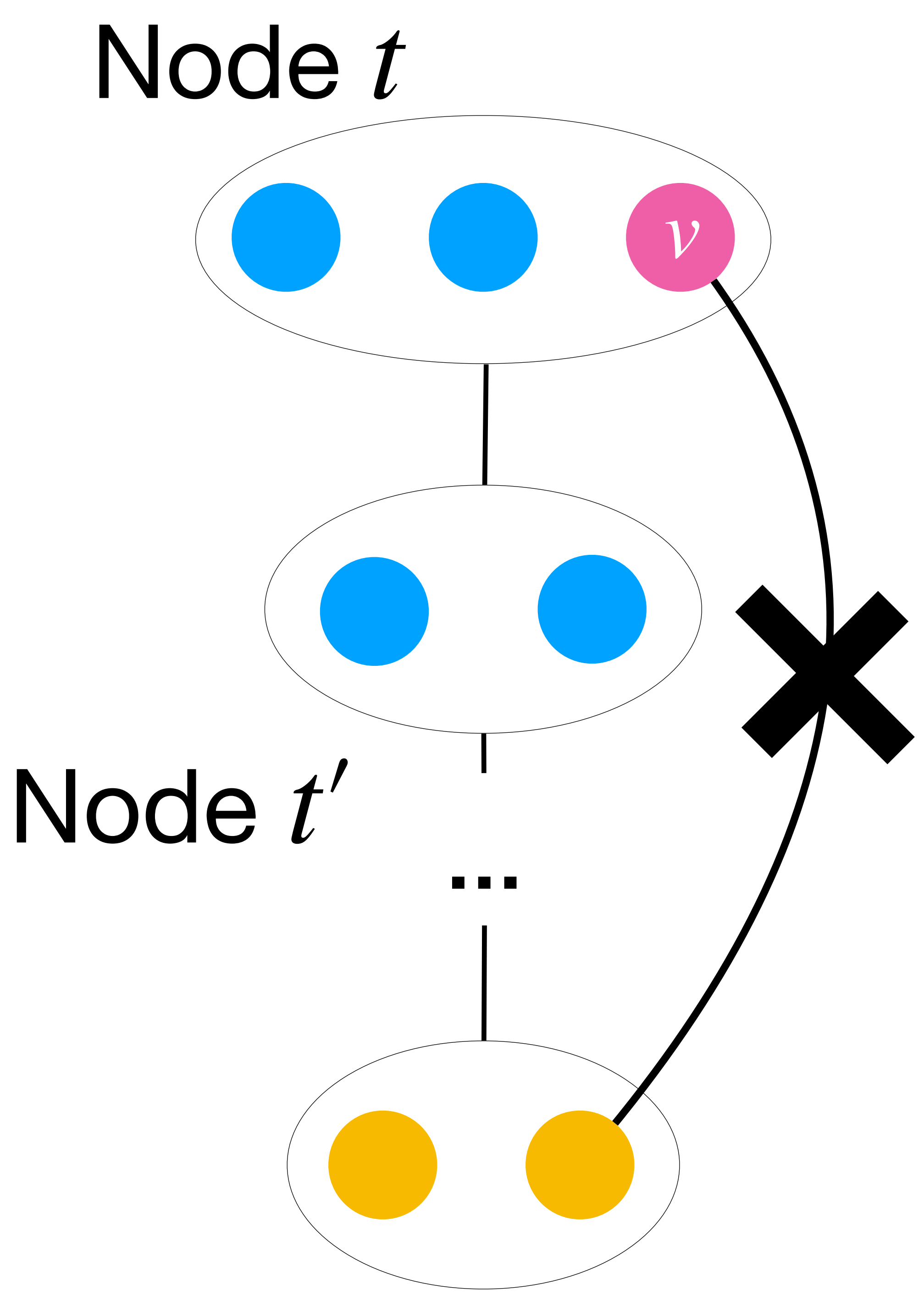


Introduce Nodes



for $\sigma : \chi(t) \to \{red, green, blue\}$

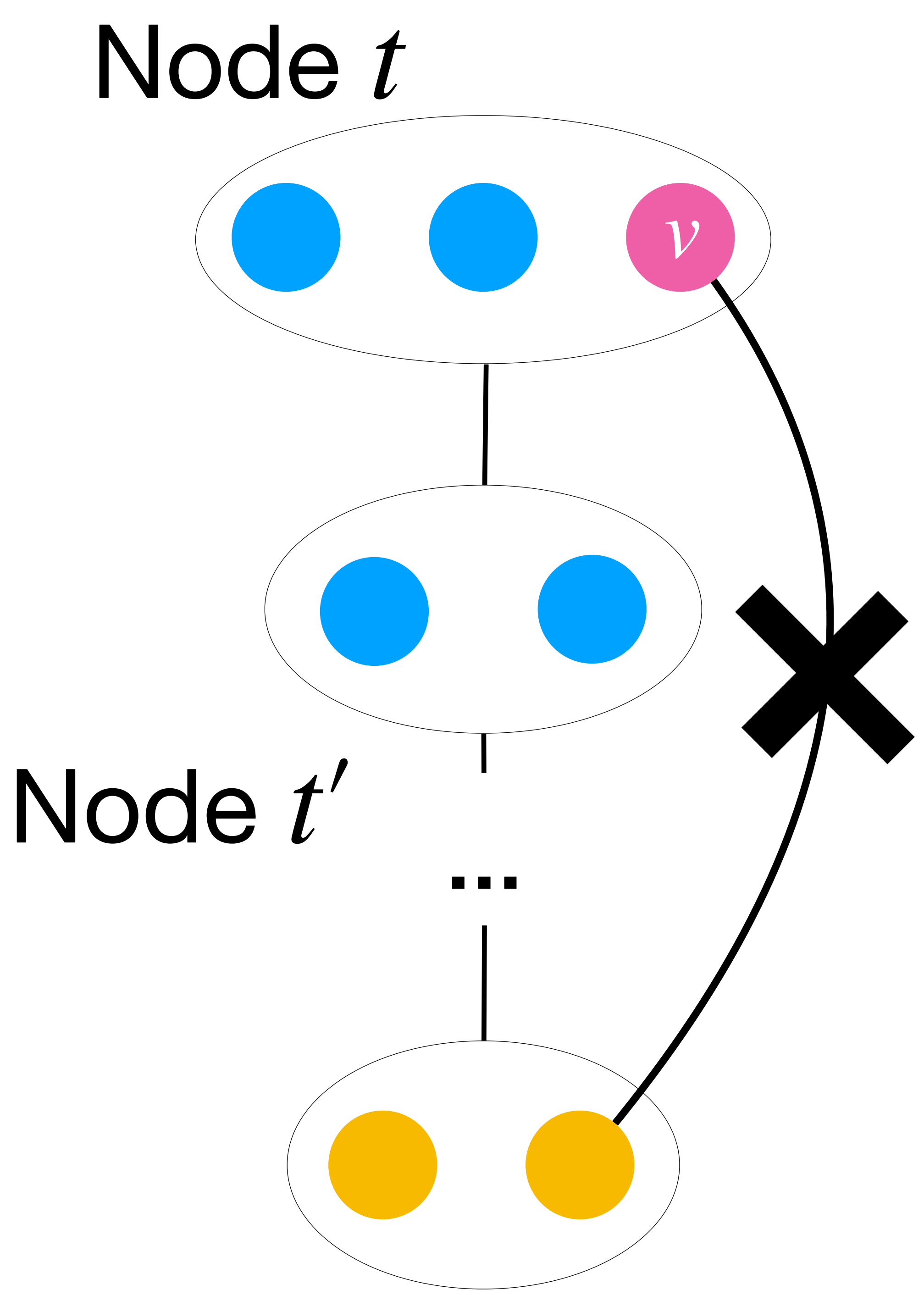
Introduce Nodes



for $\sigma : \chi(t) \to \{red, green, blue\}$

if σ is a proper coloring of $G[\chi(t)]$

Introduce Nodes

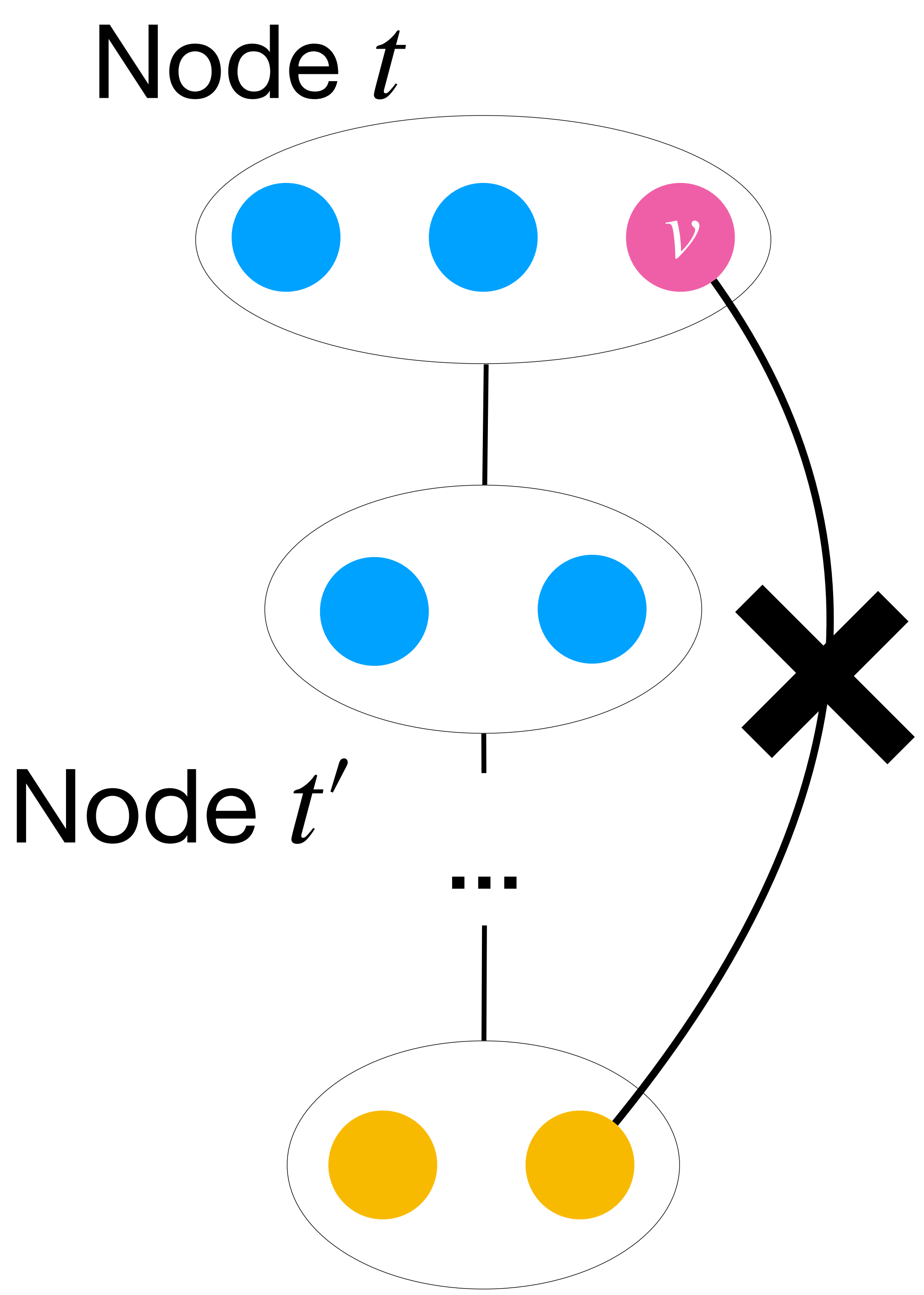


for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

if σ is a proper coloring of $G[\chi(t)]$

then $n_t(\sigma) := n_{t'}(\sigma')$

Introduce Nodes



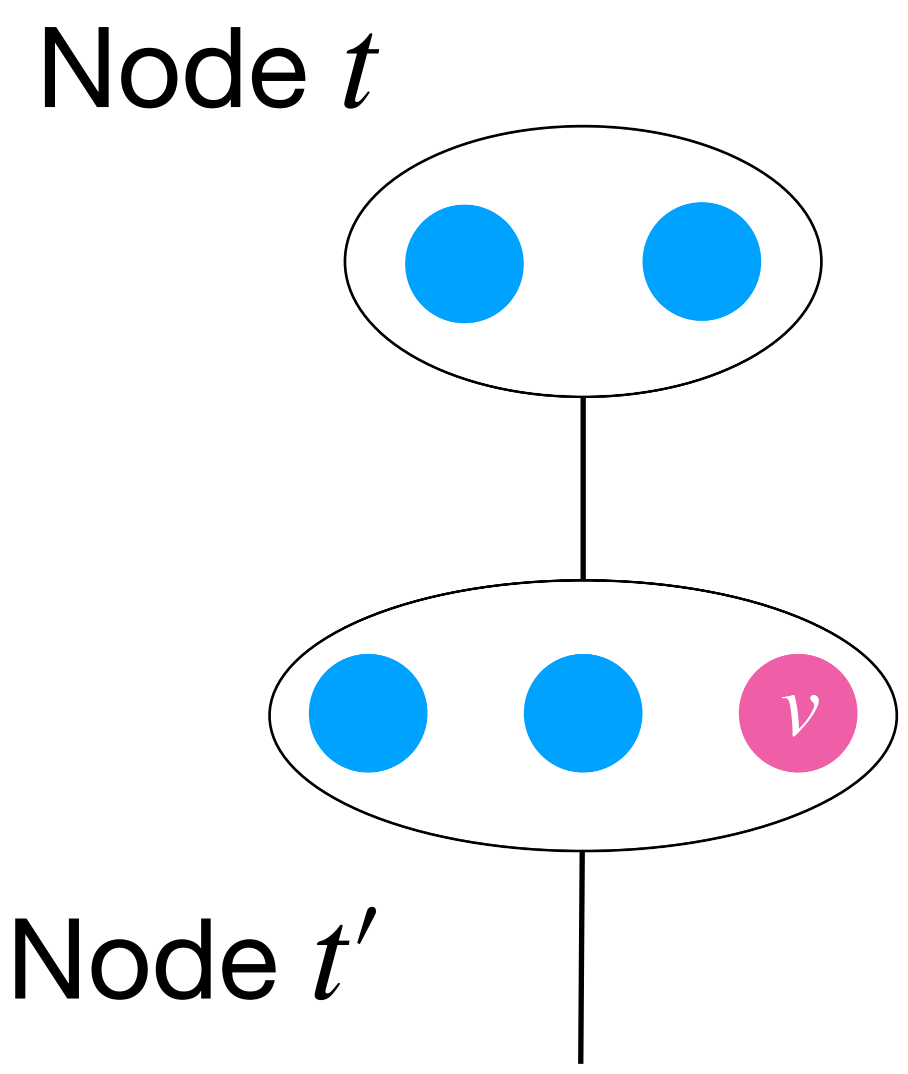
for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

