Algorithmic Meta-Theorems

192.122

WS21/22

Jan Dreier

dreier@ac.tuwien.ac.at

ALGORITHMS AND COMPLEXITY GROUP

TU WIEN Informatics

Announcements

- Next week on 26.10. no lecture (national holiday)

- There will be two exercise sessions
  - every other Friday, 9:15 online,
  - every other Friday, 11:00 in peson.

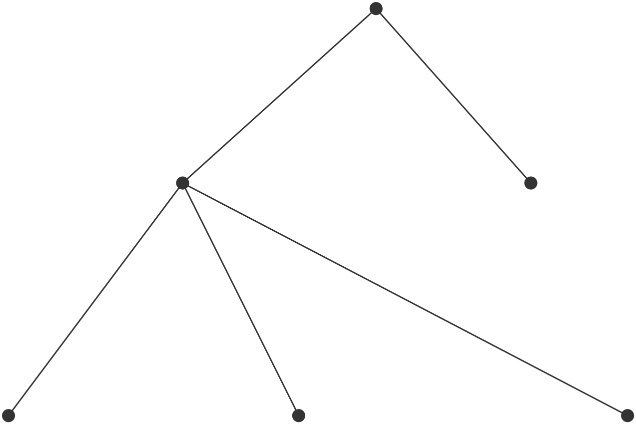
  Choose yourself which one to attend.

  First session will be on 29.10.

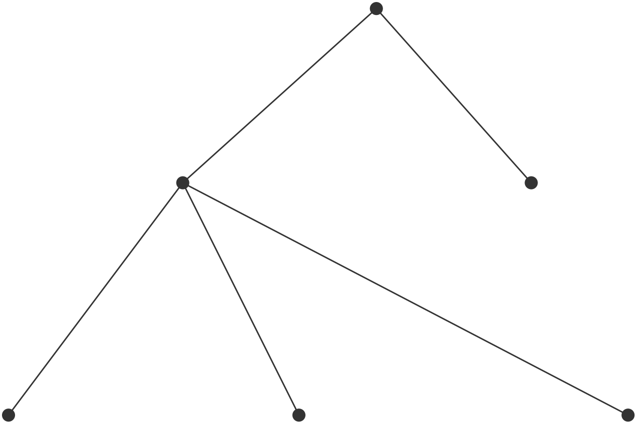
Outline

Last lecture, we solved problems on trees. Today, we introduce treewidth and learn how to solve problems on graphs with small treewidth.

Independent Set on Trees



Independent Set on Trees

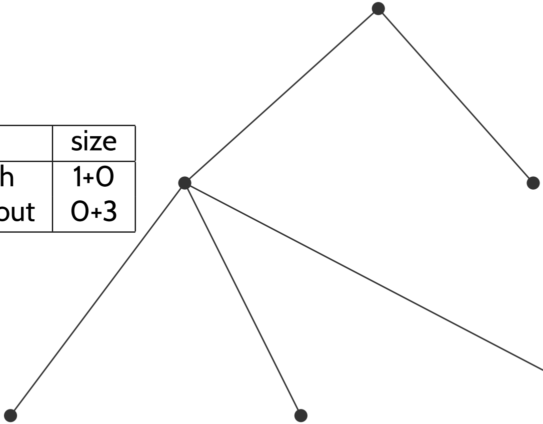


	size
with	1
without	0

	size
with	1
without	0

	size
with	1
without	0

Independent Set on Trees



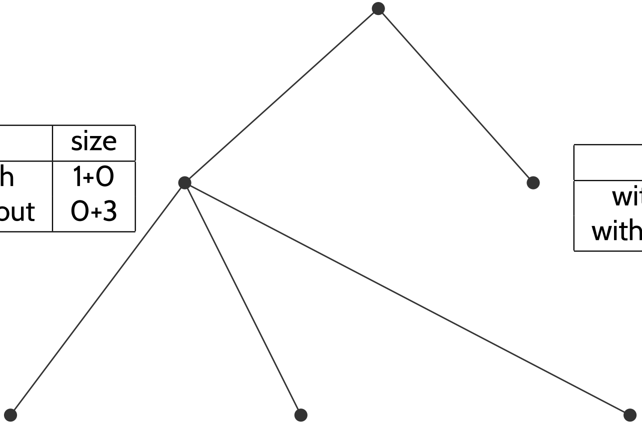
	size
with	1+0
without	0+3

	size
with	1
without	0

	size
with	1
without	0

	size
with	1
without	0

Independent Set on Trees



	size
with	1+0
without	0+3

	size
with	1
without	0

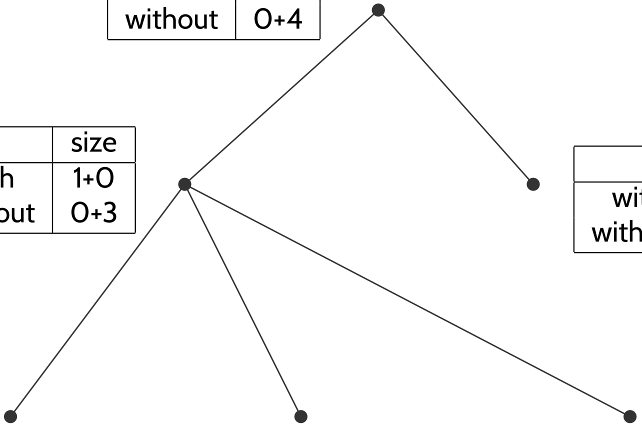
	size
with	1
without	0

	size
with	1
without	0

	size
with	1
without	0

Independent Set on Trees

	size
with	1+3
without	0+4



	size
with	1+0
without	0+3

	size
with	1
without	0

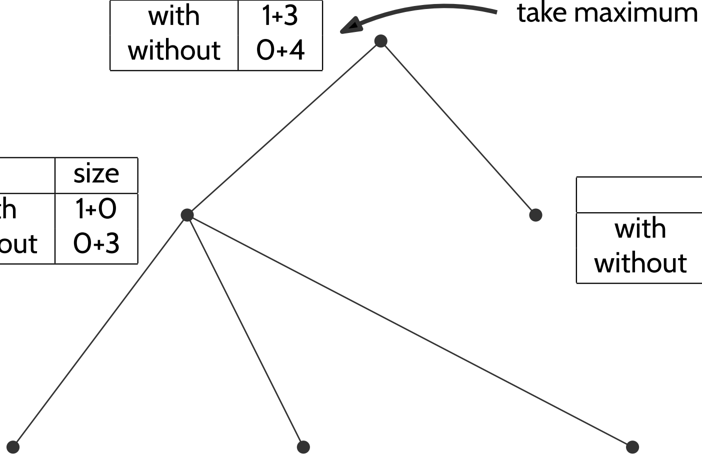
	size
with	1
without	0

	size
with	1
without	0

	size
with	1
without	0

Independent Set on Trees

	size
with	1+3
without	0+4



	size
with	1+0
without	0+3

	size
with	1
without	0

	size
with	1
without	0

	size
with	1
without	0

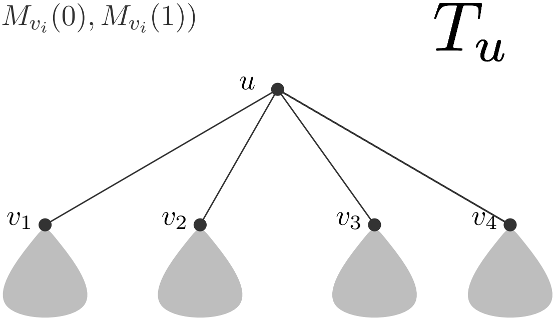
	size
with	1
without	0

Independent Set on Trees

- T_u: subtree at vertex u
- $M_u(1)$: size of maximal IS in T_u that includes u
- $M_u(0)$: size of maximal IS in T_u that excludes u

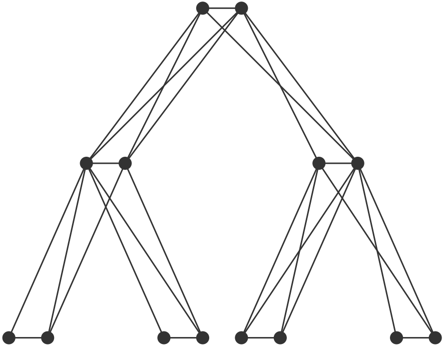
Recursively compute for a vertex u with children $v_1, \ldots, v_k$

- $M_u(1) = 1 + \sum_i M_{v_i}(0)$
- $M_u(0) = \sum_i \max(M_{v_i}(0), M_{v_i}(1))$



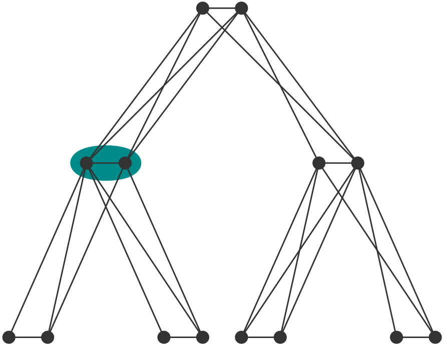
Tree-Like Graphs

- The algorithm works because every vertex is a separator.
- Can we generalize this idea to tree-like graphs?



