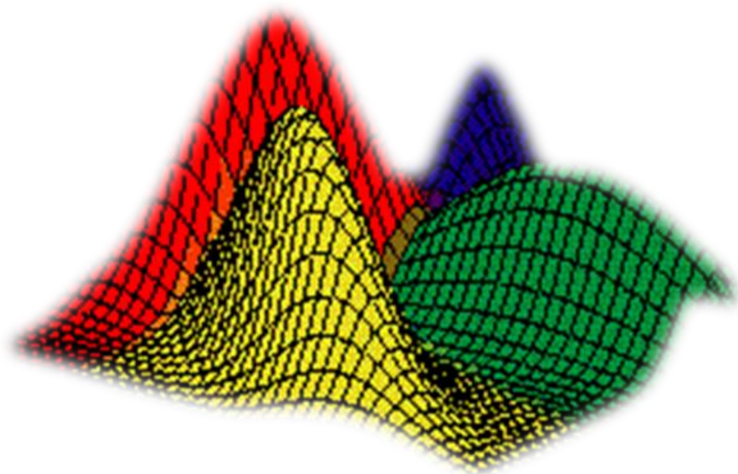
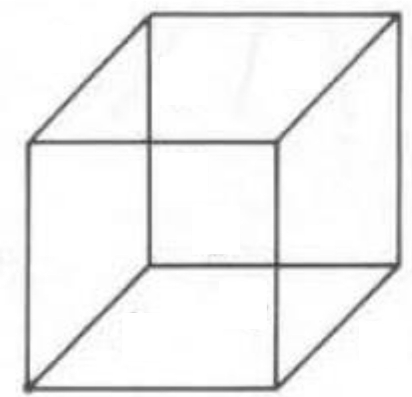


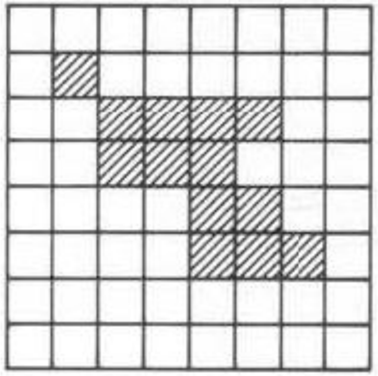
CSE 473: Pattern Recognition

Syntactic Pattern Recognition

Capability of String Grammar

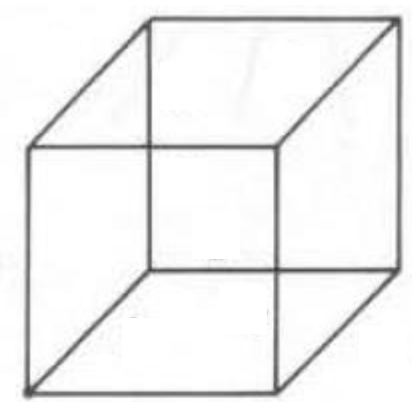
- able to represent 1D pattern
- difficult to classify patterns like these:

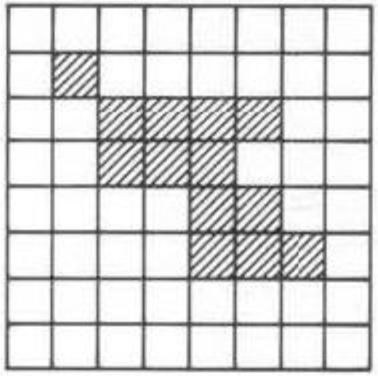




Capability of String Grammar

- These structure contains
  - information in both direction
  - hierarchical structure