if σ is a proper coloring of $G[\chi(t)]$

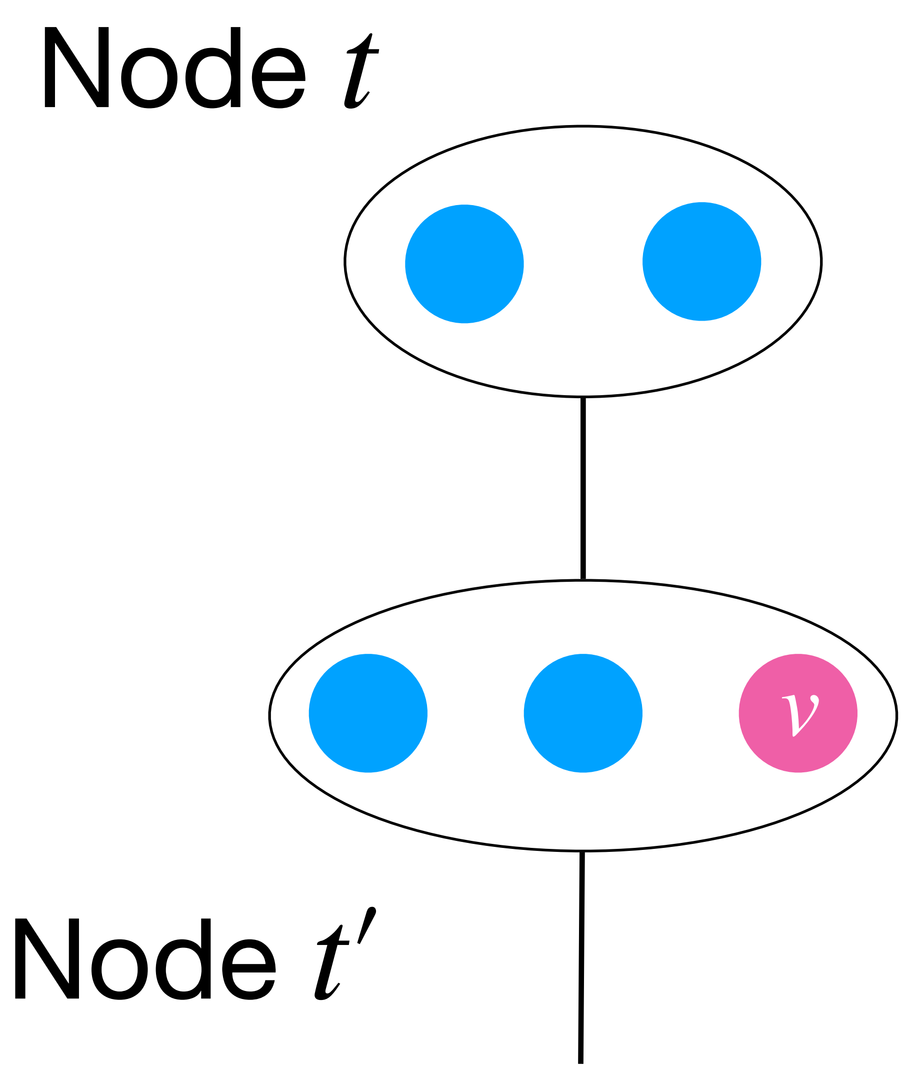
then $n_t(\sigma) := n_{t'}(\sigma')$

else $n_t(\sigma) :=$ **false**

Forget Nodes

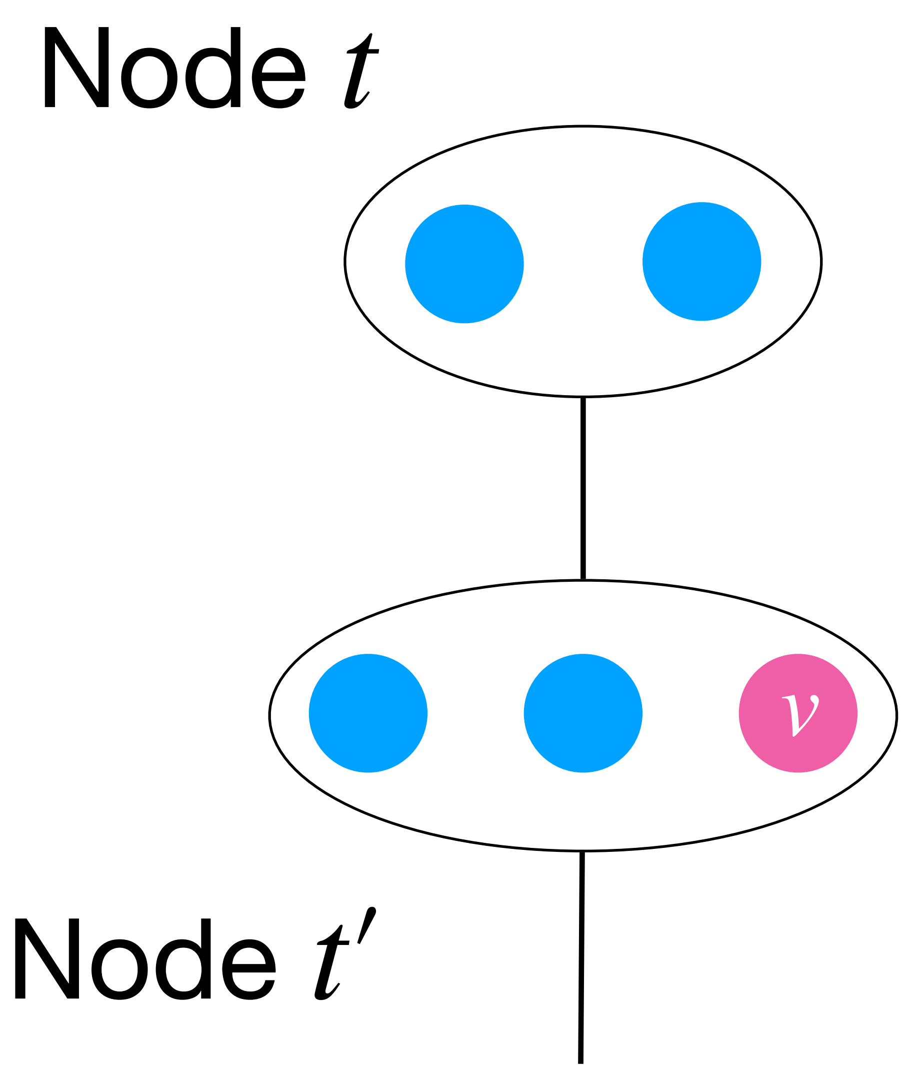


Forget Nodes



for $\sigma : \chi(t) \to \{red, green, blue\}$

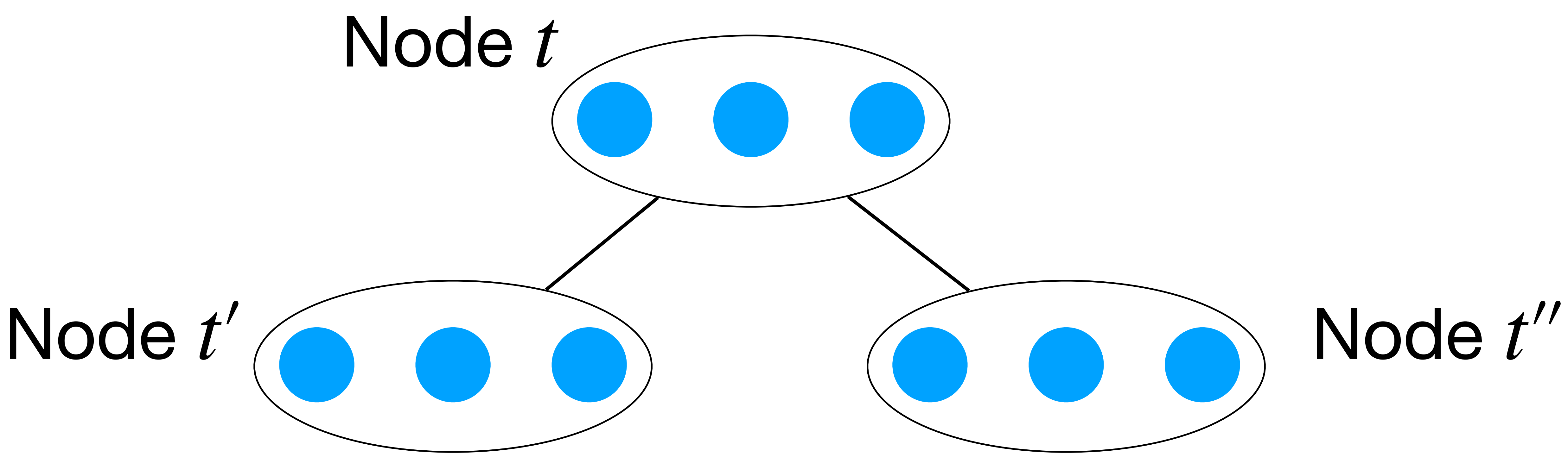
Forget Nodes



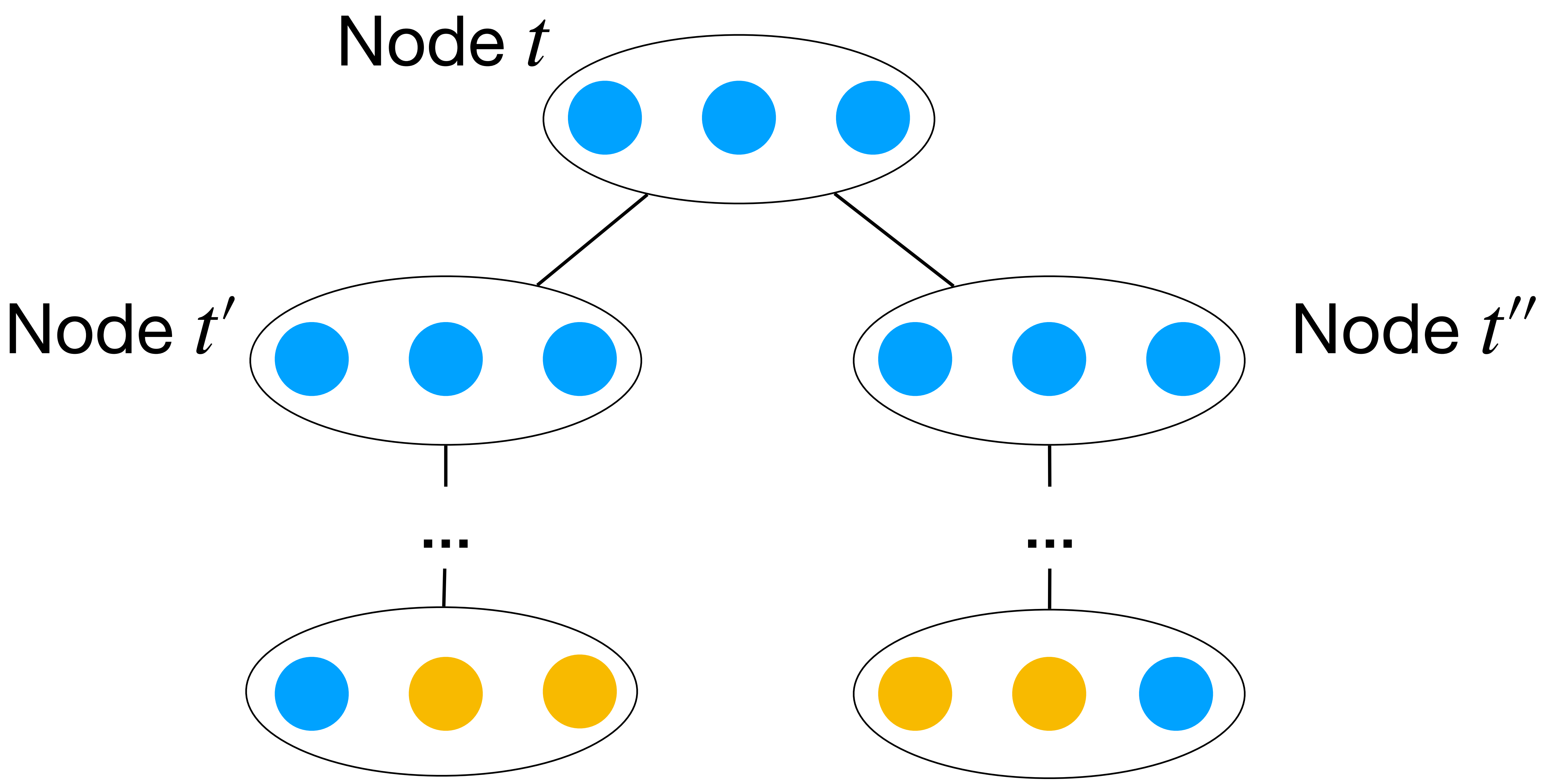
for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

then $n_t(\sigma) := \bigvee_{c \in red, green, blue} n_{t'}(\sigma' \cup \{v \mapsto c\})$