Tree-Like Graphs

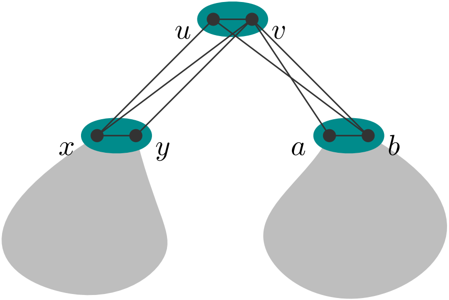
- The algorithm works because every vertex is a separator.
- Can we generalize this idea to tree-like graphs?
- Idea: Group vertices into separator-bags



Independent Set on Tree-Like Graphs

below xy

xy	size
00	5
01	6
10	3
11	$-\infty$



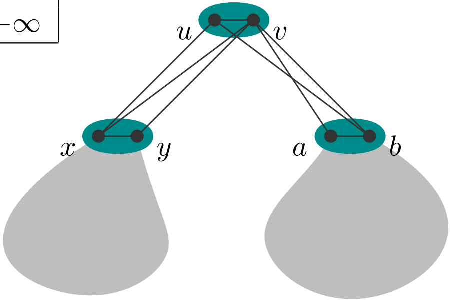
Independent Set on Tree-Like Graphs

left below uv

uv	size
00	6
01	6
10	7
11	$-\infty$

below xy

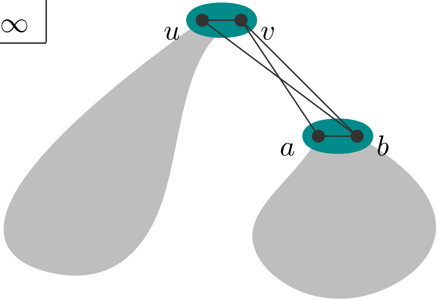
xy	size
00	5
01	6
10	3
11	$-\infty$



Independent Set on Tree-Like Graphs

left below uv

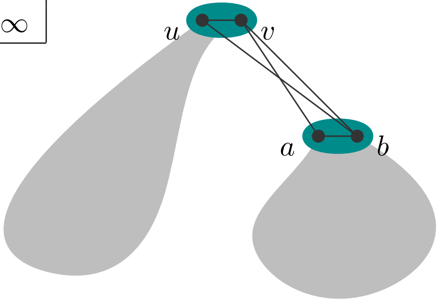
uv	size
00	6
01	6
10	7
11	$-\infty$



Independent Set on Tree-Like Graphs

left below uv

uv	size
00	6
01	6
10	7
11	$-\infty$



below ab

ab	size
00	2
01	3
10	8
11	$-\infty$

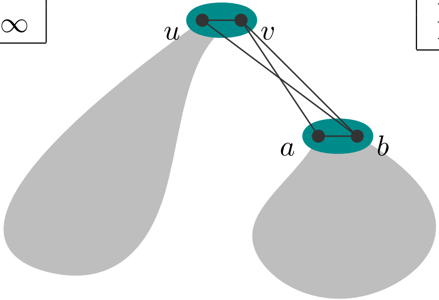
Independent Set on Tree-Like Graphs

left below uv

uv	size
00	6
01	6
10	7
11	$-\infty$

right below uv

uv	size
00	8
01	3
10	9
11	$-\infty$



below ab

ab	size
00	2
01	3
10	8
11	$-\infty$

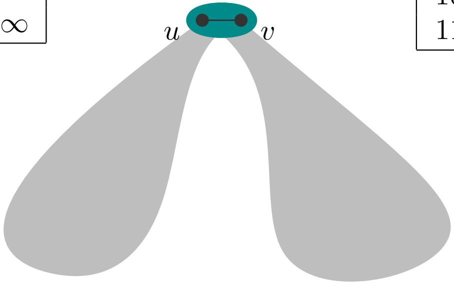
Independent Set on Tree-Like Graphs

left below uv

uv	size
00	6
01	6
10	7
11	$-\infty$

right below uv

uv	size
00	8
01	3
10	9
11	$-\infty$



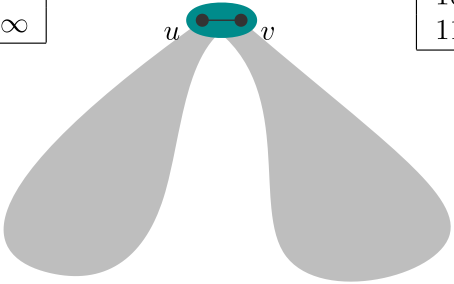
Independent Set on Tree-Like Graphs

below uv

uv	size
00	14
01	8
10	15
11	$-\infty$

left below uv

uv	size
00	6
01	6
10	7
11	$-\infty$



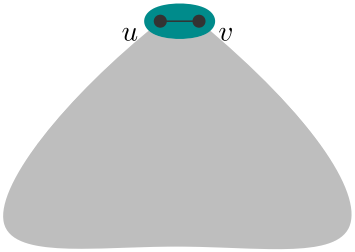
right below uv

uv	size
00	8
01	3
10	9
11	$-\infty$

Independent Set on Tree-Like Graphs

below uv

uv	size
00	14
01	8
10	15
11	$-\infty$



Contemplation

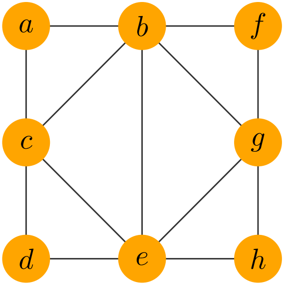
Why does this approach work? Because we have a “tree of small separators” that we can traverse upwards.

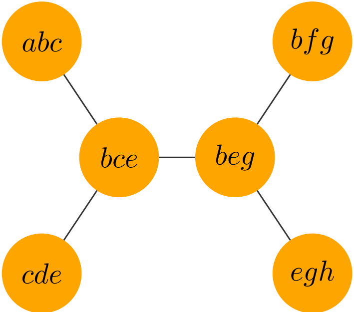
The notion of *treewidth* formalizes this.

Treewidth

A *tree decomposition* of a graph $G = (V, E)$ is a tree whose vertices are *bags* (subsets of V) and

- every vertex $v \in V$ is contained in some bag,
- every edge $uv \in V$ is contained in some bag,
- for every $v \in V$, the bags containing v are a connected subtree

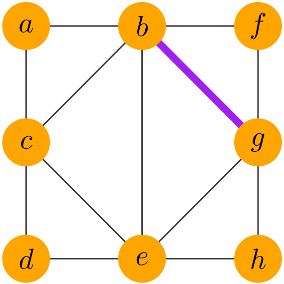


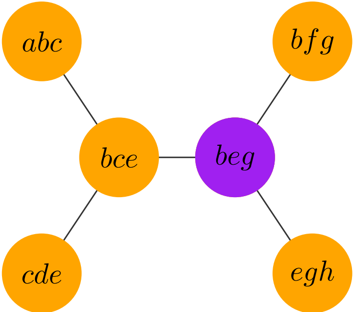


Treewidth

A *tree decomposition* of a graph $G = (V, E)$ is a tree whose vertices are *bags* (subsets of V) and

- every vertex $v \in V$ is contained in some bag,
- every edge $uv \in V$ is contained in some bag,
- for every $v \in V$, the bags containing v are a connected subtree

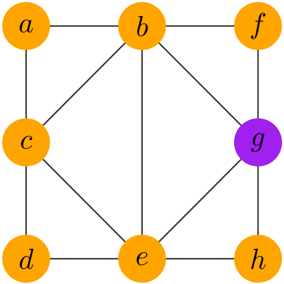


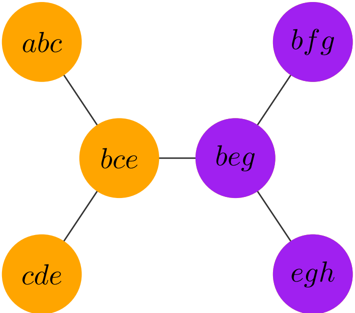


Treewidth

A *tree decomposition* of a graph $G = (V, E)$ is a tree whose vertices are *bags* (subsets of V) and

- every vertex $v \in V$ is contained in some bag,
- every edge $uv \in V$ is contained in some bag,
- for every $v \in V$, the bags containing v are a connected subtree





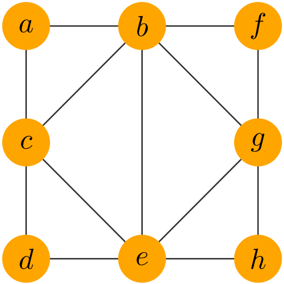
Treewidth

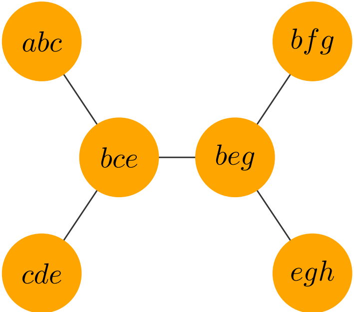
A *tree decomposition* of a graph $G = (V, E)$ is a tree whose vertices are *bags* (subsets of V) and

- every vertex $v \in V$ is contained in some bag,
- every edge $uv \in V$ is contained in some bag,
- for every $v \in V$, the bags containing v are a connected subtree

Width of decomposition: maximum bag size -1

Treewidth of G (tw(G)): minimum width of a decomposition of G





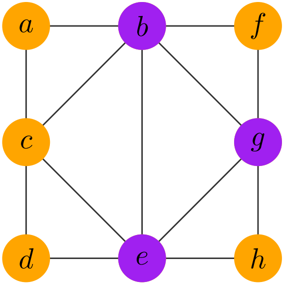
Treewidth

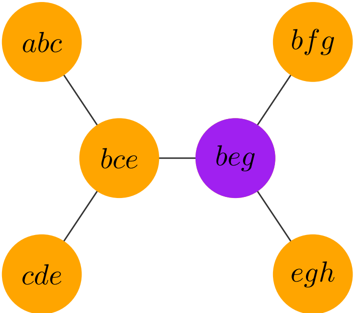
A *tree decomposition* of a graph $G = (V, E)$ is a tree whose vertices are *bags* (subsets of V) and

- every vertex $v \in V$ is contained in some bag,
- every edge $uv \in V$ is contained in some bag,
- for every $v \in V$, the bags containing v are a connected subtree

Width of decomposition: maximum bag size -1

*Treewidth of G (*tw(G)*):* minimum width of a decomposition of G





Trees and Treewidth

Trees have treewidth 1.