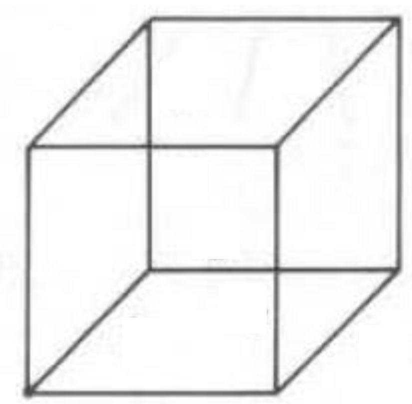


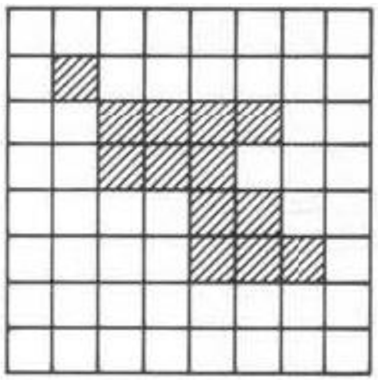


Capability of String Grammar

- These structure contains
  - information in both direction
  - hierarchical structure

Solution: Higher Order Grammar

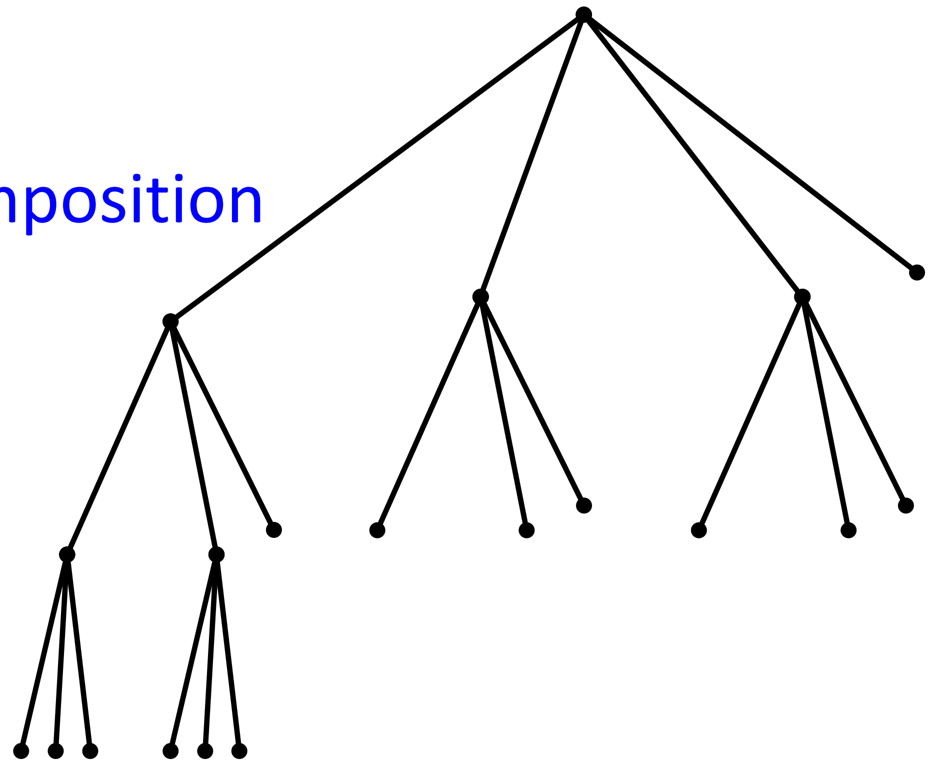




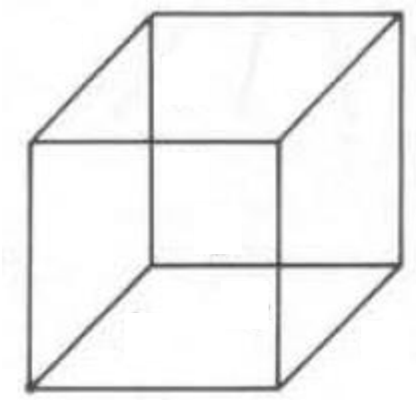
Higher Order Grammar: Tree Grammar

- Information is stored in
  - *nodes*: as primitives or sub-structures
  - *edges*: relation between primitives or sub-structures