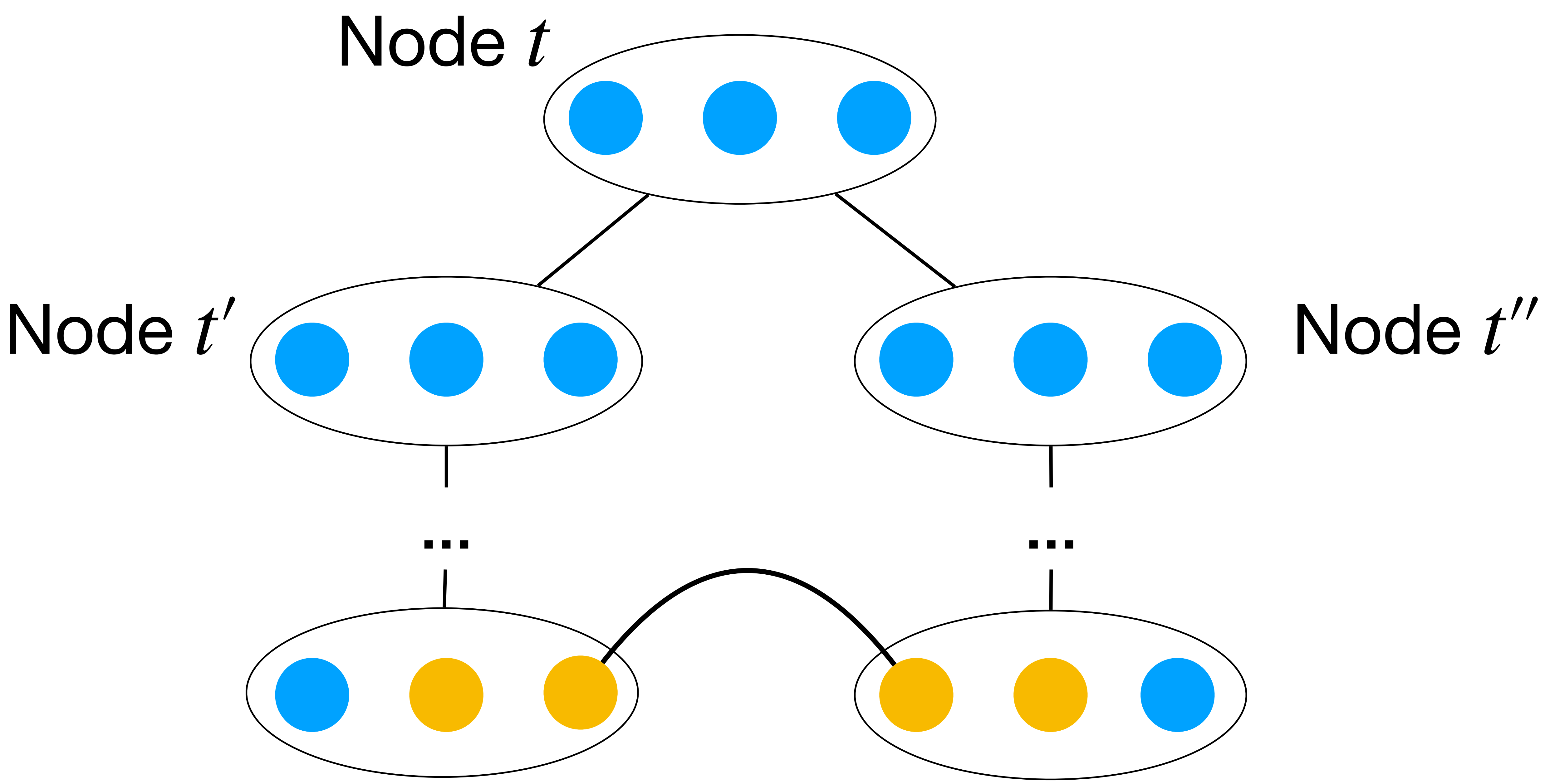
Join Nodes



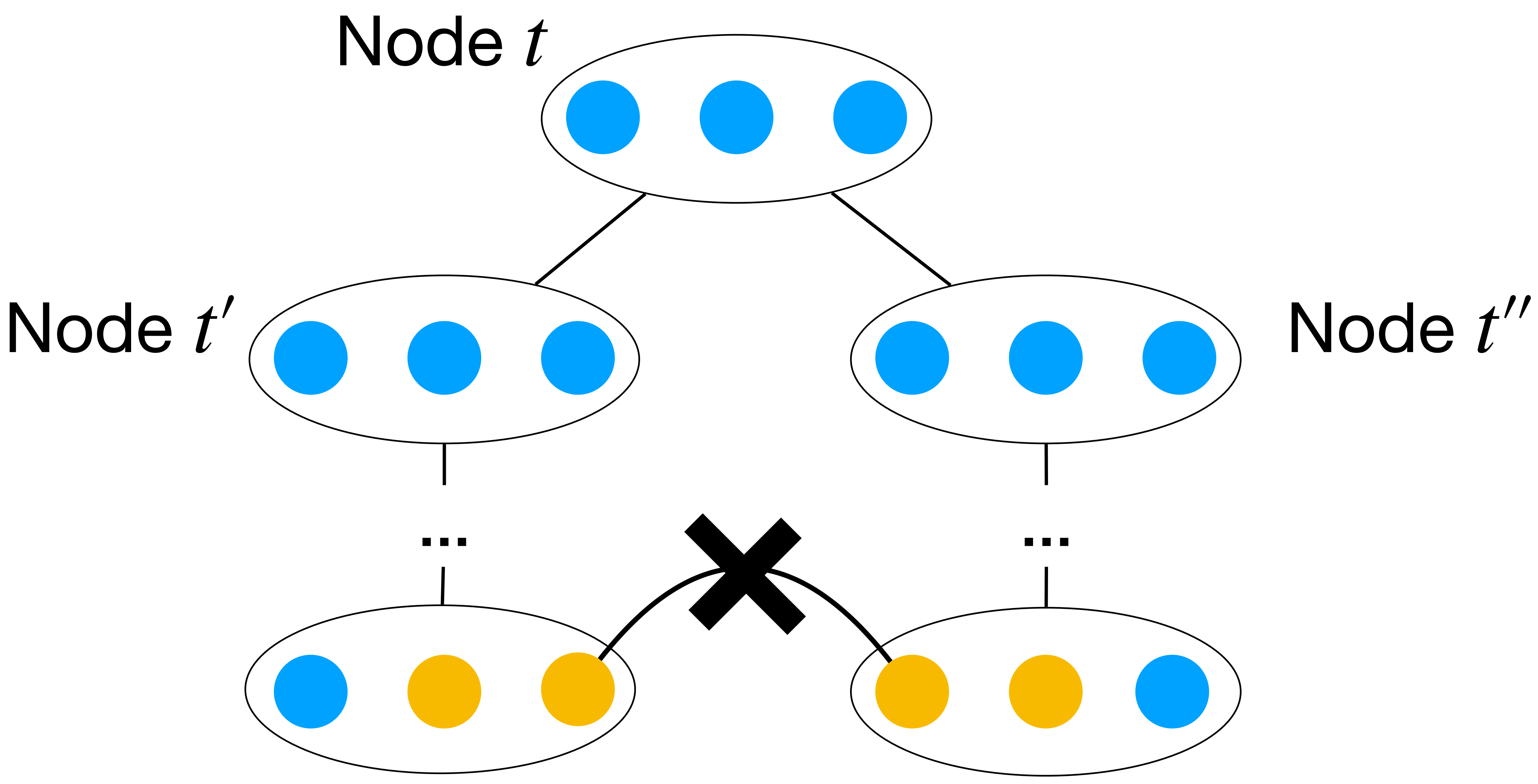
Join Nodes



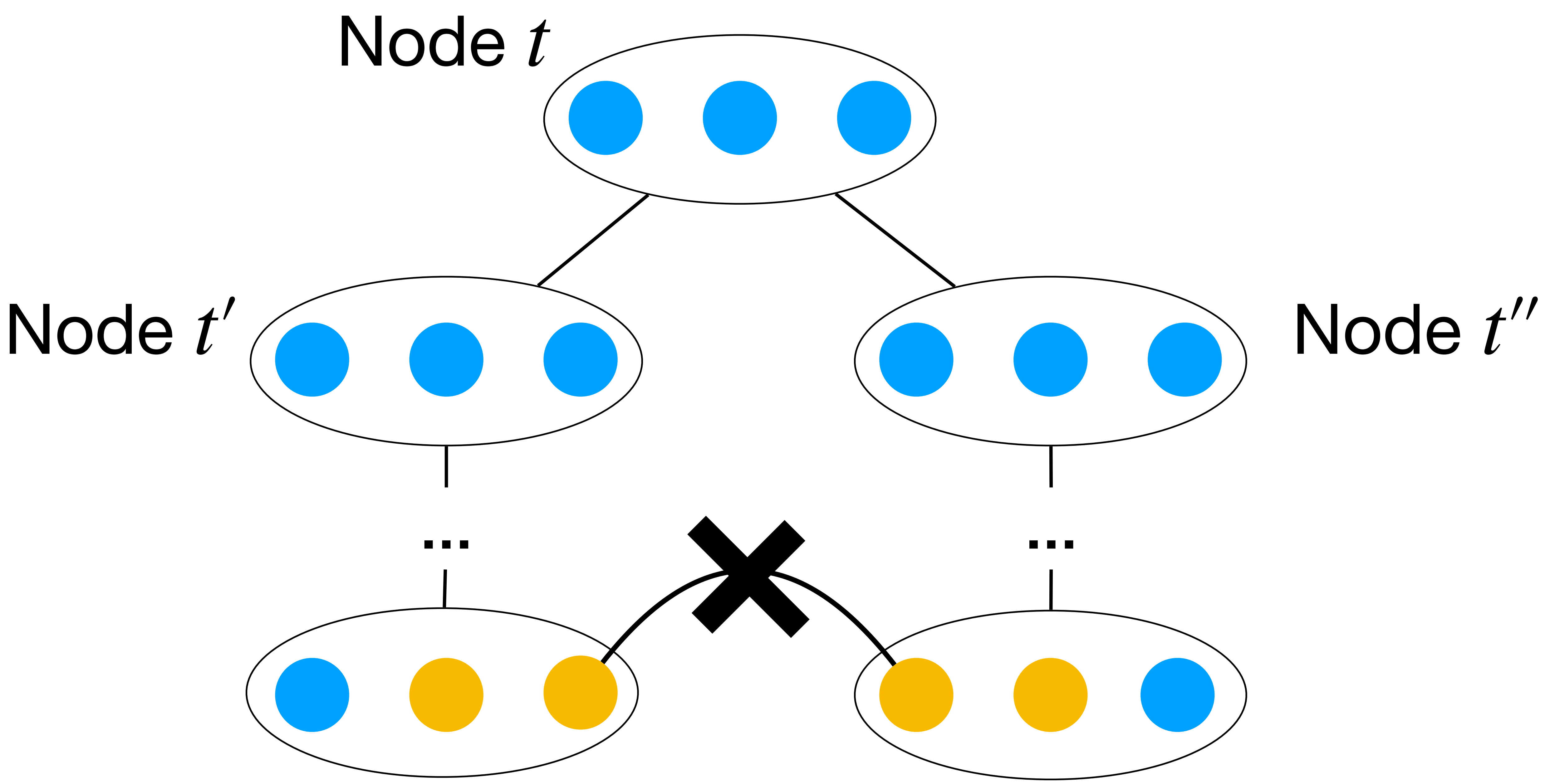
Join Nodes



Join Nodes

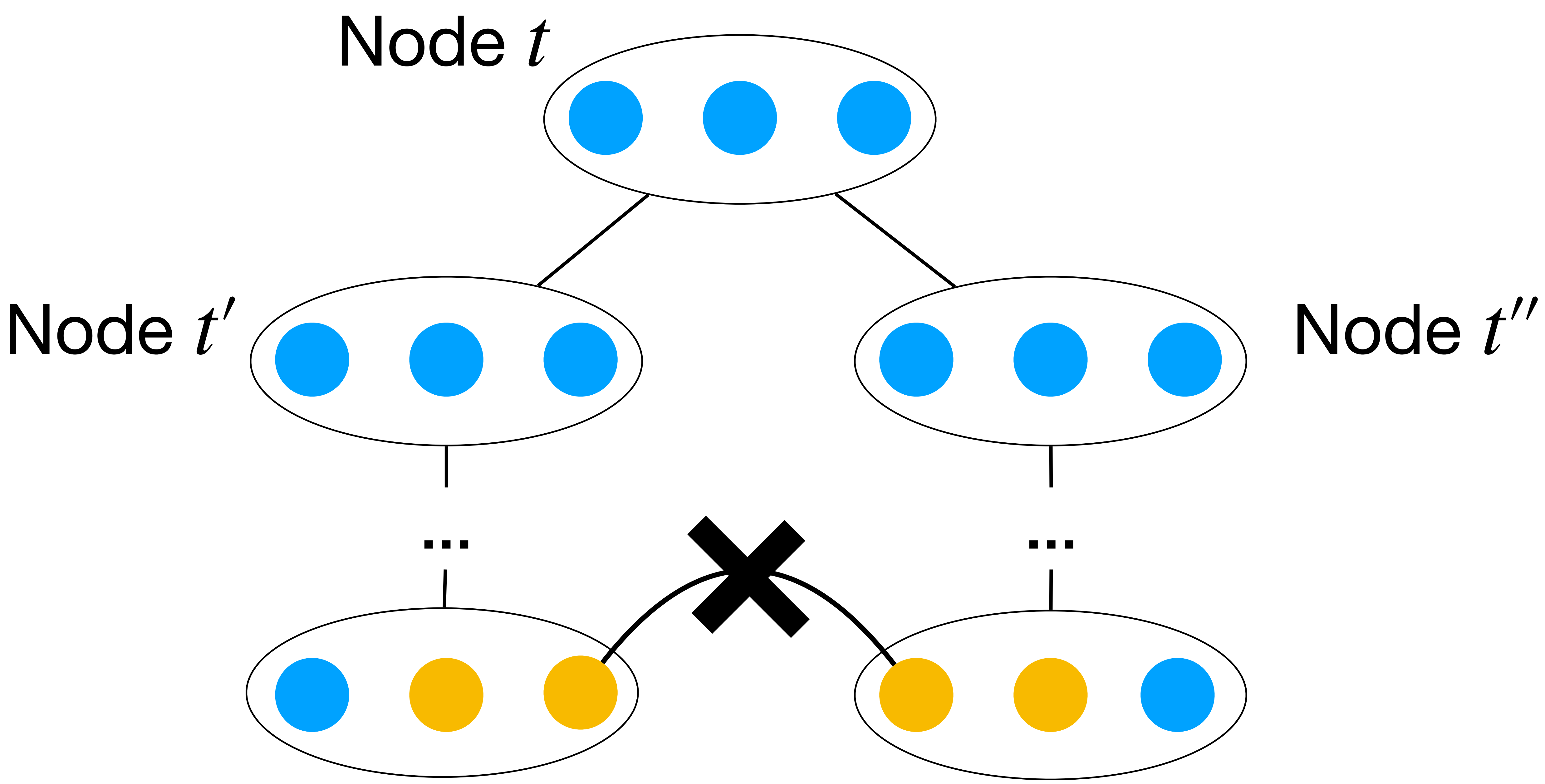


Join Nodes



for $\sigma : \chi(t) \to \{red, green, blue\}$

Join Nodes



for $\sigma : \chi(t) \rightarrow \{red, green, blue\}$

$$n_t(\sigma) := n_{t'}(\sigma) \wedge n_{t''}(\sigma)$$

3-Colorability

Theorem

3-COLORABILITY is **FPT** parameterized by the treewidth of the input graph.*

3-Colorability

Theorem

3-COLORABILITY is **FPT** parameterized by the treewidth of the input graph.*

*If we can compute a nice tree decomposition of width $g(k)$ in FPT time.

Pros and Cons

Pros and Cons

For **Dominating Set**, see

Pros and Cons

For **Dominating Set**, see

Cygan et al., **Parameterized Algorithms**

Pros and Cons

For **Dominating Set**, see

Cygan et al., **Parameterized Algorithms**

Pro: hand-crafted dynamic programming algorithm yields best running time

Pros and Cons

For **Dominating Set**, see

Cygan et al., **Parameterized Algorithms**

Pro: hand-crafted dynamic programming algorithm yields best running time

Con: involves (repeating) tedious arguments

Pros and Cons

For **Dominating Set**, see

Cygan et al., **Parameterized Algorithms**

Pro: hand-crafted dynamic programming algorithm yields best running time

Con: involves (repeating) tedious arguments

Con: coming up with the right notion of “partial solution” ($n_t(U)$) is difficult

Computing Treewidth

Treewidth is FPT

TREEWIDTH

Input: A graph G and an integer k.

Question: Does G have treewidth $\leq k$?

Parameter: k

Treewidth is FPT

TREEWIDTH

Input: A graph G and an integer k.

Question: Does G have treewidth $\leq k$?

Parameter: k

Theorem (Bodlaender)

There is a function f and an algorithm A that computes a tree decomposition of a graph G in time $f(k)\,|\,V(G)\,|$ or decides that its treewidth is greater than k.

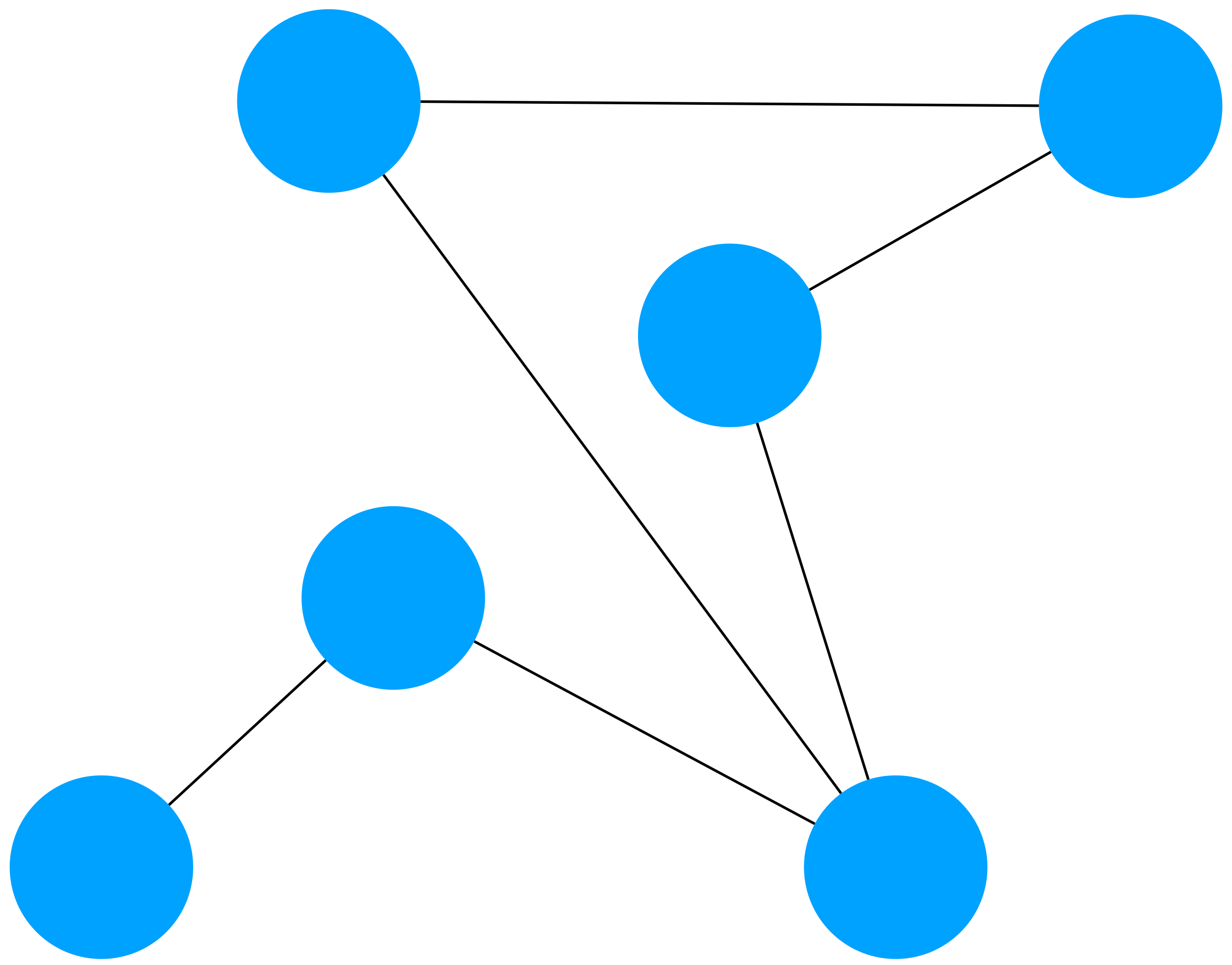
FPT 2-Approximation

Theorem (Korhonen 2021)