Theorem

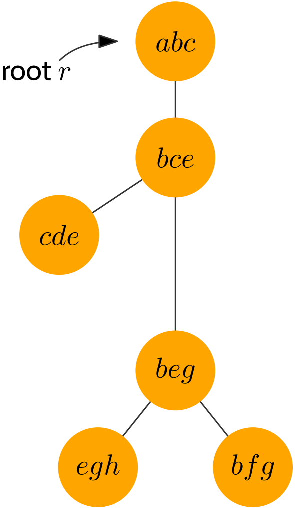
Theorem (Independent Set on Treewidth)

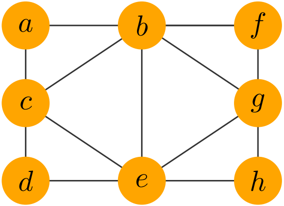
Given a graph G and a tree decomposition of G of width w, one can compute the size of a maximum independent set in time $2^w w^{O(1)} n$.

This generalizes the previous result (trees have treewidth 1).

Rooting the Decomposition

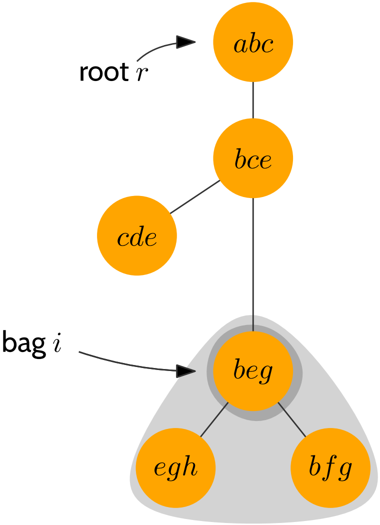
- Choose an arbitrary bag r as root and orient the tree accordingly.

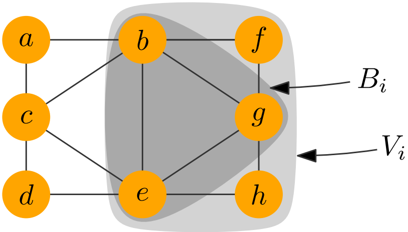




Rooting the Decomposition

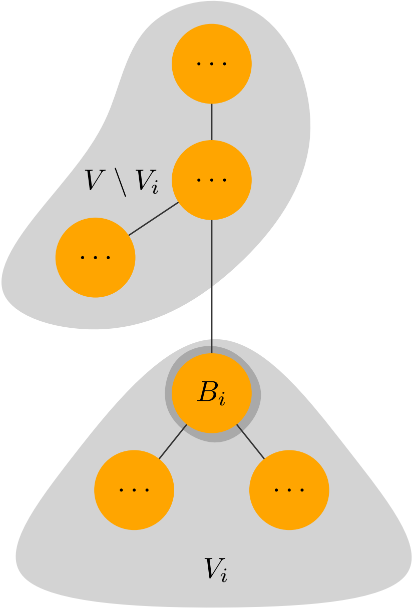
- Choose an arbitrary bag r as root and orient the tree accordingly.
- For bag i let B_i be the vertices of G contained in i and V_i be the vertices contained in i or a successor of i.





Separators

We will compute solutions in V_i w.r.t "boundary" B_i.

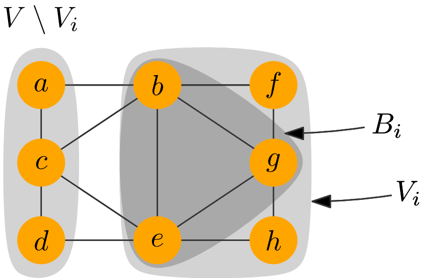


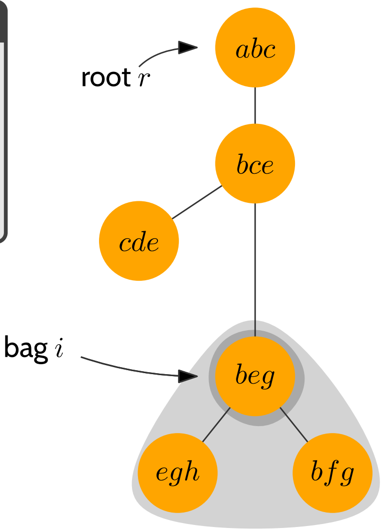
Separators

Theorem

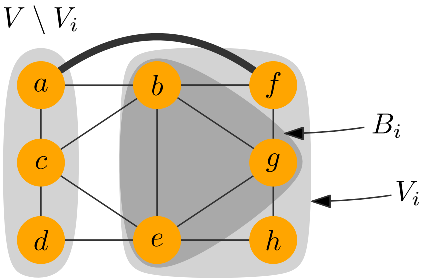
For every bag i, B_i is a separator between V_i and $V \setminus V_i$.

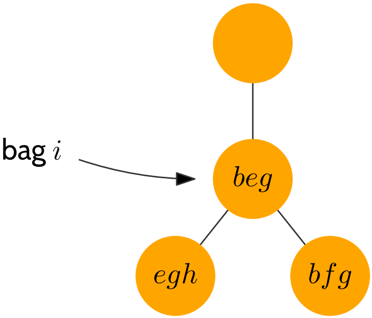
(i.e., every path between V_i and $V \setminus V_i$ goes through B_i).





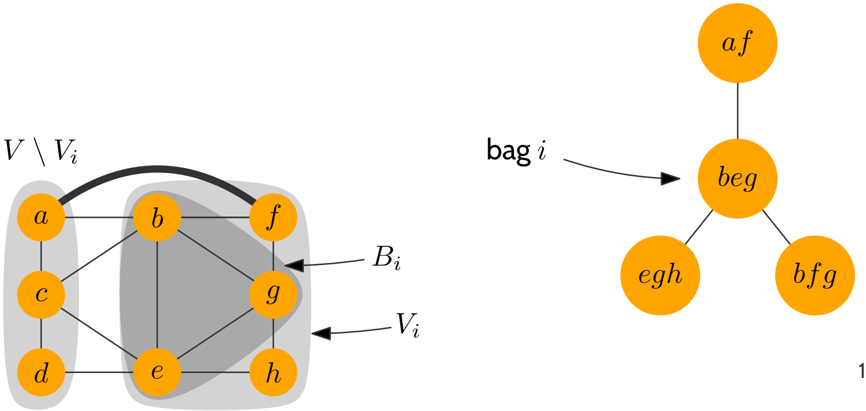
- Assume for contradition the statement is false and there is an edge such as drawn below





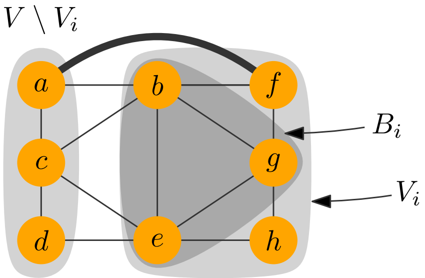
Separators / Proof

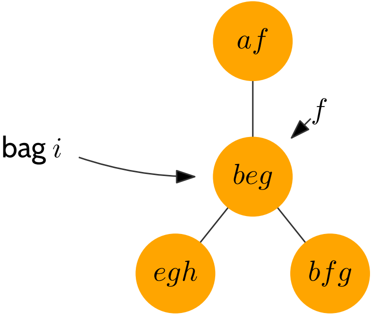
- Assume for contradition the statement is false and there is an edge such as drawn below
- Then a and f occur together in some bag (maybe above i).



Separators / Proof

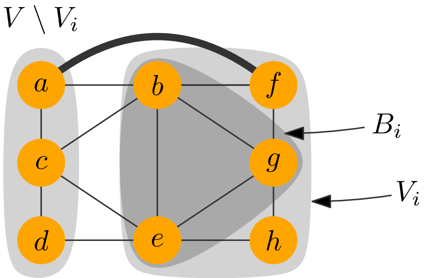
- Assume for contradition the statement is false and there is an edge such as drawn below
- Then a and f occur together in some bag (maybe above i).
- The bags containing f induce a subtree, thus bag i also contains f. A contradiction.

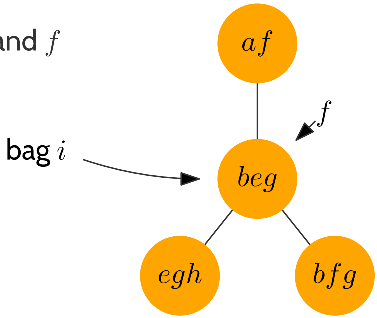




Separators / Proof

- Assume for contradition the statement is false and there is an edge such as drawn below
- Then a and f occur together in some bag (maybe above i).
- The bags containing f induce a subtree, thus bag i also contains f. A contradiction.
- We get a similar contradiction if a and f occur together somewhere else.



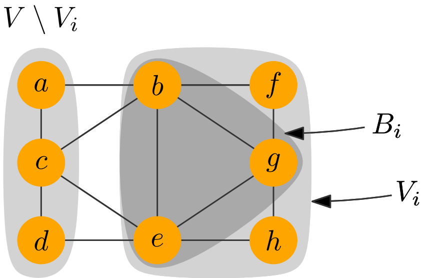


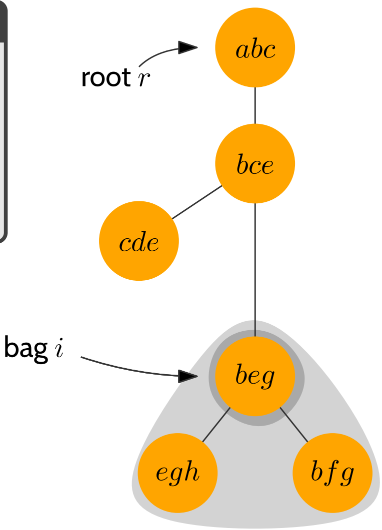
Separators

Theorem

For every bag i, B_i is a separator between V_i and $V \setminus V_i$.