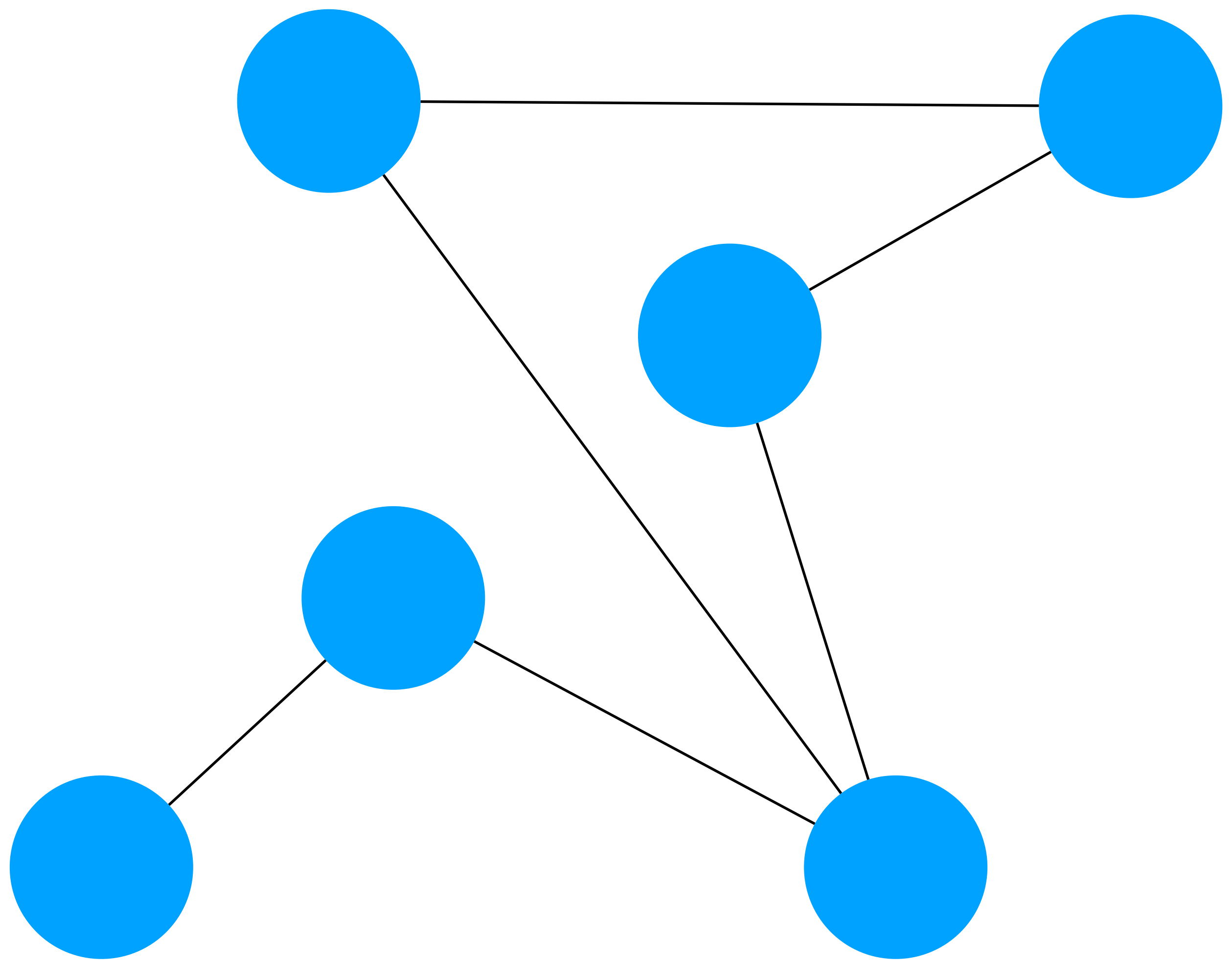
There is an algorithm that, given an n-vertex graph G and an integer k, in time $2^{O(k)}n$ either outputs a tree decomposition of width at most $2k+1$ or determines that the treewidth of G is larger than k.

Heuristics

Elimination Orderings



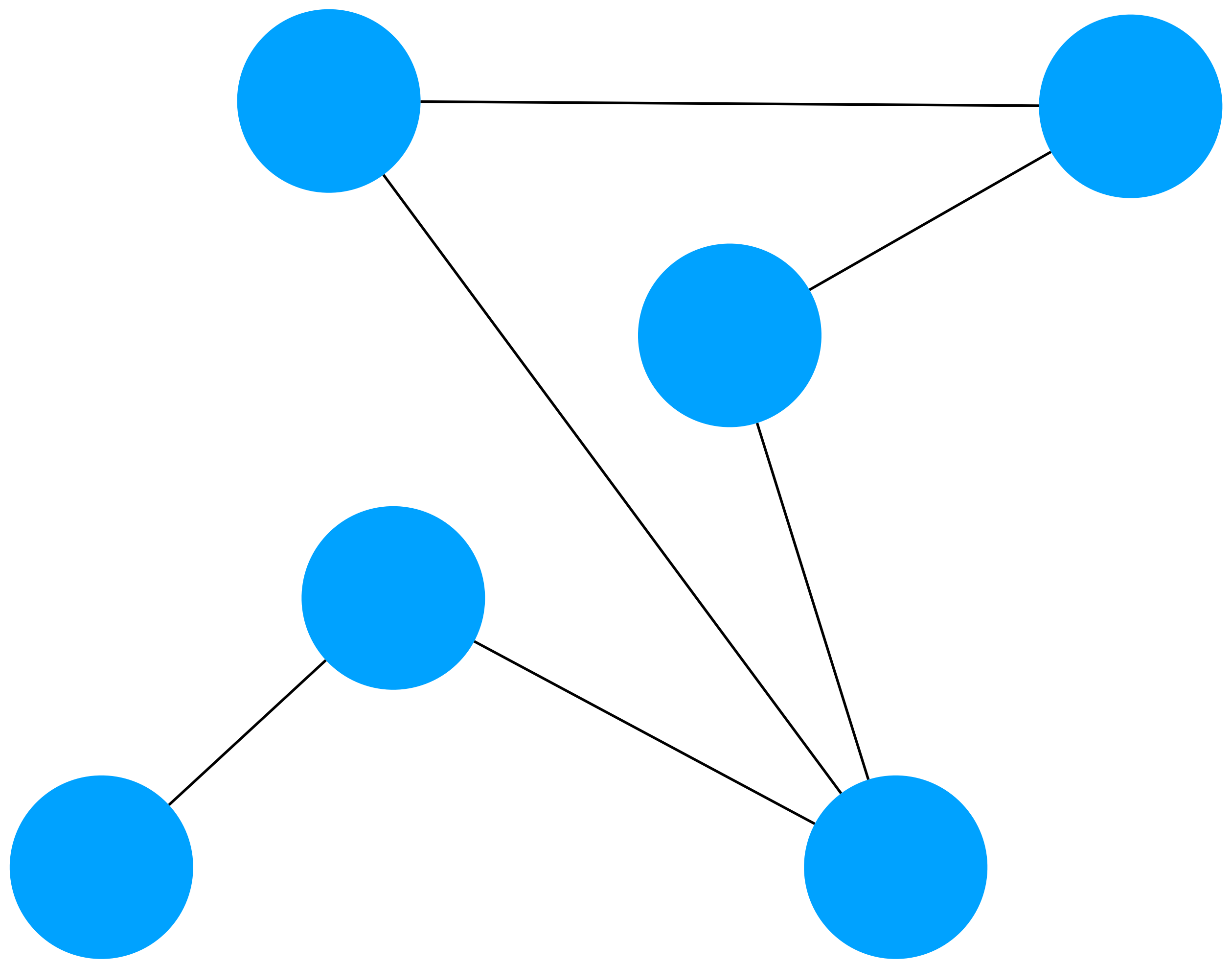
Elimination Orderings



Definition

Let $G = (V, E)$ be a graph. An **elimination ordering** of G is simply an ordering $\sigma = (v_1, \ldots, v_n)$ of V.

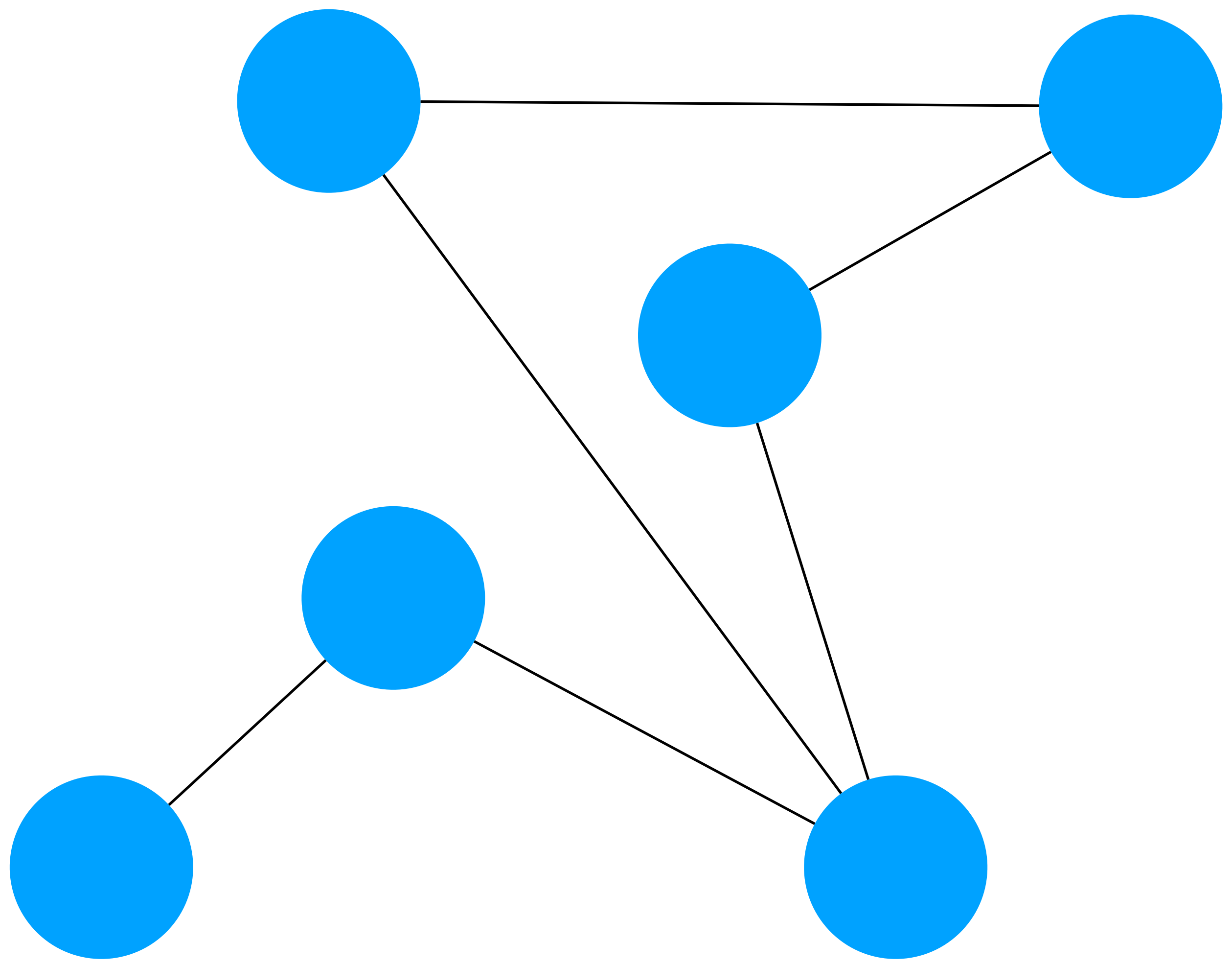
Elimination Orderings



Definition

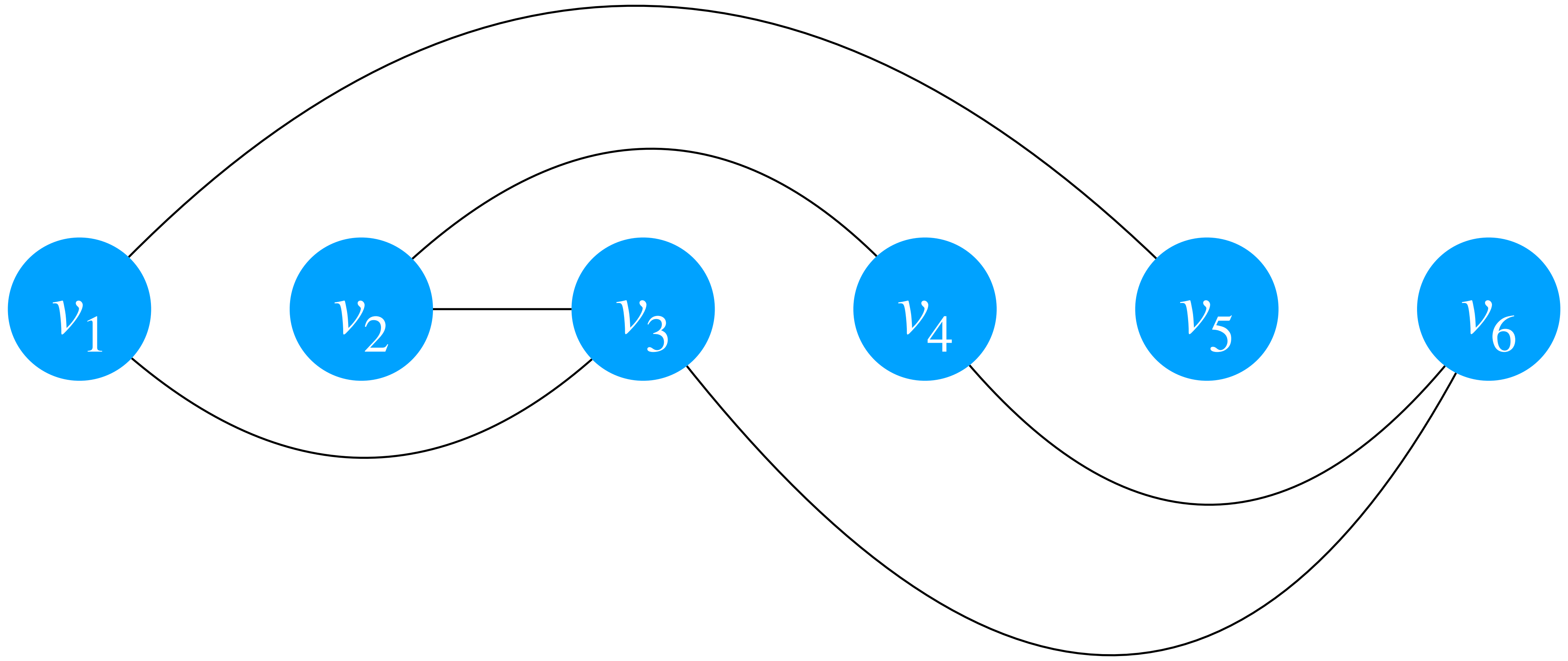
Let $G = (V, E)$ be a graph. An **elimination ordering** of G is simply an ordering $\sigma = (v_1, \ldots, v_n)$ of V.

Elimination Orderings

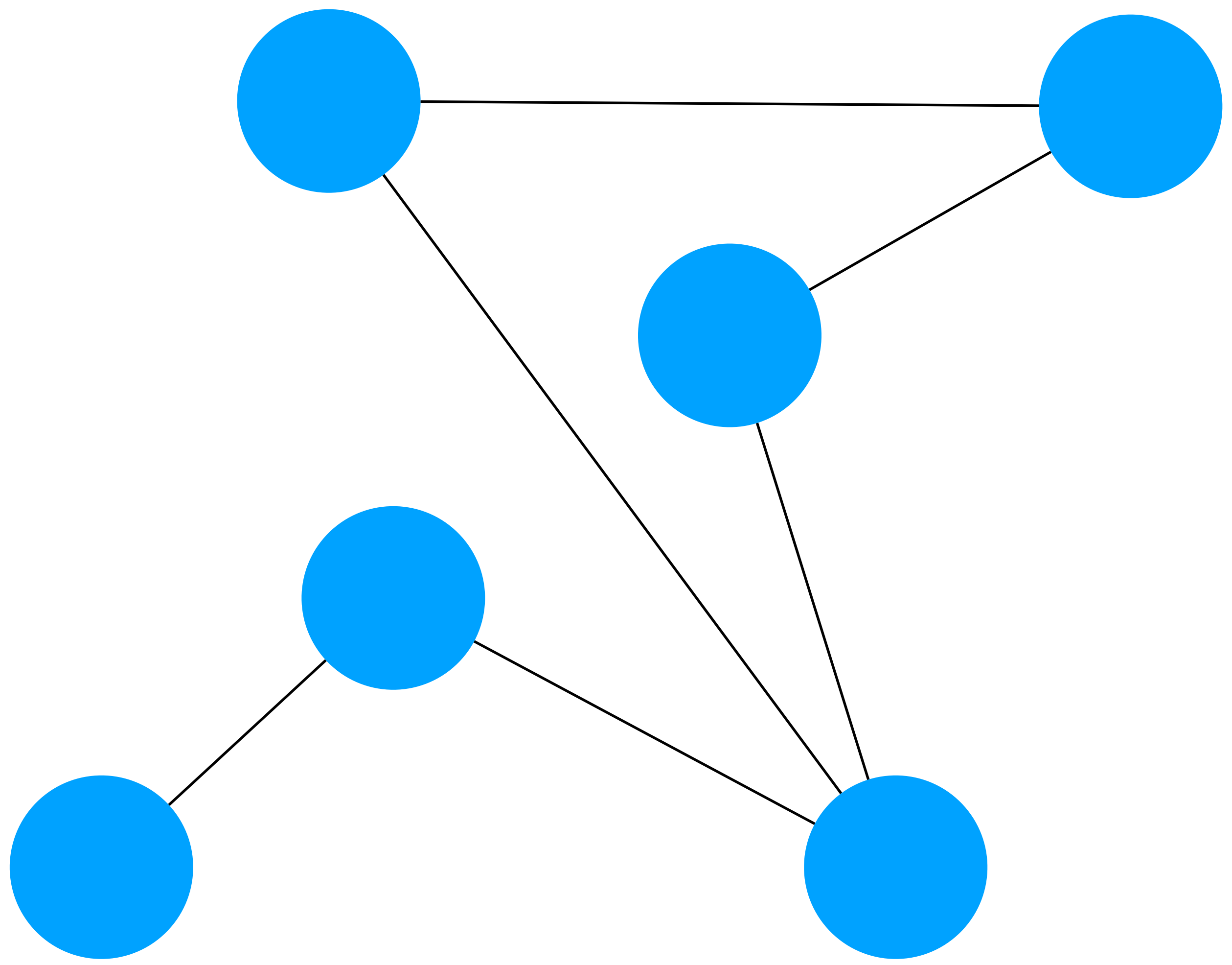


Definition

Let $G = (V, E)$ be a graph. An **elimination ordering** of G is simply an ordering $\sigma = (v_1, \ldots, v_n)$ of V.

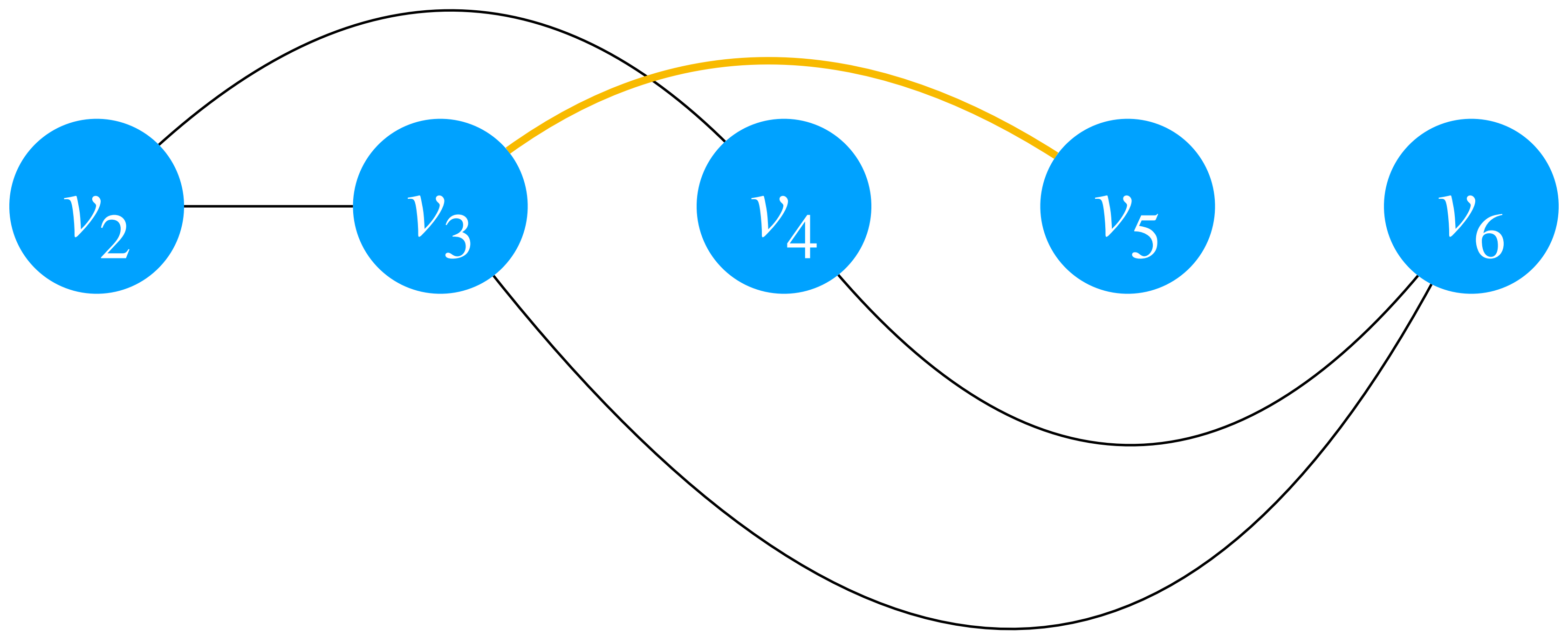


Elimination Orderings

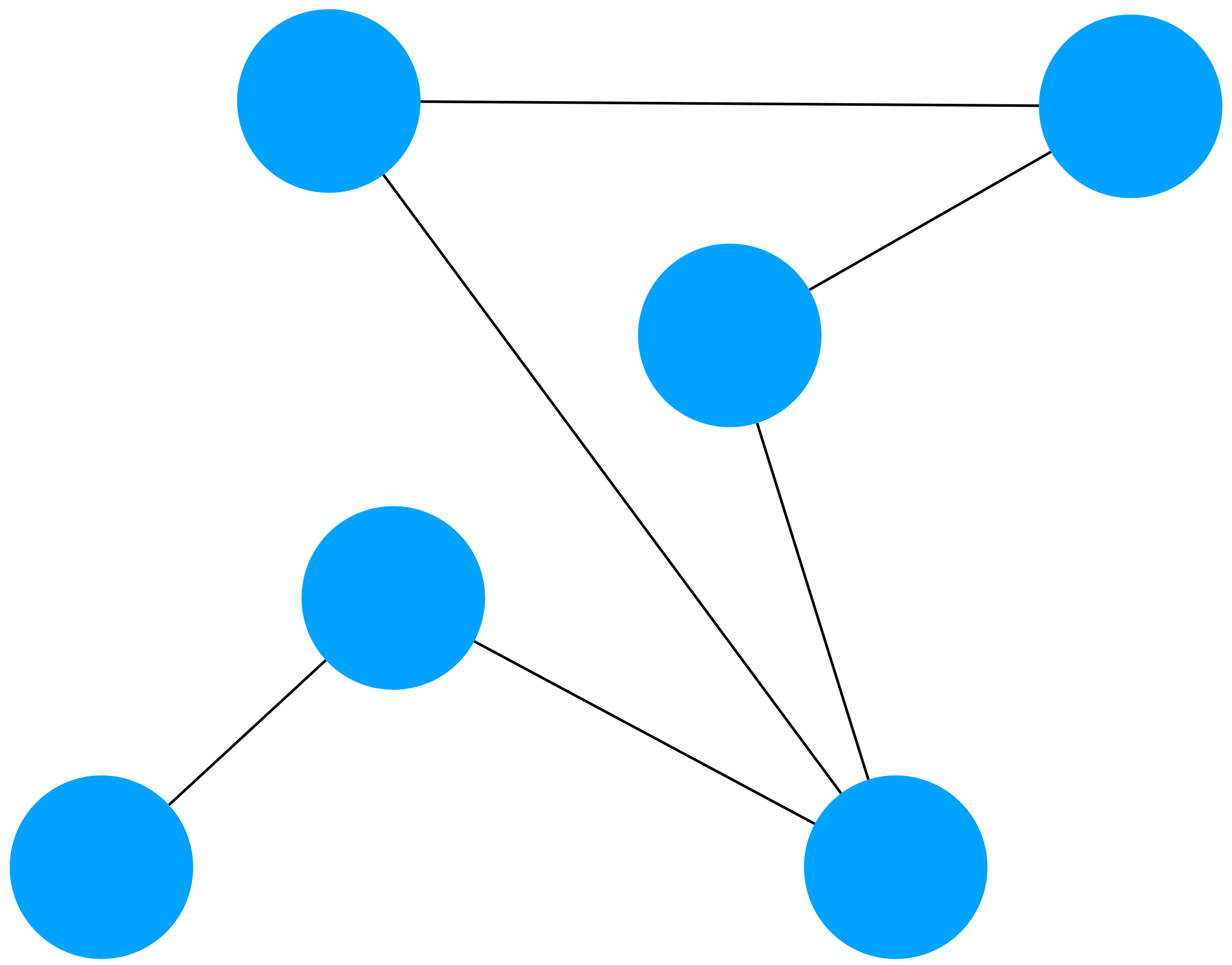


Definition

Let $G = (V, E)$ be a graph. An **elimination ordering** of G is simply an ordering $\sigma = (v_1, \ldots, v_n)$ of V.

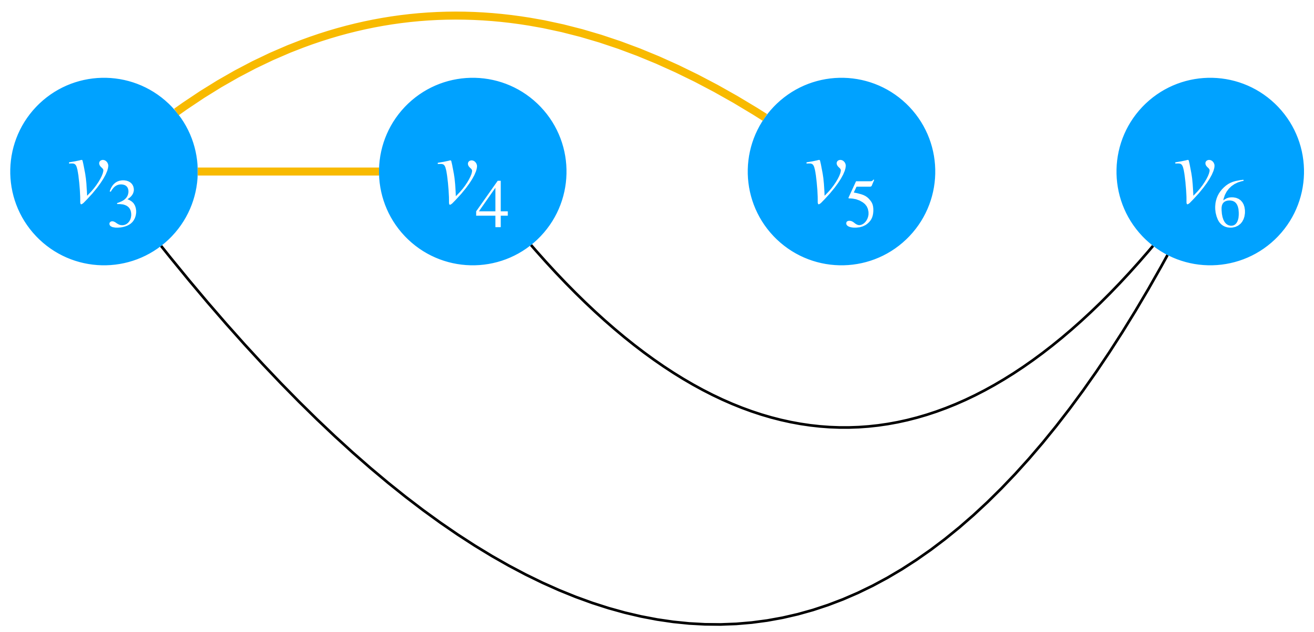


Elimination Orderings

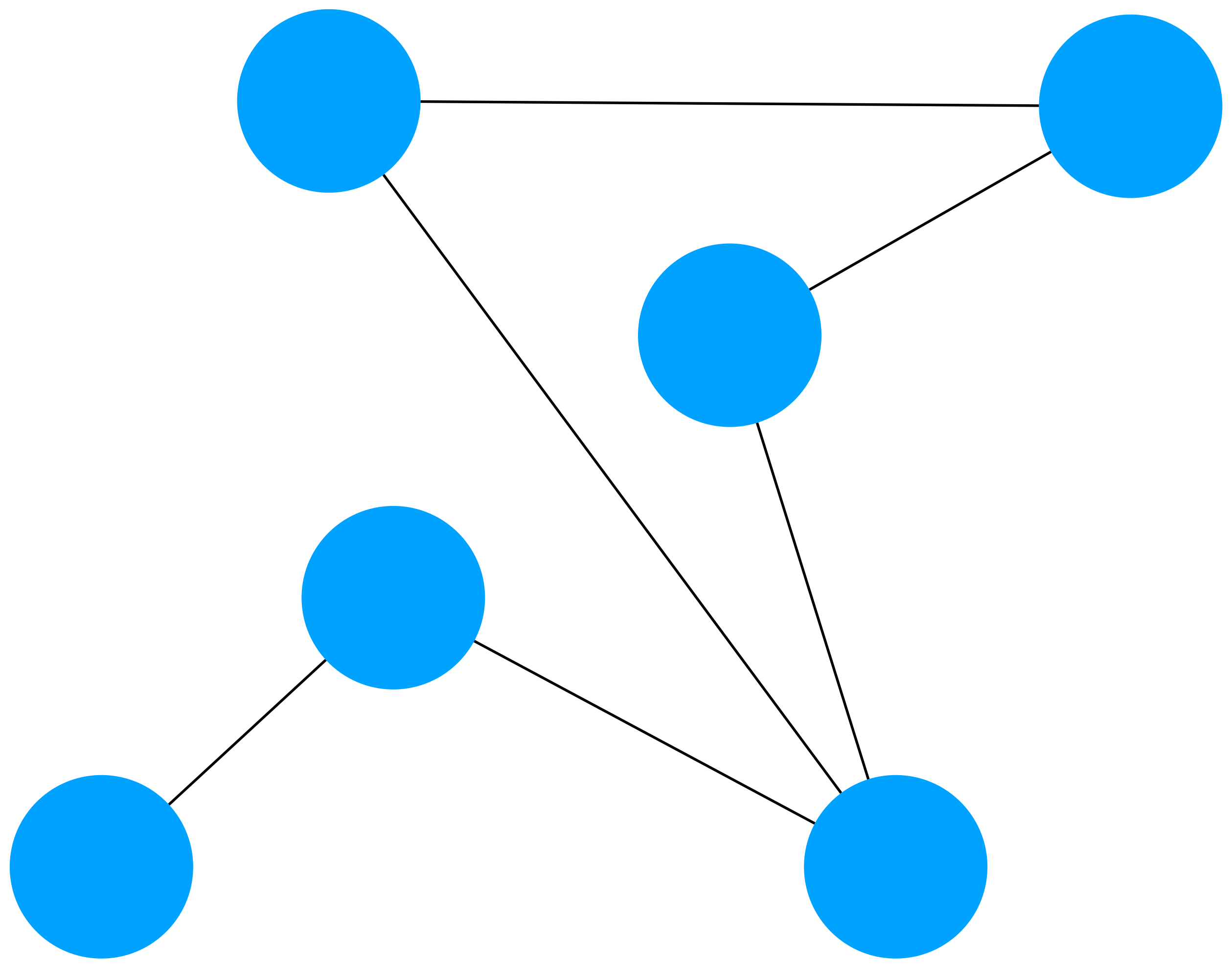


Definition

Let $G = (V, E)$ be a graph. An **elimination ordering** of G is simply an ordering $\sigma = (v_1, \ldots, v_n)$ of V.