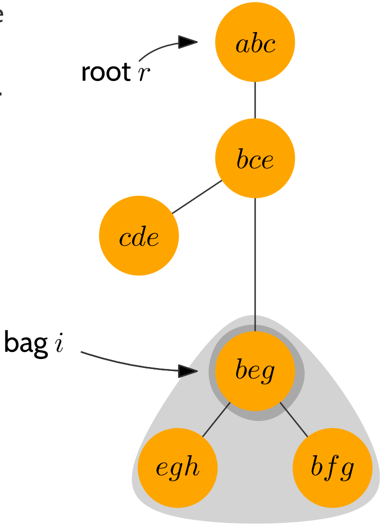
(i.e., every path between V_i and $V \setminus V_i$ goes through B_i).

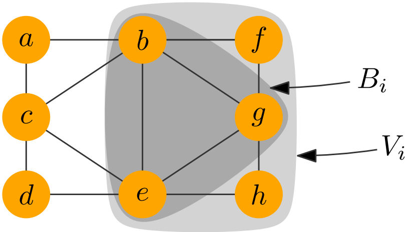




Dynamic Programming

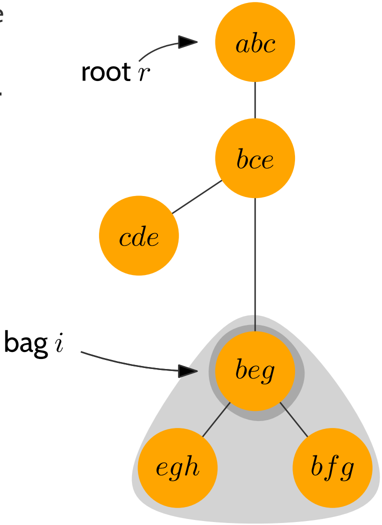
- For every set $S \subseteq B_i$, let $M_i(S)$ be the maximum size of an independent set in $G[V_i]$ that intersects B_i exactly in S.



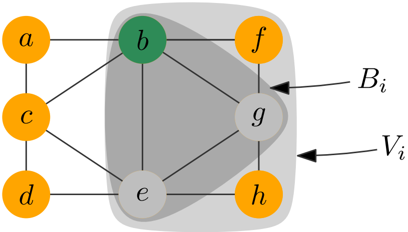


Dynamic Programming

- For every set $S \subseteq B_i$, let $M_i(S)$ be the maximum size of an independent set in $G[V_i]$ that intersects B_i exactly in S.



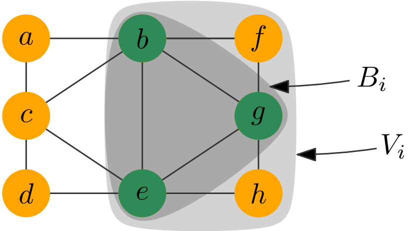
$M_i(\{b\}) = ?$

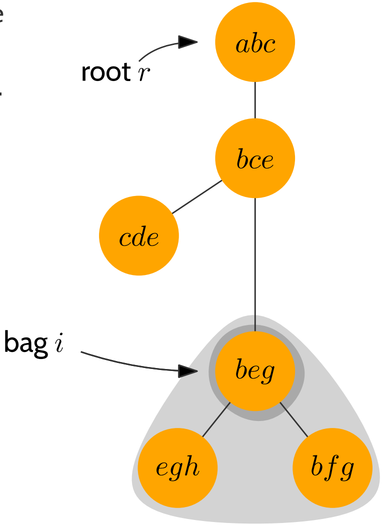


Dynamic Programming

- For every set $S \subseteq B_i$, let $M_i(S)$ be the maximum size of an independent set in $G[V_i]$ that intersects B_i exactly in S.

$$M_i(\{b, e, g\}) = ?$$

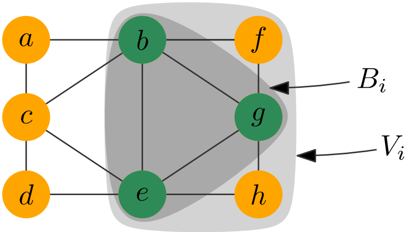


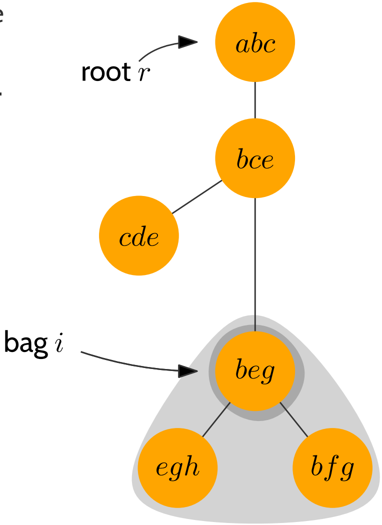


Dynamic Programming

- For every set $S \subseteq B_i$, let $M_i(S)$ be the maximum size of an independent set in $G[V_i]$ that intersects B_i exactly in S.
- The *table entries* of a bag i are the values $M_i(S)$ for all $S \subseteq B_i$.

$$M_i(\{b, e, g\}) = ?$$

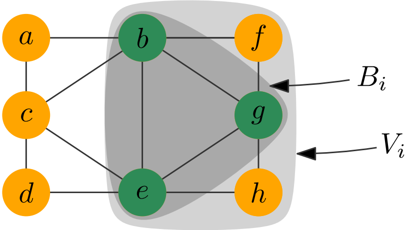


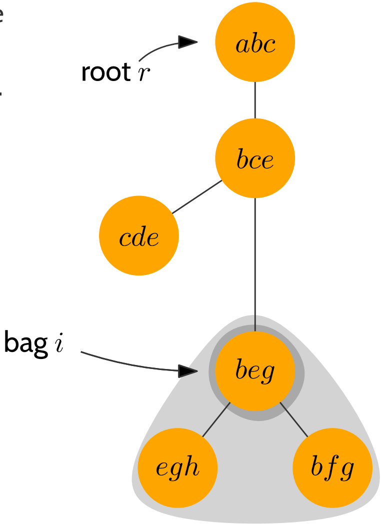


Dynamic Programming

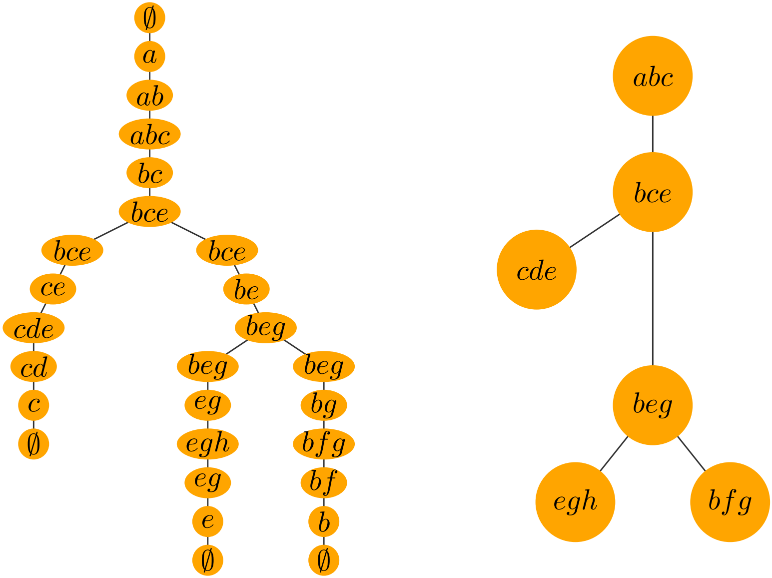
- For every set $S \subseteq B_i$, let $M_i(S)$ be the maximum size of an independent set in $G[V_i]$ that intersects B_i exactly in S.
- The *table entries* of a bag i are the values $M_i(S)$ for all $S \subseteq B_i$.
- We will compute all table entries inductively, starting at the leafs.

$$M_i(\{b, e, g\}) = ?$$

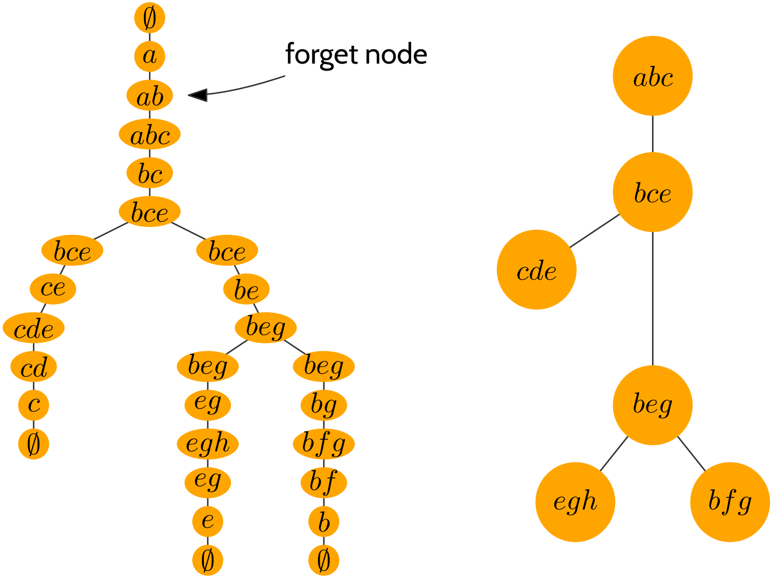




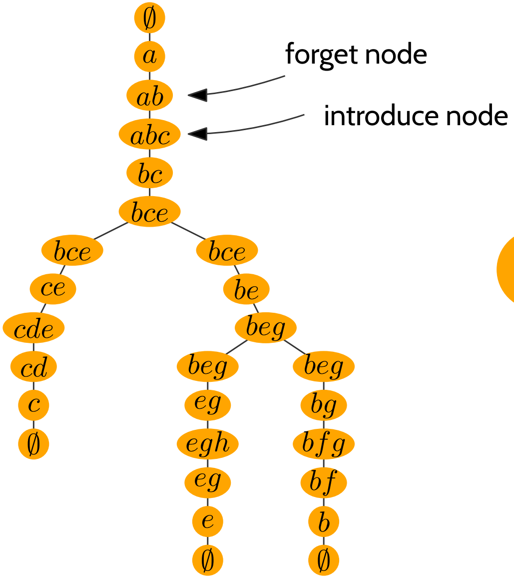
Which One is Nicer?