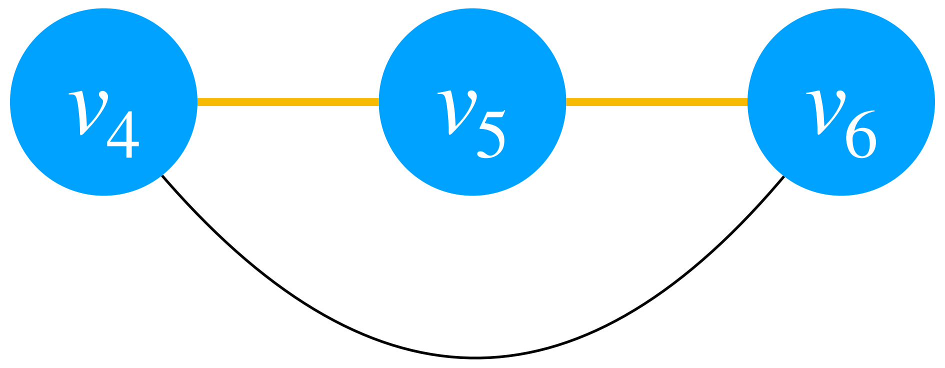


Elimination Orderings

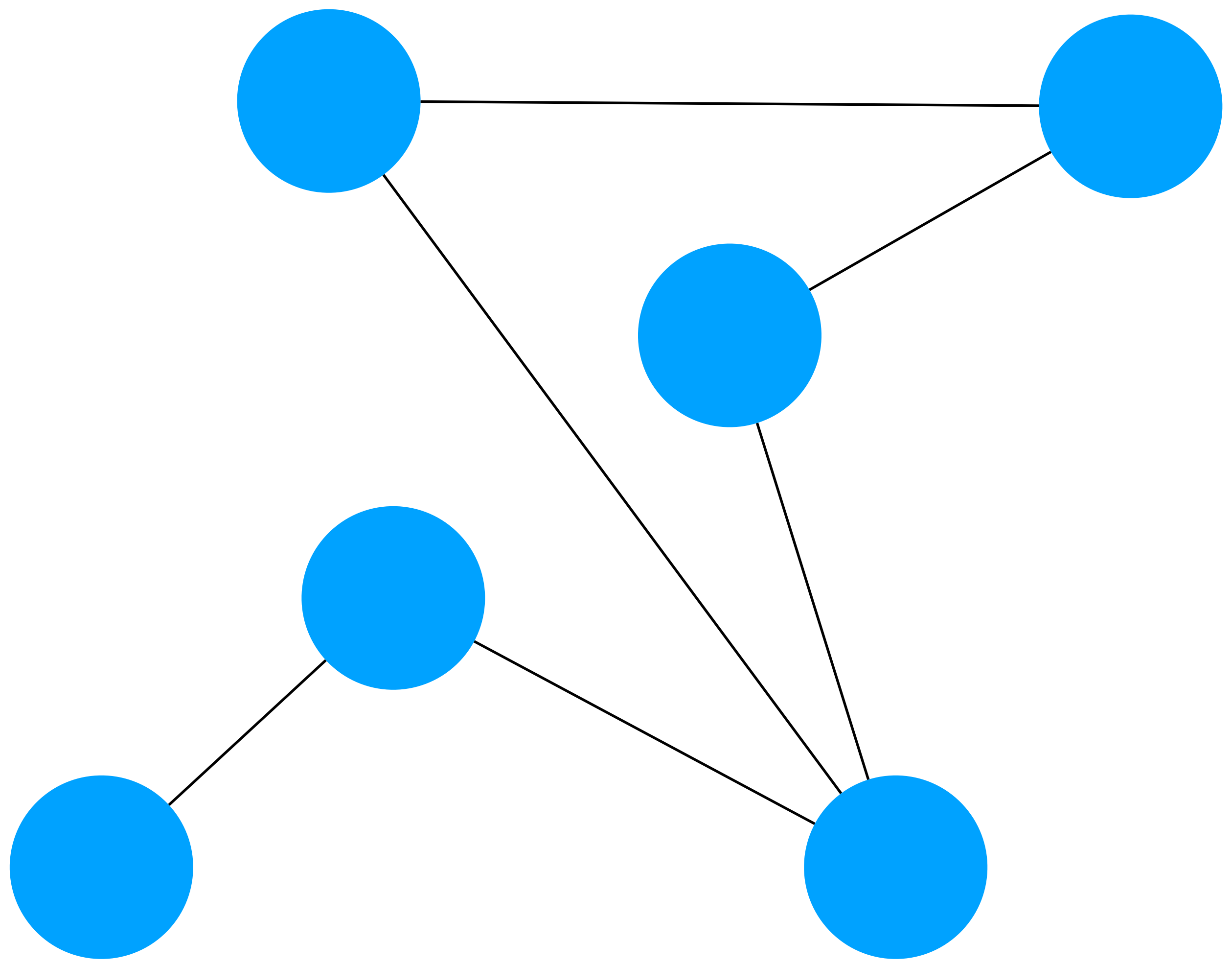


Definition

Let $G = (V, E)$ be a graph. An **elimination ordering** of G is simply an ordering $\sigma = (v_1, \ldots, v_n)$ of V.

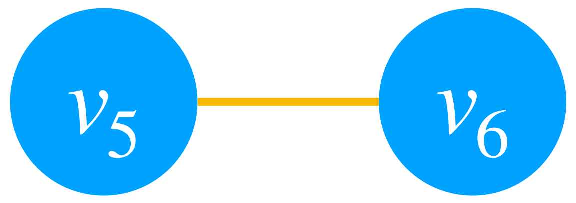


Elimination Orderings

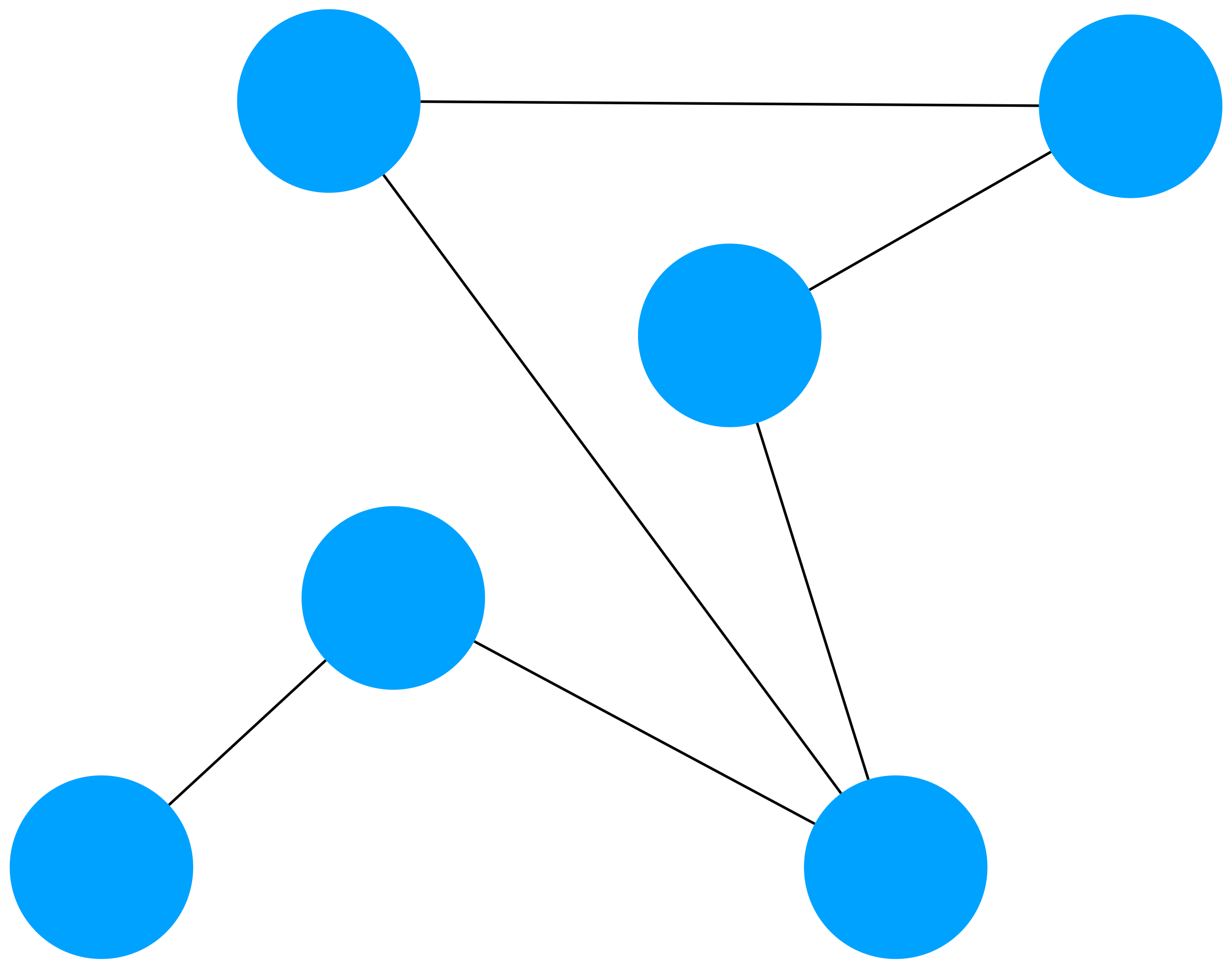


Definition

Let $G = (V, E)$ be a graph. An **elimination ordering** of G is simply an ordering $\sigma = (v_1, \ldots, v_n)$ of V.



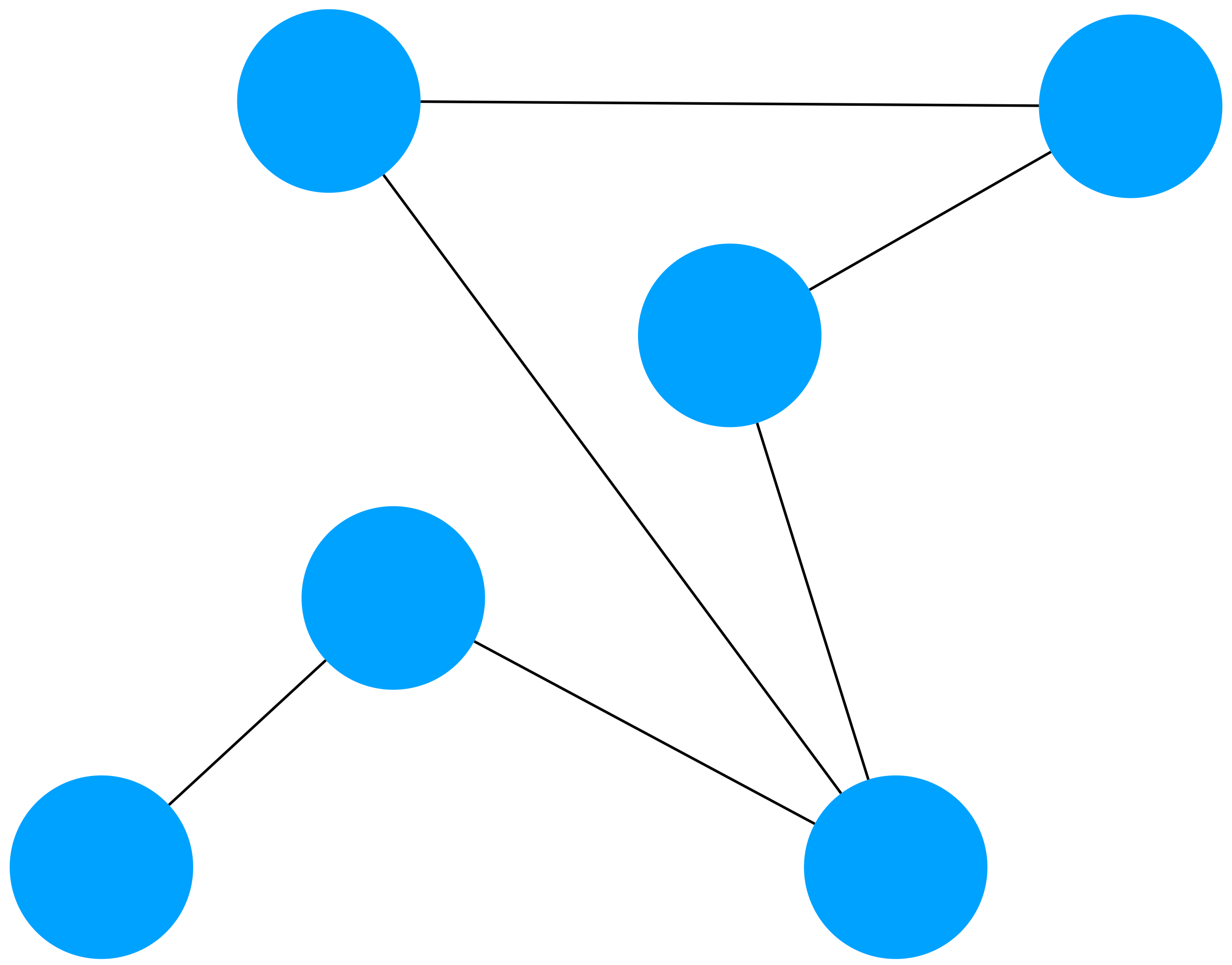
Elimination Orderings



Definition

Let $G = (V, E)$ be a graph. An **elimination ordering** of G is simply an ordering $\sigma = (v_1, \ldots, v_n)$ of V.

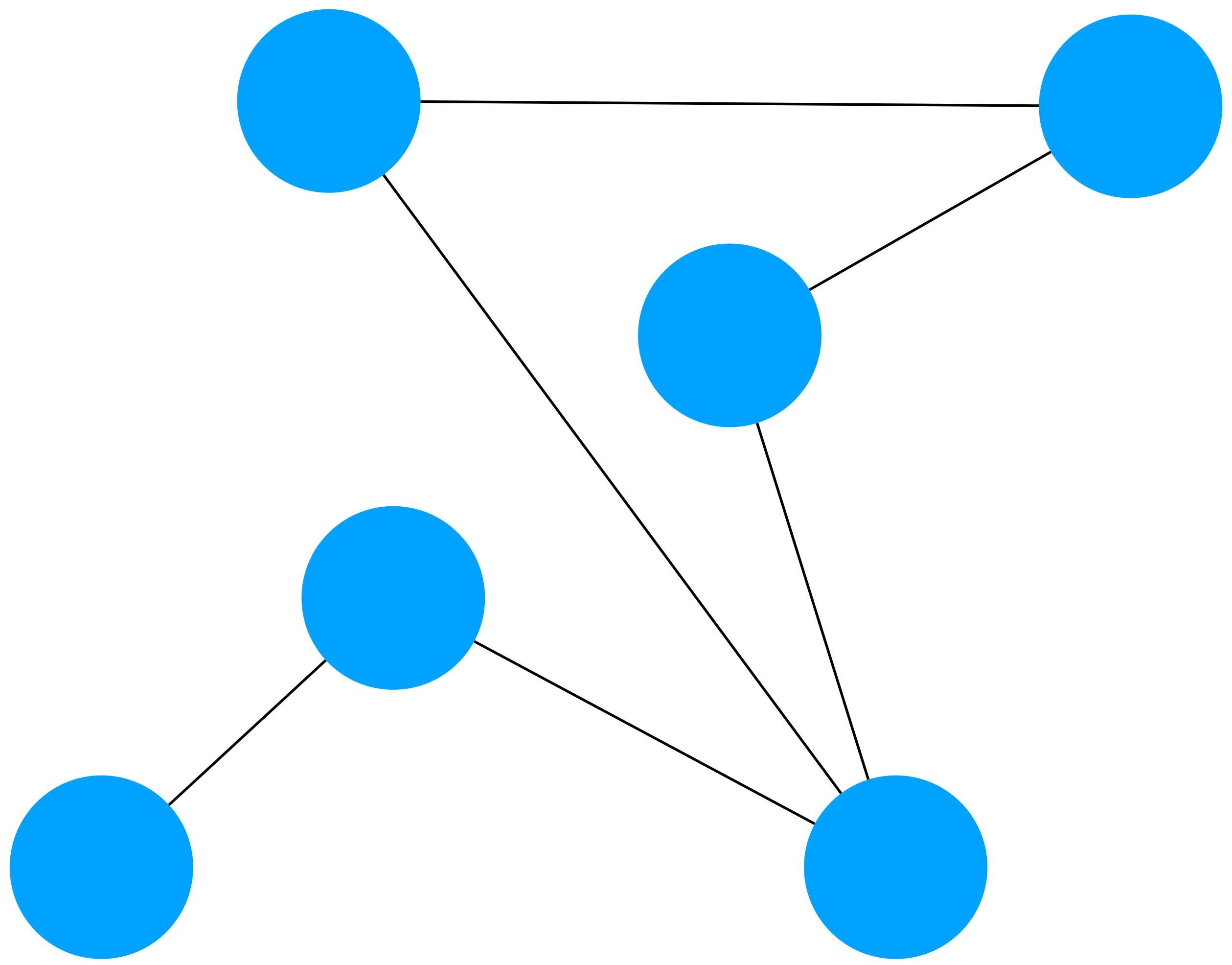
Elimination Orderings



Definition

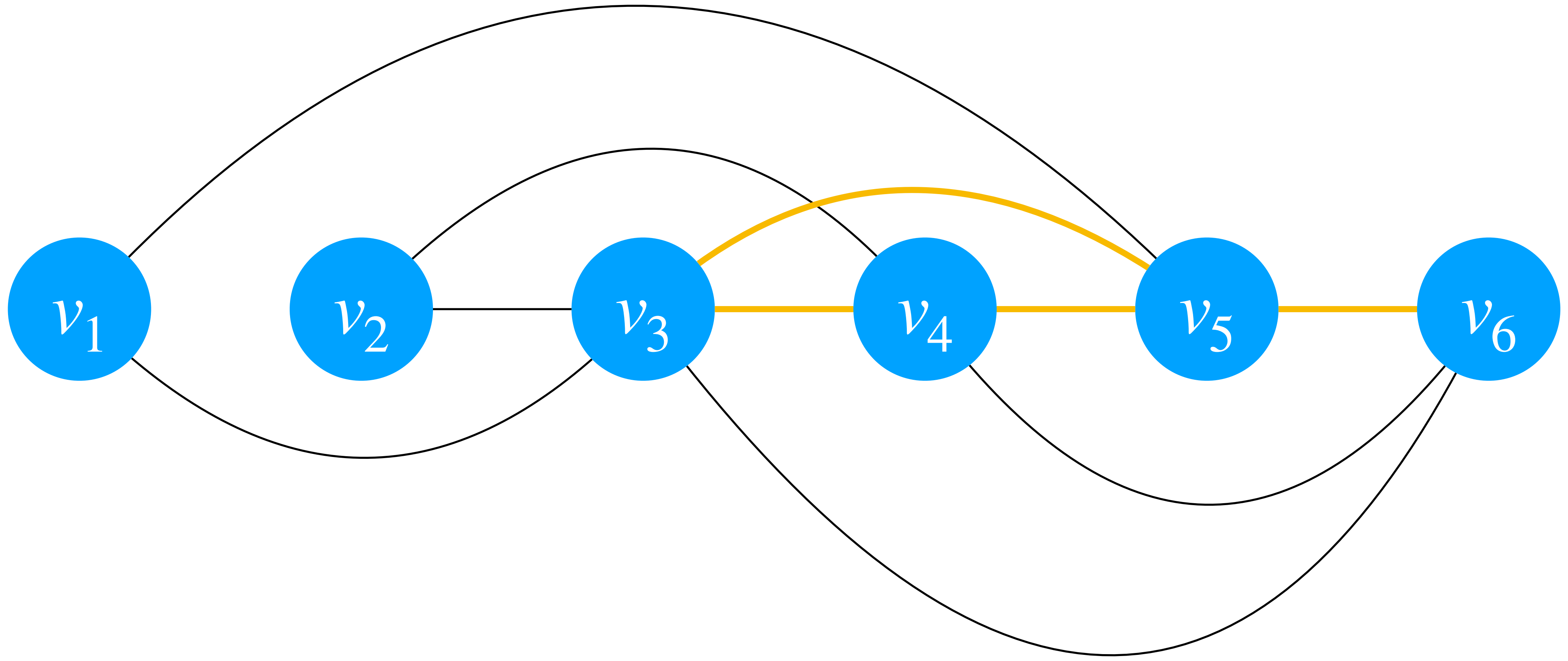
Let $G = (V, E)$ be a graph. An **elimination ordering** of G is simply an ordering $\sigma = (v_1, \ldots, v_n)$ of V.