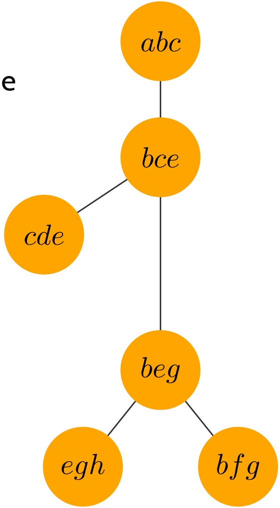


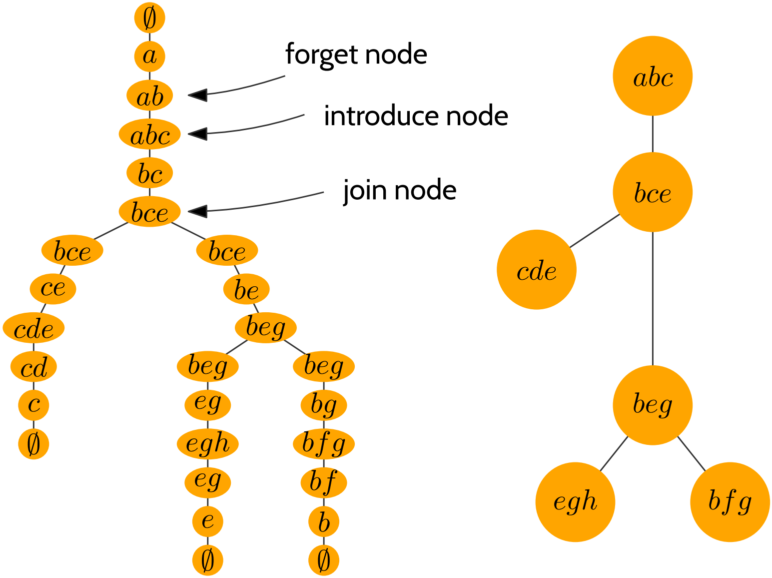
Which One is Nicer?



Which One is Nicer?

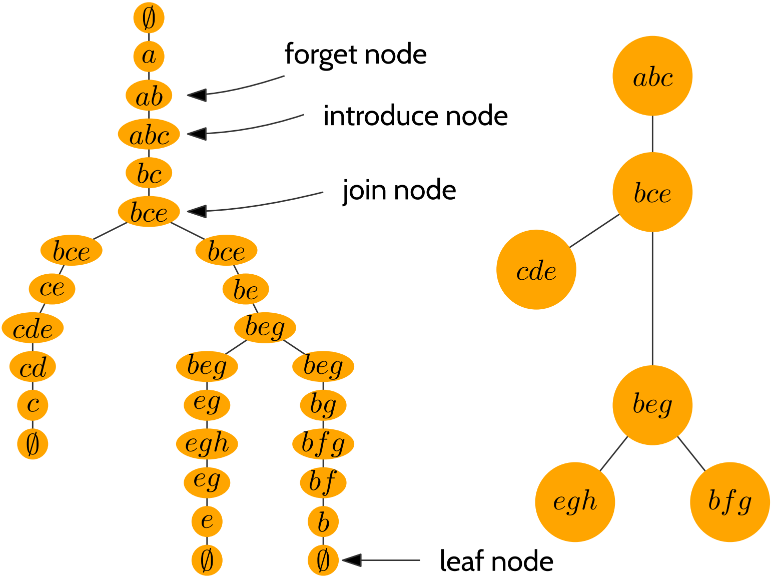




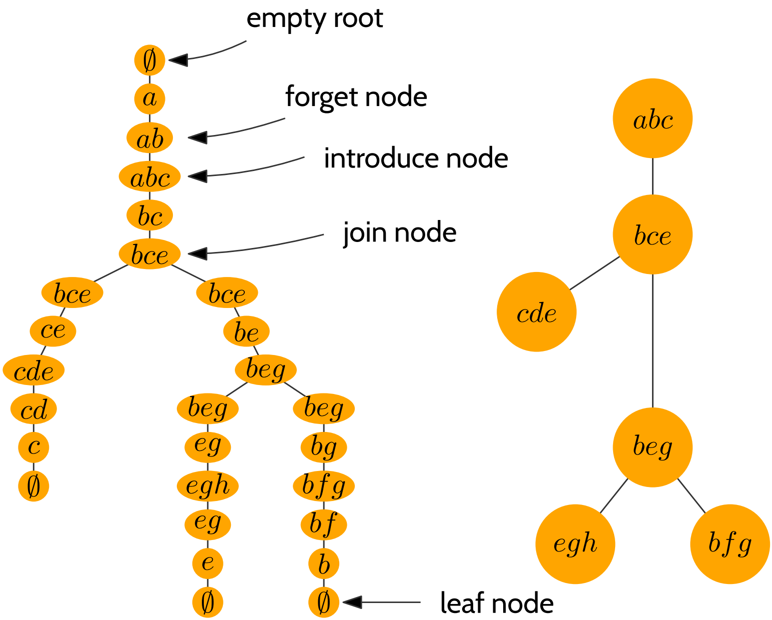


$\emptyset$
a
ab
abc
bc
bce
bce
ce
cde
cd
c
$\emptyset$
bce
be
beg
beg
eg
egh
eg
e
$\emptyset$
beg
bg
bfg
bf
b
$\emptyset$
forget node
introduce node
join node
abc
bce
cde
beg
egh
bfg

Which One is Nicer?



Which One is Nicer?



Nice Tree Decompositions

Nice tree decompositions consist of

- *Leaf nodes:* have no children and are empty
- *Introduce nodes:* have one child and contain exactly one vertex more than it
- *Forget nodes:* have one child and contain exactly one vertex less than it
- *Join nodes:* have exactly two identical children

There is an algorithm that will convert a tree decomposition of width w in time $O(nw^2)$ into a nice tree decomposition of width w with $O(nw)$ bags.

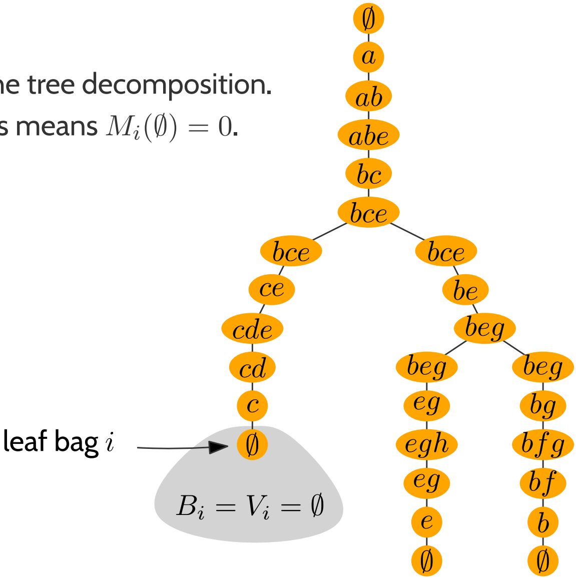
Computing Independent Set

- $M_i(S)$ is the maximum size of an independent set in $G[V_i]$ that intersects the bag vertices B_i exactly in S.
- The *table entries* of a bag i are the values $M_i(S)$ for all $S \subseteq B_i$. For every bag, there are $2^{|B_i|} \leq 2^w$ entries.
- We express the table entries of a bag in terms of the entries of its children.

Recurrence Relation

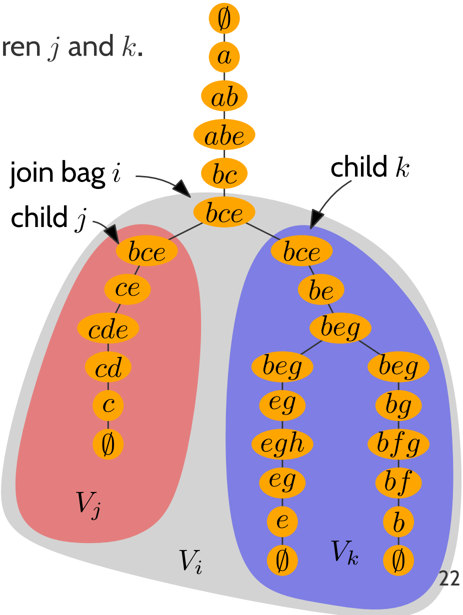
Let i be a leaf bag in the tree decomposition.

Then $B_i = V_i = \emptyset$. This means $M_i(\emptyset) = 0$.



Recurrence Relation

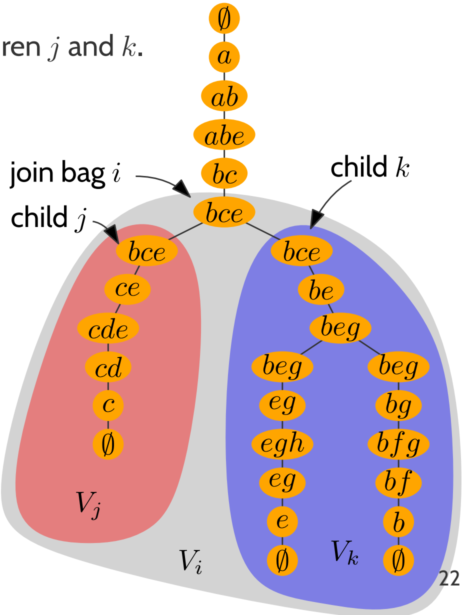
Assume i is a join node with children j and k.



Recurrence Relation

Assume i is a join node with children j and k.

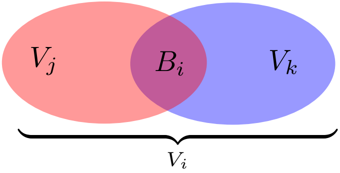
- Since the decomposition is nice, $B_i = B_j = B_k$

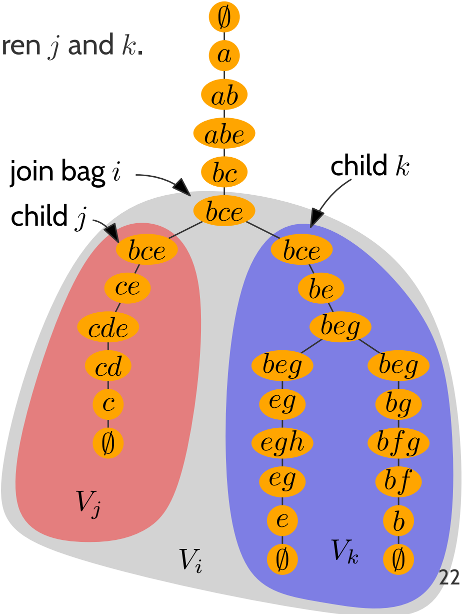


Recurrence Relation

Assume i is a join node with children j and k.

- Since the decomposition is nice, $B_i = B_j = B_k$
- Thus, $V_i = V_j \cup V_k$
- Also $V_j \cap V_k = B_i$ separates V_j and V_k

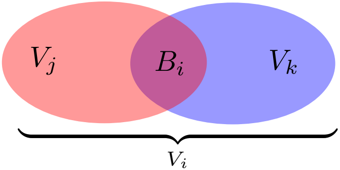


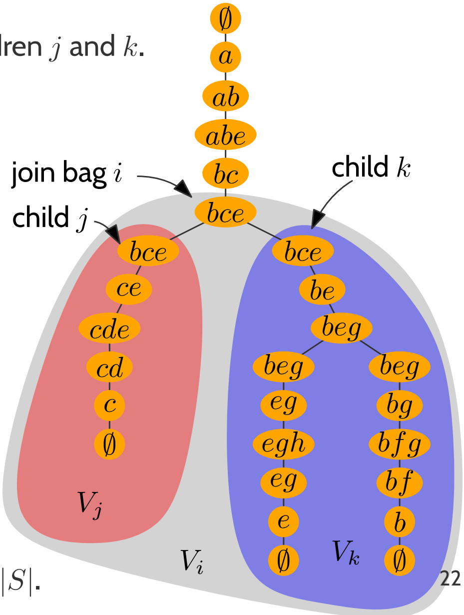


Recurrence Relation

Assume i is a join node with children j and k.

- Since the decomposition is nice, $B_i = B_j = B_k$
- Thus, $V_i = V_j \cup V_k$
- Also $V_j \cap V_k = B_i$ separates V_j and V_k

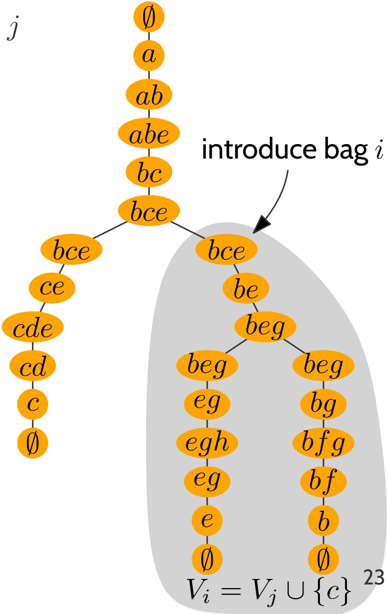




- Therefore
$$M_i(S) = M_j(S) + M_k(S) - |S|.$$

Recurrence Relation

Assume i is an introduce node with child j and $B_i = B_j \cup \{c\}$.

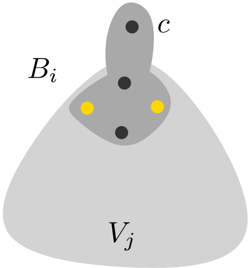


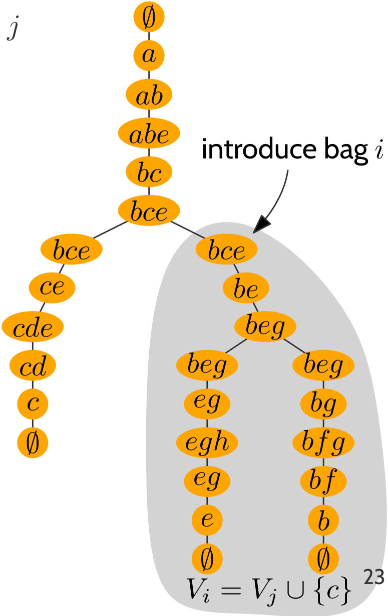
Recurrence Relation

Assume i is an introduce node with child j and $B_i = B_j \cup \{c\}$.

$M_i(S) =$

$\Bigg\{$



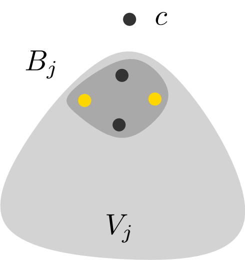


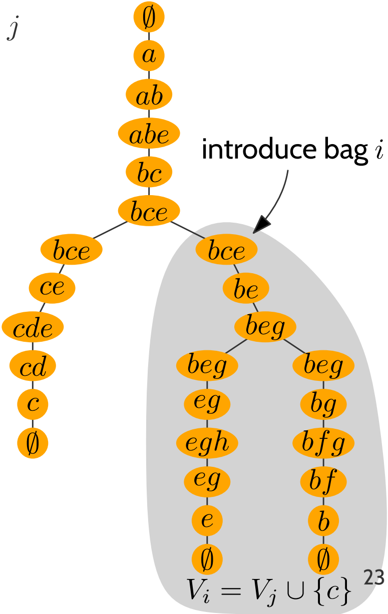
Recurrence Relation

Assume i is an introduce node with child j and $B_i = B_j \cup \{c\}$.

$$M_i(S) =$$

$$\begin{cases} M_j(S) & \text{if } c \notin S, \\ & \\ & \end{cases}$$

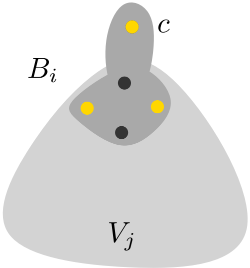


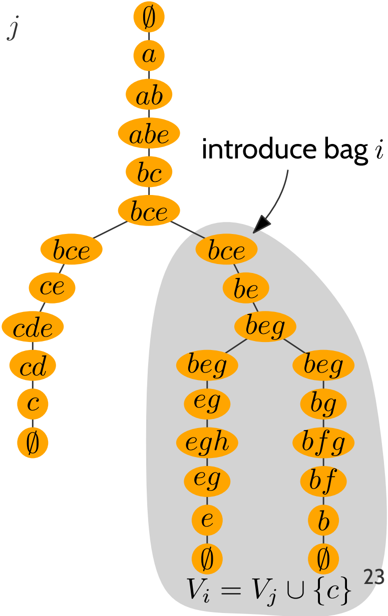


Recurrence Relation

Assume i is an introduce node with child j and $B_i = B_j \cup \{c\}$.

$$M_i(S) = \begin{cases} M_j(S) & \text{if } c \notin S, \\ & \\ & \end{cases}$$

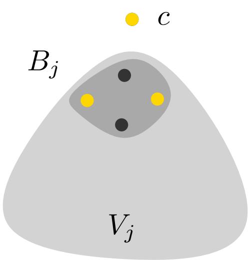


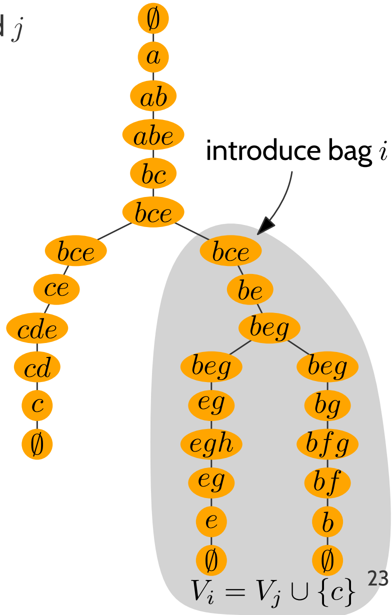


Recurrence Relation

Assume i is an introduce node with child j and $B_i = B_j \cup \{c\}$.