Elimination Orderings



Definition

Let $G = (V, E)$ be a graph. An **elimination ordering** of G is simply an ordering $\sigma = (v_1, \dots, v_n)$ of V.



The **width** of σ is the maximum degree of a vertex upon elimination.

Tree Decompositions from Orderings

Tree Decompositions from Orderings

DecompositionFromOrdering($G, \langle v_1, \ldots, v_n \rangle$):

Tree Decompositions from Orderings

DecompositionFromOrdering($G, \langle v_1, \ldots, v_n \rangle$**):**

if $|V(G)| = 1$

Tree Decompositions from Orderings

DecompositionFromOrdering($G, \langle v_1, \ldots, v_n \rangle$**):**

if $|V(G)| = 1$

 return $T = (\{t\}, \varnothing), \chi = \{t \mapsto \{v_1\}\}$

Tree Decompositions from Orderings

DecompositionFromOrdering($G, \langle v_1, \ldots, v_n \rangle$**):**

if $|V(G)| = 1$

 return $T = (\{t\}, \varnothing), \chi = \{t \mapsto \{v_1\}\}$

G' := eliminate v_1 from G

Tree Decompositions from Orderings

DecompositionFromOrdering($G, \langle v_1, \ldots, v_n \rangle$**):**

if $|V(G)| = 1$

return $T = (\{t\}, \emptyset), \chi = \{t \mapsto \{v_1\}\}$

$G' :=$ eliminate v_1 from G

$T, \chi :=$ **DecompositionFromOrdering**($G', \langle v_2, \ldots, v_n \rangle$)

Tree Decompositions from Orderings

DecompositionFromOrdering($G, \langle v_1, \ldots, v_n \rangle$**):**

if $|V(G)| = 1$

 return $T = (\{t\}, \emptyset), \chi = \{t \mapsto \{v_1\}\}$

$G' :=$ eliminate v_1 from G

 $T, \chi :=$ **DecompositionFromOrdering**$(G', \langle v_2, \ldots, v_n \rangle)$

 $t :=$ node of T such that $N_G(v_1) \subseteq \chi(t)$

Tree Decompositions from Orderings

DecompositionFromOrdering($G, \langle v_1, \ldots, v_n \rangle$**):**

if $|V(G)| = 1$

return $T = (\{t\}, \emptyset), \chi = \{t \mapsto \{v_1\}\}$

G' := eliminate v_1 from G

T, χ := **DecompositionFromOrdering**($G', \langle v_2, \ldots, v_n \rangle$)

t := node of T such that $N_G(v_1) \subseteq \chi(t)$

t' := new node

Tree Decompositions from Orderings

DecompositionFromOrdering($G, \langle v_1, \ldots, v_n \rangle$**):**

if $|V(G)| = 1$

  return $T = (\{t\}, \emptyset), \chi = \{t \mapsto \{v_1\}\}$

$G' :=$ eliminate v_1 from G

  $T, \chi :=$ **DecompositionFromOrdering**$(G', \langle v_2, \ldots, v_n \rangle)$

  $t :=$ node of T such that $N_G(v_1) \subseteq \chi(t)$

  $t' :=$ new node

  $\chi' := \chi \cup \{t' \mapsto N_G[v_1]\}$

Tree Decompositions from Orderings

DecompositionFromOrdering($G, \langle v_1, \ldots, v_n \rangle$**):**

if $|V(G)| = 1$

  return $T = (\{t\}, \emptyset), \chi = \{t \mapsto \{v_1\}\}$

G' := eliminate v_1 from G

  T, χ := **DecompositionFromOrdering**($G', \langle v_2, \ldots, v_n \rangle$)

  t := node of T such that $N_G(v_1) \subseteq \chi(t)$

  t' := new node

  $\chi' := \chi \cup \{t' \mapsto N_G[v_1]\}$

  T' := add the edge tt' to T

Tree Decompositions from Orderings

DecompositionFromOrdering($G, \langle v_1, \ldots, v_n \rangle$**):**

if $|V(G)| = 1$

  return $T = (\{t\}, \emptyset), \chi = \{t \mapsto \{v_1\}\}$

G' := eliminate v_1 from G

  T, χ := **DecompositionFromOrdering**($G', \langle v_2, \ldots, v_n \rangle$)

  t := node of T such that $N_G(v_1) \subseteq \chi(t)$

  t' := new node

  $\chi' := \chi \cup \{t' \mapsto N_G[v_1]\}$

  T' := add the edge tt' to T

return T', χ'

Greedy Heuristics

Greedy Heuristics

GreedyOrderingX(G):

Greedy Heuristics

GreedyOrderingX(G):

$\pi := ()$

Greedy Heuristics

GreedyOrderingX(G):

$\pi := ()$

for $i := 1$ **to** n

Greedy Heuristics

GreedyOrderingX(G):

$\pi := ()$

for $i := 1$ **to** n

$v :=$ vertex of G **optimal with respect to X**

Greedy Heuristics

GreedyOrderingX(G):

$\pi := ()$

for $i := 1$ **to** n

$v :=$ vertex of G **optimal with respect to X**

$\pi := \pi, v$

Greedy Heuristics

GreedyOrderingX(G):

$\pi := ()$

for $i := 1$ **to** n

  $v :=$ vertex of G **optimal with respect to X**

  $\pi := \pi, v$

  $G :=$ graph obtained from G by eliminating v

Greedy Heuristics

GreedyOrderingX(G)**:**

$\pi := ()$

for $i := 1$ **to** n

$v :=$ vertex of G **optimal with respect to X**

$\pi := \pi, v$

$G :=$ graph obtained from G by eliminating v

return π

Greedy Heuristics

GreedyOrderingX(G):