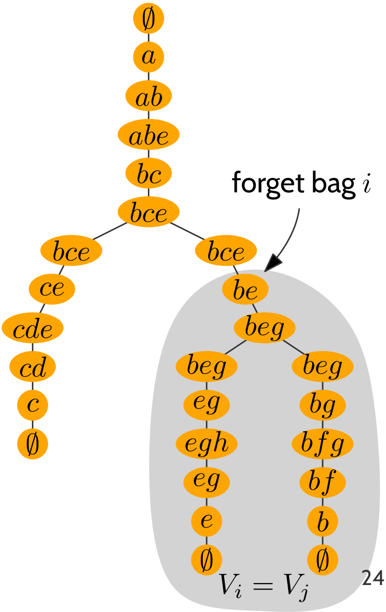
$$M_i(S) = \begin{cases} M_j(S) & \text{if } c \notin S, \\ M_j(S \setminus \{c\}) + 1 & \text{if } c \in S \text{ and } S \text{ is IS}, \\ 0 & \text{otherwise.} \end{cases}$$





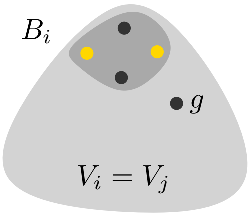
Recurrence Relation

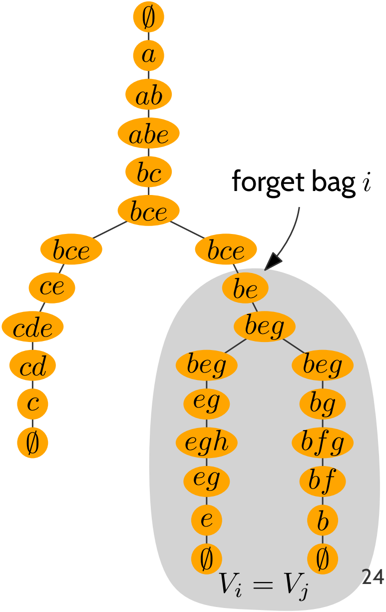
Assume i is a forget node with child j and $B_i = B_j \setminus \{g\}$.



Recurrence Relation

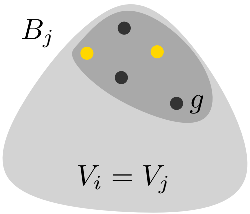
Assume i is a forget node with child j and $B_i = B_j \setminus \{g\}$.

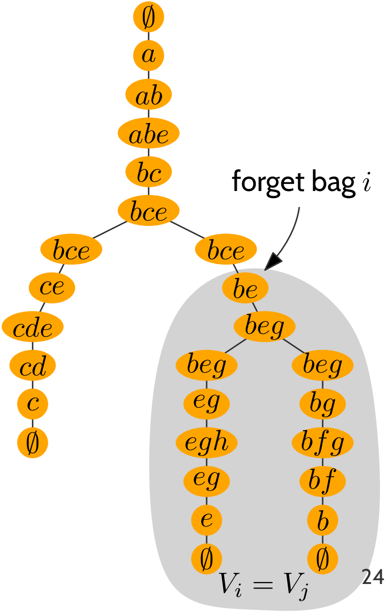




Recurrence Relation

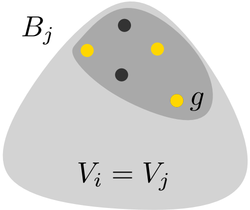
Assume i is a forget node with child j and $B_i = B_j \setminus \{g\}$.

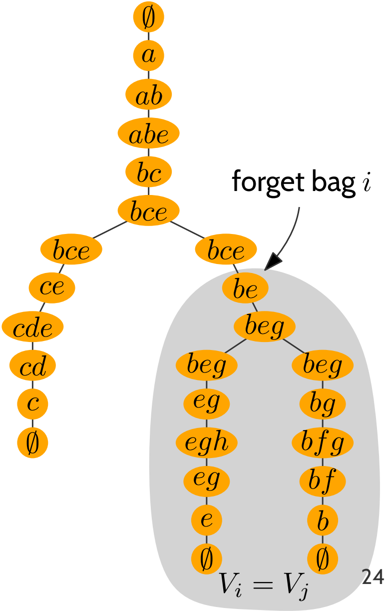




Recurrence Relation

Assume i is a forget node with child j and $B_i = B_j \setminus \{g\}$.

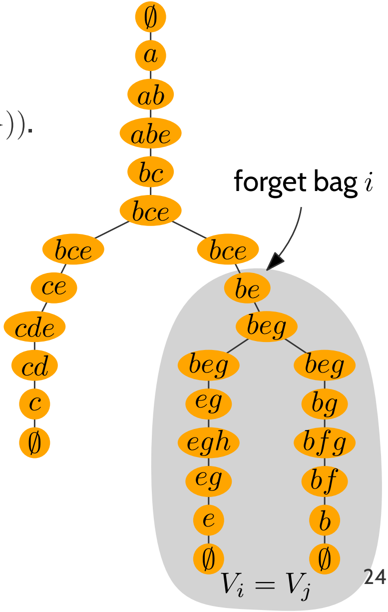


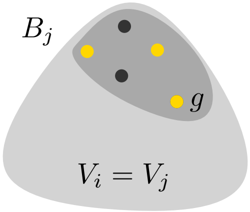


Recurrence Relation

Assume i is a forget node with child j
and $B_i = B_j \setminus \{g\}$.

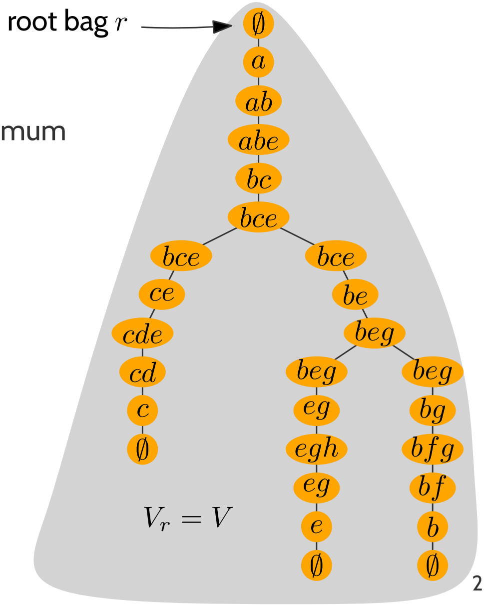
Then $M_i(S) = \max(M_j(S), M_j(S \cup \{g\}))$.





Obtaining the Answer

The answer is in the root bag r.
$M_r(\emptyset)$ equals the size of a maximum independent set in $G[V_r] = G$.



Theorem

Theorem (Independent Set on Treewidth)

Given a graph G and a tree decomposition of G of width w, one can compute (the size of) a maximum independent set in time $2^w w^{O(1)} n$.

Computing Tree Decompositions

Theorem (Korhonen 2021)

There is an algorithm that, given an n-vertex graph G and an integer w, runs in time $2^{O(w)}n$ and computes a tree decomposition of G of width at most $2w+1$ or concludes that the treewidth of G exceeds w.

Computing Tree Decompositions

Theorem (Korhonen 2021)

There is an algorithm that, given an n-vertex graph G and an integer w, runs in time $2^{O(w)}n$ and computes a tree decomposition of G of width at most $2w+1$ or concludes that the treewidth of G exceeds w.

We can find a good enough decomposition by trying increasing values of w. This yields:

Computing Tree Decompositions

Theorem (Korhonen 2021)

There is an algorithm that, given an n-vertex graph G and an integer w, runs in time $2^{O(w)}n$ and computes a tree decomposition of G of width at most $2w+1$ or concludes that the treewidth of G exceeds w.

We can find a good enough decomposition by trying increasing values of w. This yields:

Theorem

One can compute (the size of) a maximum independent set in time $2^{O(\mathsf{tw}(G))}n$.

Summary

- Treewidth is a powerful width parameter that describes tree-like graphs.

Summary

- Treewidth is a powerful width parameter that describes tree-like graphs.
- Independent Set can be solved in time $f(w)n$ on graphs with treewidth w.

Summary

- Treewidth is a powerful width parameter that describes tree-like graphs.
- Independent Set can be solved in time $f(w)n$ on graphs with treewidth w.
- The same dynamic programming technique leads to algorithms for many other problems.

Summary

- Treewidth is a powerful width parameter that describes tree-like graphs.
- Independent Set can be solved in time $f(w)n$ on graphs with treewidth w.
- The same dynamic programming technique leads to algorithms for many other problems.
- $\Rightarrow$ Graphs with small treewidth are "algorithmically tractable".

Not Everything is Easy on Bounded Treewidth

Nishizeki, Vygen, Zhou 2001

Finding edge-disjoint paths between source-sink pairs is hard on graphs with treewidth two.

Outline

- There are many more problems one can solve on graphs with small treewidth.
  - coloring
  - independent set
  - clique
  - dominating set
  - feedback vertex set
  - hamilton path
  - …
- We don’t want to write down a separate dynamic programming algorithm for each of them. Instead, we present a meta-algorithm that solves all of them.
- To do so, we first need some background in logic.

Logic vs Graph-Theory

Graph-theorists and logicans use different languages:

- graph $\leftrightarrow$ structure

Logic vs Graph-Theory

Graph-theorists and logicans use different languages:

- graph $\leftrightarrow$ structure
- induced subgraph $\leftrightarrow$ substructure

Logic vs Graph-Theory

Graph-theorists and logicans use different languages:

- graph $\leftrightarrow$ structure
- induced subgraph $\leftrightarrow$ substructure
- subgraph $\leftrightarrow$ weak substructure

Logic vs Graph-Theory

Graph-theorists and logicans use different languages:

- graph $\leftrightarrow$ structure
- induced subgraph $\leftrightarrow$ substructure
- subgraph $\leftrightarrow$ weak substructure
- all vertices (or edges) of a graph $\leftrightarrow$ universe of the structure

Logic vs Graph-Theory

Graph-theorists and logicans use different languages:

- graph $\leftrightarrow$ structure
- induced subgraph $\leftrightarrow$ substructure
- subgraph $\leftrightarrow$ weak substructure
- all vertices (or edges) of a graph $\leftrightarrow$ universe of the structure
- vertex (or edge) $\leftrightarrow$ element

Logic vs Graph-Theory

Graph-theorists and logicans use different languages:

- graph $\leftrightarrow$ structure
- induced subgraph $\leftrightarrow$ substructure
- subgraph $\leftrightarrow$ weak substructure
- all vertices (or edges) of a graph $\leftrightarrow$ universe of the structure
- vertex (or edge) $\leftrightarrow$ element
- adjacency $\leftrightarrow$ binary relation

Logic vs Graph-Theory

Graph-theorists and logicans use different languages:

- graph $\leftrightarrow$ structure
- induced subgraph $\leftrightarrow$ substructure
- subgraph $\leftrightarrow$ weak substructure
- all vertices (or edges) of a graph $\leftrightarrow$ universe of the structure
- vertex (or edge) $\leftrightarrow$ element
- adjacency $\leftrightarrow$ binary relation
- colors $\leftrightarrow$ unary relation

Logic vs Graph-Theory

Graph-theorists and logicans use different languages:

- graph $\leftrightarrow$ structure
- induced subgraph $\leftrightarrow$ substructure
- subgraph $\leftrightarrow$ weak substructure
- all vertices (or edges) of a graph $\leftrightarrow$ universe of the structure
- vertex (or edge) $\leftrightarrow$ element
- adjacency $\leftrightarrow$ binary relation
- colors $\leftrightarrow$ unary relation
- ...

If we want to use tools from logic (and don't want to embarras us in front of logicans) we will have to learn some of their language and formalism.

Graphs as Structures

Logic works on *structures*. We can easily see graphs as structures.

Graphs as Structures

Logic works on *structures*. We can easily see graphs as structures.

- Each structure has a *signature* τ: a set of relational symbols with given arities.

Graphs as Structures

Logic works on *structures*. We can easily see graphs as structures.

- Each structure has a *signature* τ: a set of relational symbols with given arities.
- We interpret *colored undirected graphs* as τ-structures with $\tau = \{\sim, c_1, c_2, \dots\}$ where

Graphs as Structures

Logic works on *structures*. We can easily see graphs as structures.

- Each structure has a *signature* τ: a set of relational symbols with given arities.
- We interpret *colored undirected graphs* as τ-structures with $\tau = \{\sim, c_1, c_2, \ldots\}$ where
  - the universe are the vertices

Graphs as Structures

Logic works on *structures*. We can easily see graphs as structures.

- Each structure has a *signature* τ: a set of relational symbols with given arities.
- We interpret *colored undirected graphs* as τ-structures with $\tau = \{\sim, c_1, c_2, \ldots\}$ where
  - the universe are the vertices
  - $\sim$ denotes the binary adjacency relation between vertices

Graphs as Structures

Logic works on *structures*. We can easily see graphs as structures.

- Each structure has a *signature* τ: a set of relational symbols with given arities.
- We interpret *colored undirected graphs* as τ-structures with $\tau = \{\sim, c_1, c_2, \dots\}$ where
  - the universe are the vertices
  - $\sim$ denotes the binary adjacency relation between vertices
  - c_i denotes the unary relation “the vertex is colored with color i”

Graphs as Structures

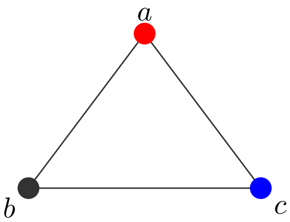
Logic works on *structures*. We can easily see graphs as structures.

- Each structure has a *signature* τ: a set of relational symbols with given arities.
- We interpret *colored undirected graphs* as τ-structures with $\tau = \{\sim, c_1, c_2, \dots\}$ where
  - the universe are the vertices
  - $\sim$ denotes the binary adjacency relation between vertices
  - c_i denotes the unary relation "the vertex is colored with color i"
- It is sometimes convenient to use other signatures
  - $\tau = \{\rightsquigarrow\}$ for directed graphs
  - $\tau = \{\sim, U, V\}$ for directed bipartite graphs
  - ...

Example

This graph is a structure G with

- universe $V = \{a, b, c\}$
- symmetrical binary relation
  $\sim := \{(a,b),(b,a),(b,c),(c,b)(a,c),(c,a)\}$
- unary relations $c_1 := \{a\}$, $c_2 := \{c\}$



Monadic Second-Order Logic (Syntax)

For a given signature τ, monadic second-order logic has ...

- element-variables ($x, y, z, \dots$) and set-variables ($X, Y, Z, \dots$)
- relations $=$ (equality) and $x \in X$ (membership), as well as the relations from τ.
- quantifiers $\exists$ and $\forall$, as well as operators $\wedge$, $\vee$ and $\neg$

Monadic Second-Order Logic (Syntax)

For a given signature τ, monadic second-order logic has …

- element-variables ($x, y, z, \dots$) and set-variables ($X, Y, Z, \dots$)
- relations $=$ (equality) and $x \in X$ (membership), as well as the relations from τ.
- quantifiers $\exists$ and $\forall$, as well as operators $\wedge$, $\vee$ and $\neg$

We mostly work on colored undirected graphs with $\tau = \{\sim, c_1, c_2, \dots\}$. Here, we call the logic MSO_1.

Monadic Second-Order Logic (Syntax)

For a given signature τ, monadic second-order logic has …

- element-variables ($x, y, z, \ldots$) and set-variables ($X, Y, Z, \ldots$)
- relations $=$ (equality) and $x \in X$ (membership), as well as the relations from τ.
- quantifiers $\exists$ and $\forall$, as well as operators $\wedge$, $\vee$ and $\neg$

We mostly work on colored undirected graphs with $\tau = \{\sim, c_1, c_2, \ldots\}$. Here, we call the logic MSO_1.

Instead of prefix notation ($\sim(x, y)$) we use infix notation ($x{\sim}y$) when convenient and add parentheses when it avoids confusion.

Monadic Second-Order Logic (Semantics)

What do these formulas mean?

$$\neg\exists r\ \exists x\ \exists g\ \big(\textsf{red}(r) \wedge \textsf{green}(g) \wedge r{\sim}x \wedge x{\sim}g\big)$$

Monadic Second-Order Logic (Semantics)

What do these formulas mean?

$$\neg\exists r\, \exists x\, \exists g\, \big(\mathsf{red}(r) \wedge \mathsf{green}(g) \wedge r{\sim}x \wedge x{\sim}g\big)$$

$$\varphi(X) \equiv \forall x\big(x \in X \vee \exists y\, y \in X \wedge x{\sim}y\big)$$

Some Observations

You do not need disjunctions ($\vee$).

$$\varphi \vee \psi \equiv \neg(\neg\psi \wedge \neg\varphi)$$

Some Observations

You do not need disjunctions ($\vee$).

$$\varphi \vee \psi \equiv \neg(\neg\psi \wedge \neg\varphi)$$

You do not need universal quantifiers.

$$\forall x\, \varphi \equiv \neg\exists x\, \neg\varphi$$

Some Observations

You do not need disjunctions ($\vee$).

$$\varphi \vee \psi \equiv \neg(\neg\psi \wedge \neg\varphi)$$

You do not need universal quantifiers.

$$\forall x\, \varphi \equiv \neg\exists x\, \neg\varphi$$

You can assume that all quantifier are in the beginning (Prenex normal form).

$$\varphi \wedge \exists x\, \psi \equiv \exists x\, \varphi \wedge \psi$$

Models

- If φ is a sentence (a formula without free variables), we write $G \models \varphi$ to indicate that φ holds on G (i.e., G is a model of φ).

Models

- If φ is a sentence (a formula without free variables), we write $G \models \varphi$ to indicate that φ holds on G (i.e., G is a model of φ).

- We say a graph property/problem is *expressible* in a logic if there exists a sentence φ such that for every graph G holds $G \models \varphi$ iff G is a yes-instance.

Examples

Can we express these properties in MSO_1?

- "G has at least 2 vertices"

Examples

Can we express these properties in MSO$_1$?

- "G has at least 2 vertices"

$$\exists x \exists y x \neq y$$

Examples

Can we express these properties in MSO_1?

- "G has at least 2 vertices"

$$\exists x \exists y x \neq y$$

- "G is connected"

Examples

Can we express these properties in MSO$_1$?

- "G has at least 2 vertices"

$$\exists x \exists y x \neq y$$

- "G is connected"

$$\forall X \forall Y \Big((\forall z z \in X \vee z \in Y) \to \big(\exists x \exists y (x \in X \wedge y \in Y \wedge x \sim y) \big) \Big)$$

Examples

Can we express these properties in MSO$_1$?

- "G has a proper 3 coloring"

Examples

Can we express these properties in MSO_1?

- "G has a proper 3 coloring"

$$\varphi \equiv \exists R \exists G \exists B$$

$$\Big(\forall x x \in R \vee x \in G \vee x \in B\Big)$$

$$\wedge \Big(\forall x \neg(x \in R \wedge x \in G) \wedge \neg(x \in G \wedge x \in B) \wedge \neg(x \in R \wedge x \in B)\Big)$$

$$\wedge \Big(\forall x \forall y \big((x \in R \wedge y \in R) \vee (x \in G \wedge y \in G) \vee (x \in B \wedge y \in B)\big)$$

$$\rightarrow \neg x \sim y\Big)$$