$\pi := ()$

for $i := 1$ **to** n

  $v :=$ vertex of G **optimal with respect to X**

  $\pi := \pi, v$

  $G :=$ graph obtained from G by eliminating v

return π

1. **Min Degree/Size** minimum degree

Greedy Heuristics

GreedyOrderingX(G)**:**

$\pi := ()$

for $i := 1$ **to** n

$v :=$ vertex of G **optimal with respect to X**

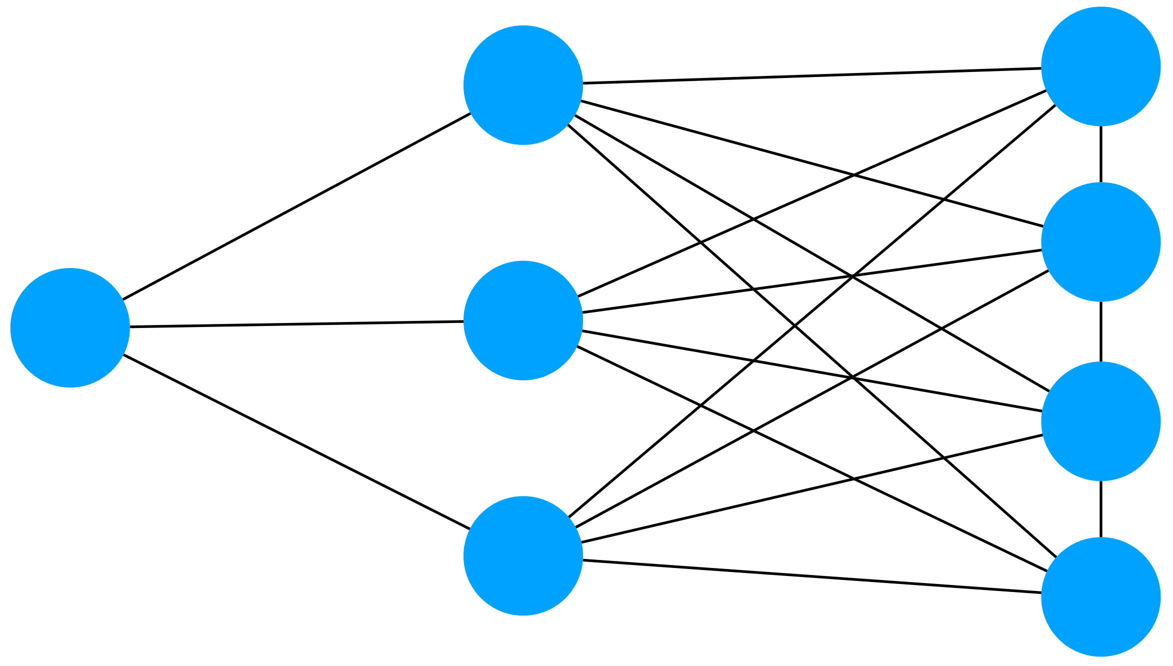
$\pi := \pi, v$

$G :=$ graph obtained from G by eliminating v

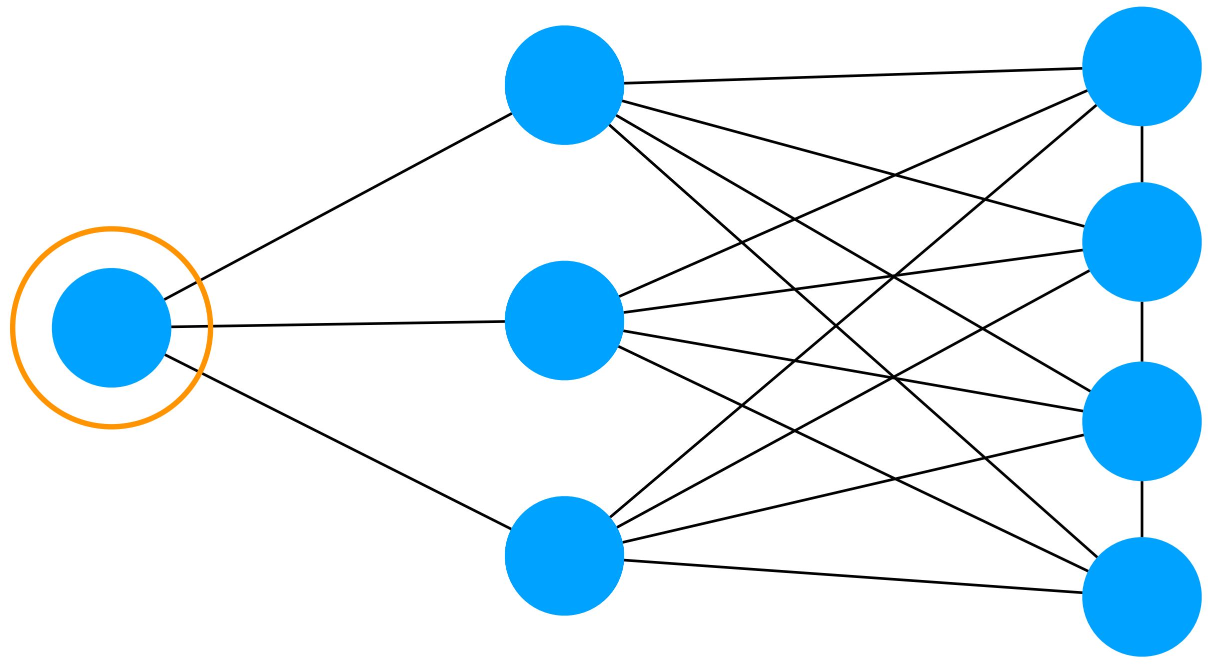
return π

1. **Min Degree/Size** minimum degree
2. **Min Fill** fewest fill-in edges

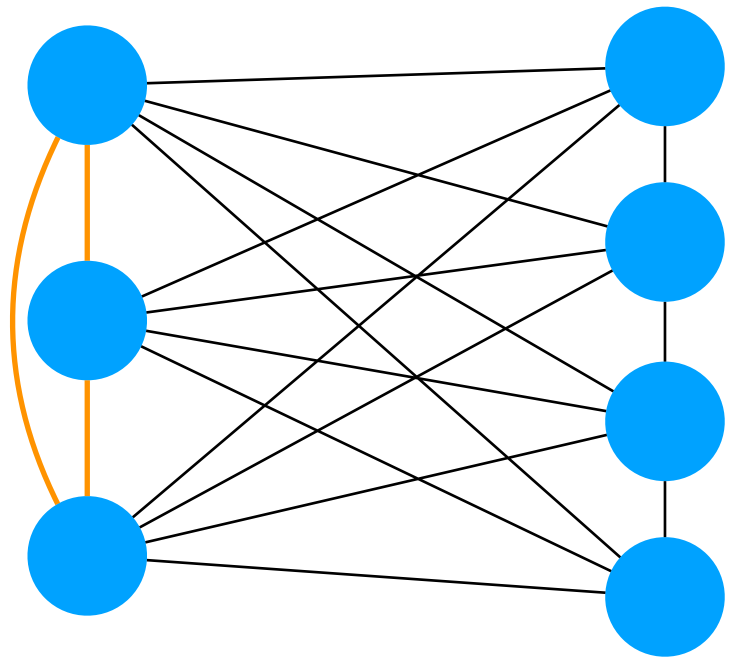
Min Degree and Min Fill



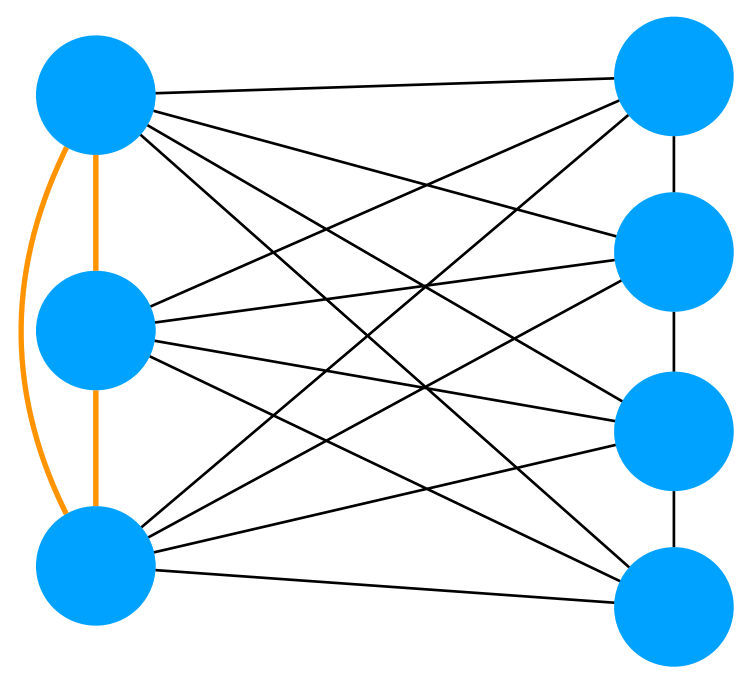
Min Degree and Min Fill

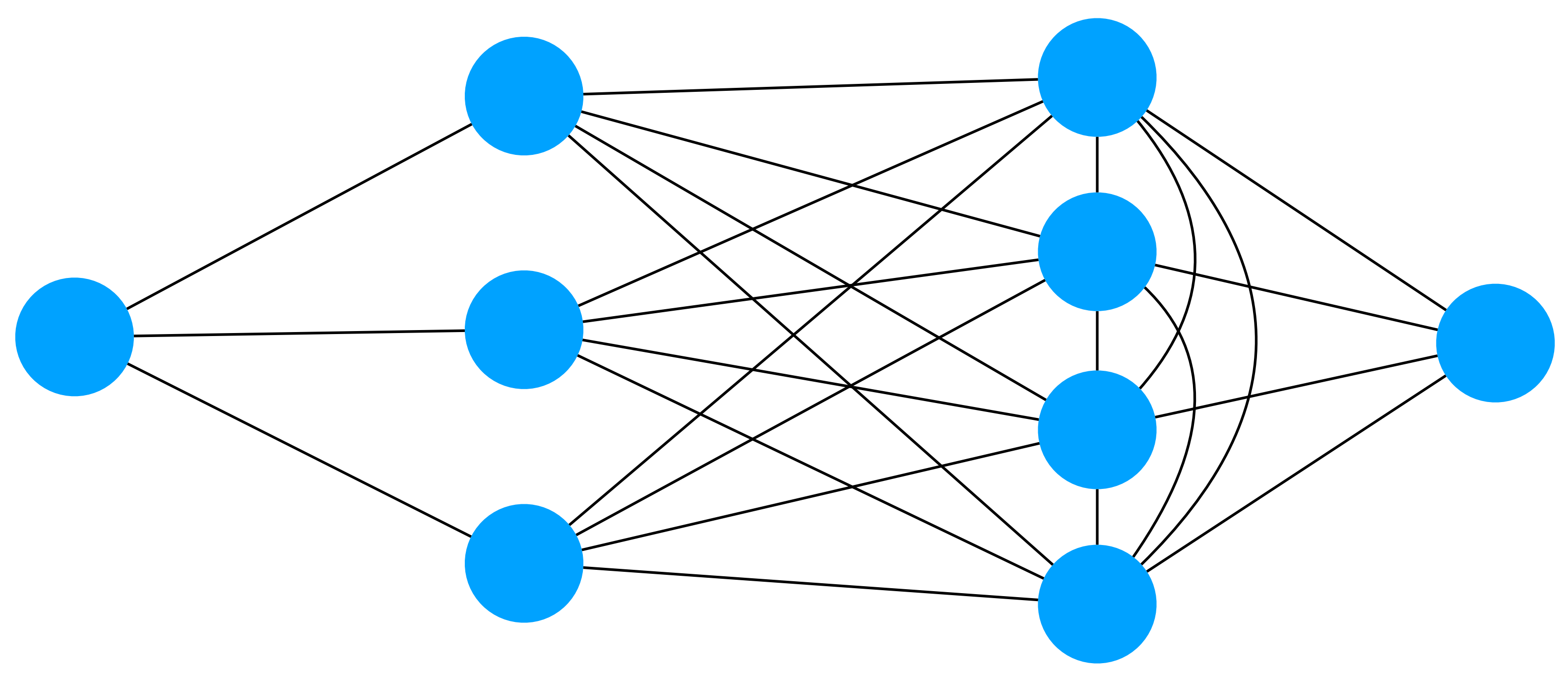


Min Degree and Min Fill

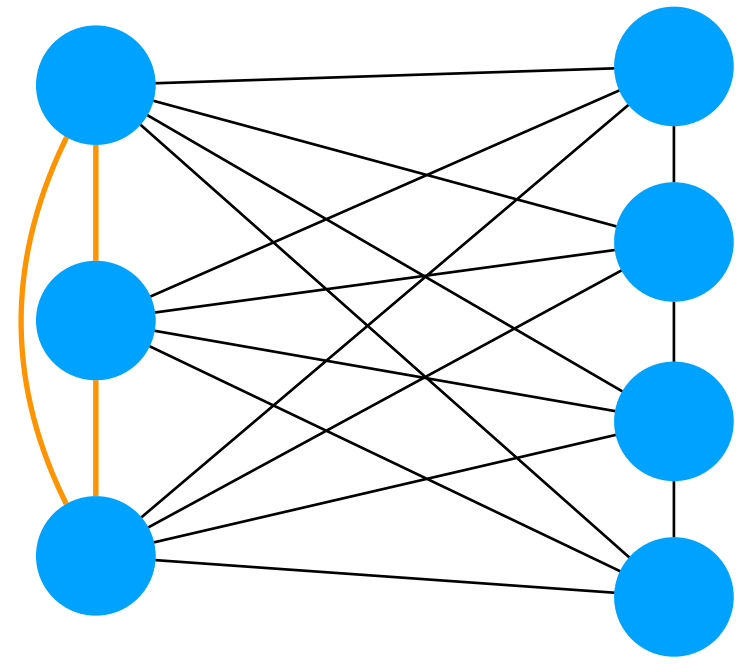


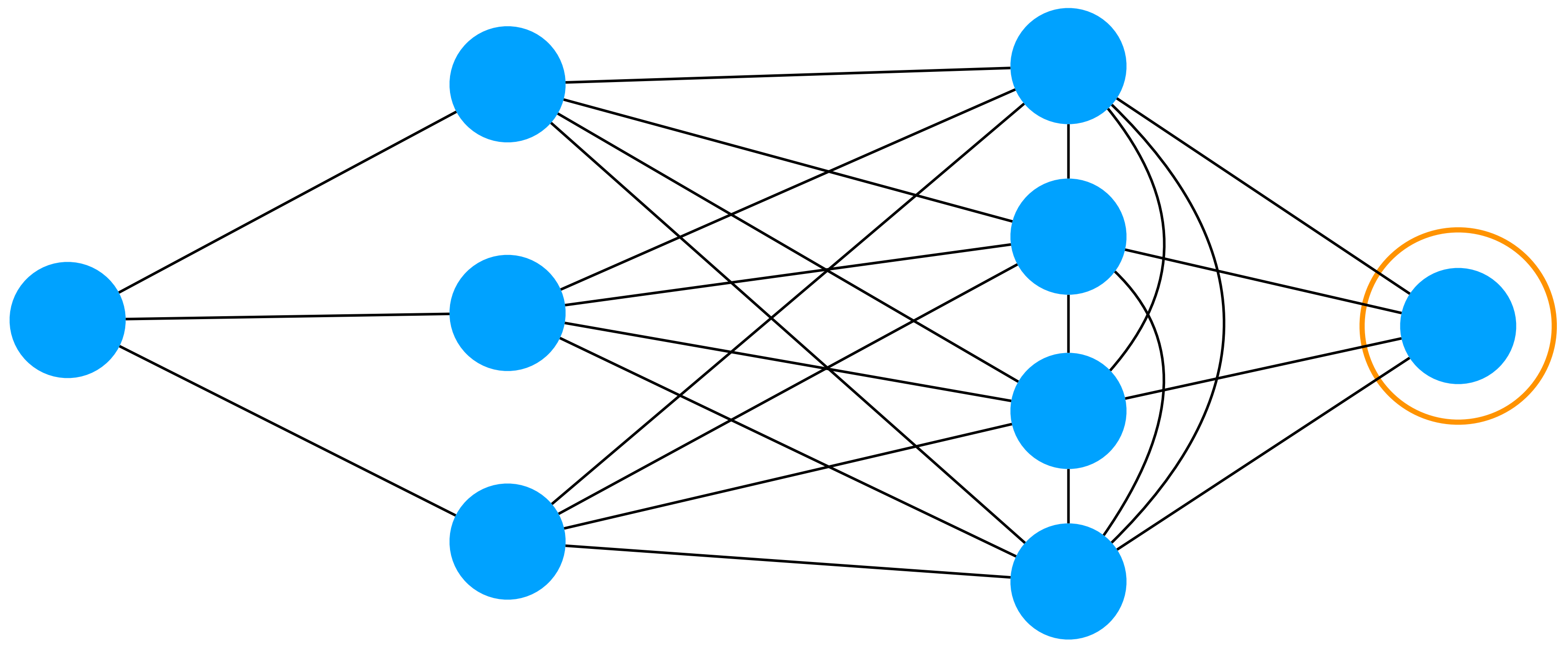
Min Degree and Min Fill



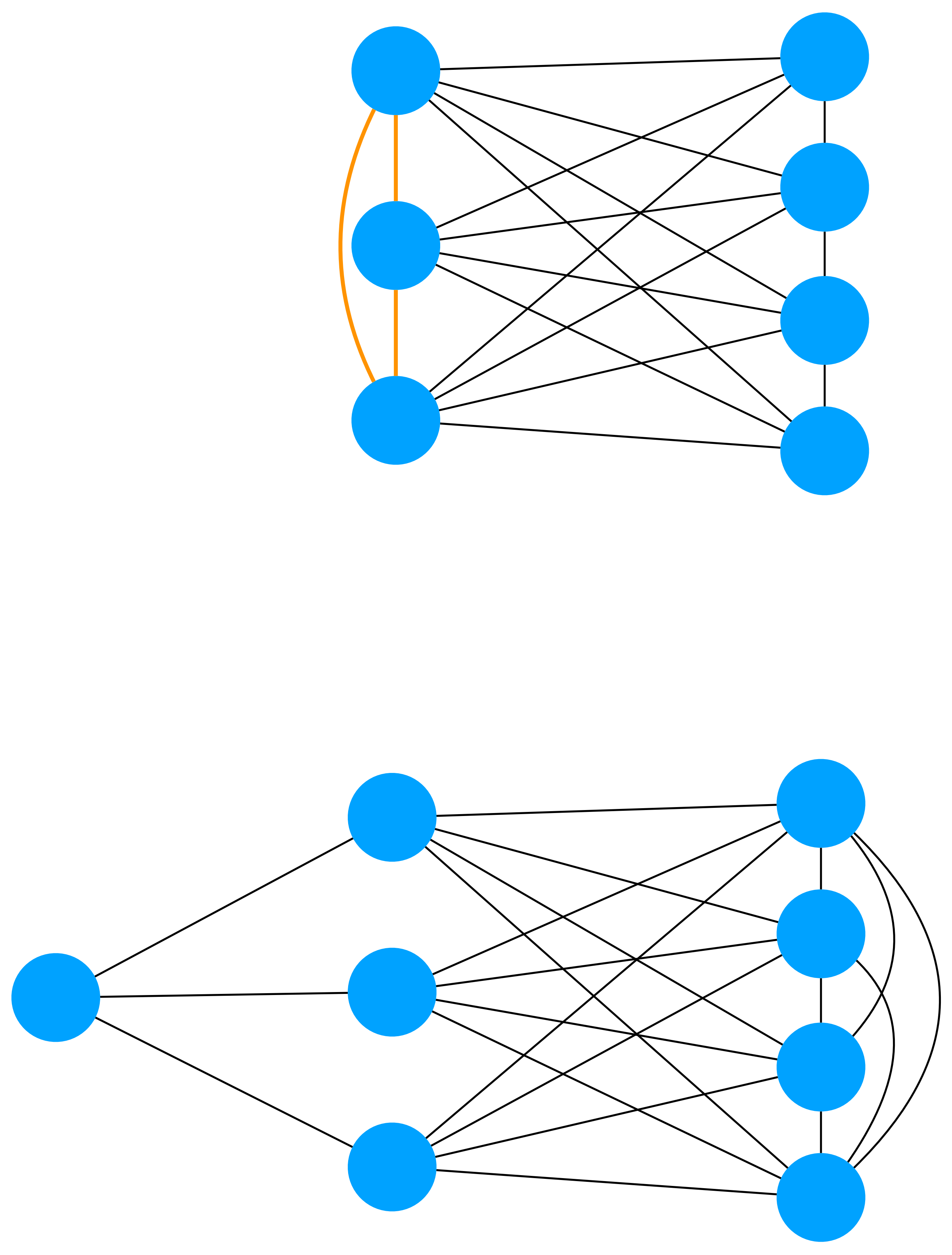


Min Degree and Min Fill





Min Degree and Min Fill



Maximum Cardinality Search

Maximum Cardinality Search

MCS(G):

Maximum Cardinality Search

MCS(G):

$\pi := ()$

Maximum Cardinality Search

MCS(G):

$\pi := ()$

for $i := n$ **to** 1

Maximum Cardinality Search

MCS(G):

$\pi := ()$

for $i := n$ **to** 1

$v :=$ vertex of G **with most neighbors in π**

Maximum Cardinality Search

MCS(G):

$\pi := ()$

for $i := n$ **to** 1

  $v :=$ vertex of G **with most neighbors in π**

  $\pi := v, \pi$

Maximum Cardinality Search

MCS(G):
$\pi := ()$
for $i := n$ **to** 1
  $v :=$ vertex of G **with most neighbors in π**
  $\pi := v, \pi$
return π

Heuristics in Practice

Heuristics in Practice

- These heuristics do **surprisingly well** in practice.

Heuristics in Practice

- These heuristics do **surprisingly well** in practice.
- **Min-Fill** generates better (in terms of width) orderings.

Heuristics in Practice

- These heuristics do **surprisingly well** in practice.
- **Min-Fill** generates better (in terms of width) orderings.
- Can generate good elimination orderings for large graphs.

Heuristics in Practice

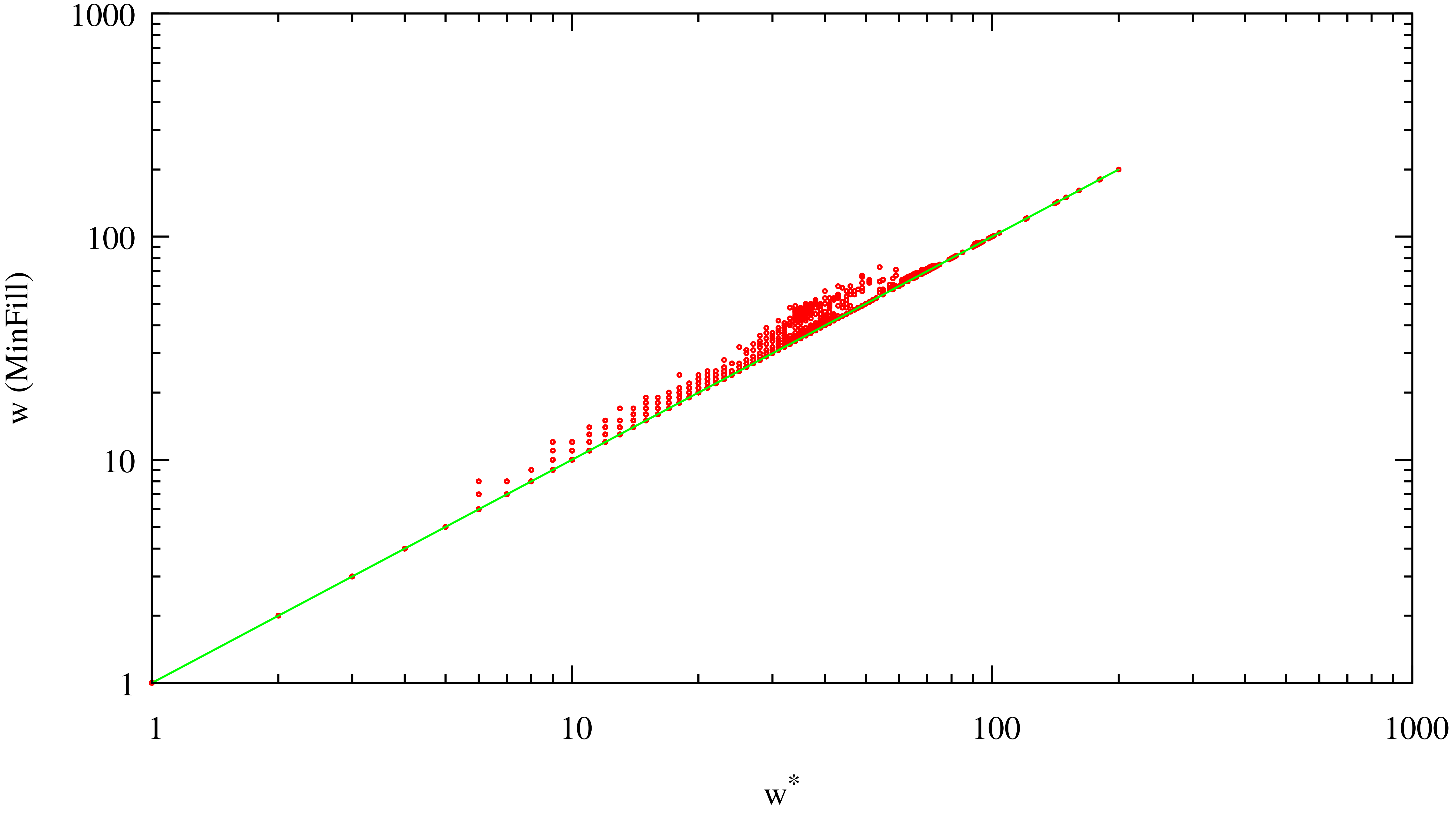


Fig. 2. The width obtained with MinFill vs the treewidth.

Jegou et al. ICTAI 2018