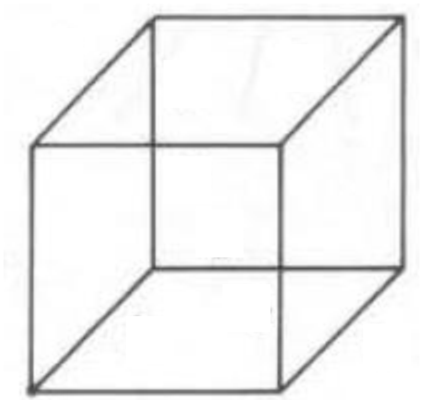
- Allows hierarchical decomposition of a complex structure

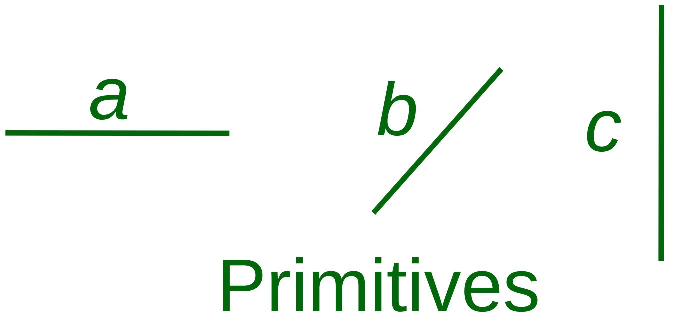


Representation using Tree (1)

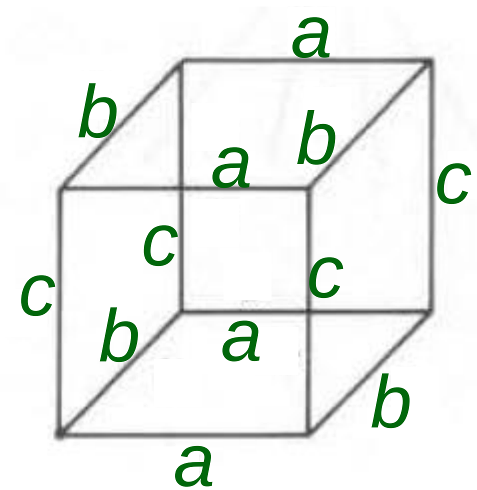


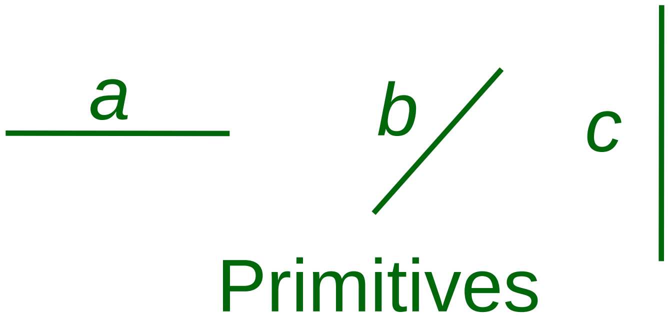
Representation using Tree (1)



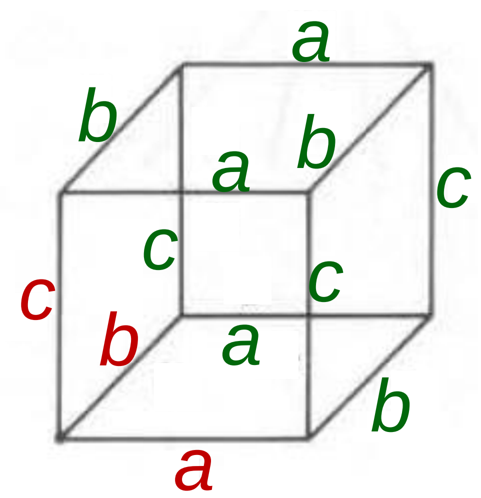


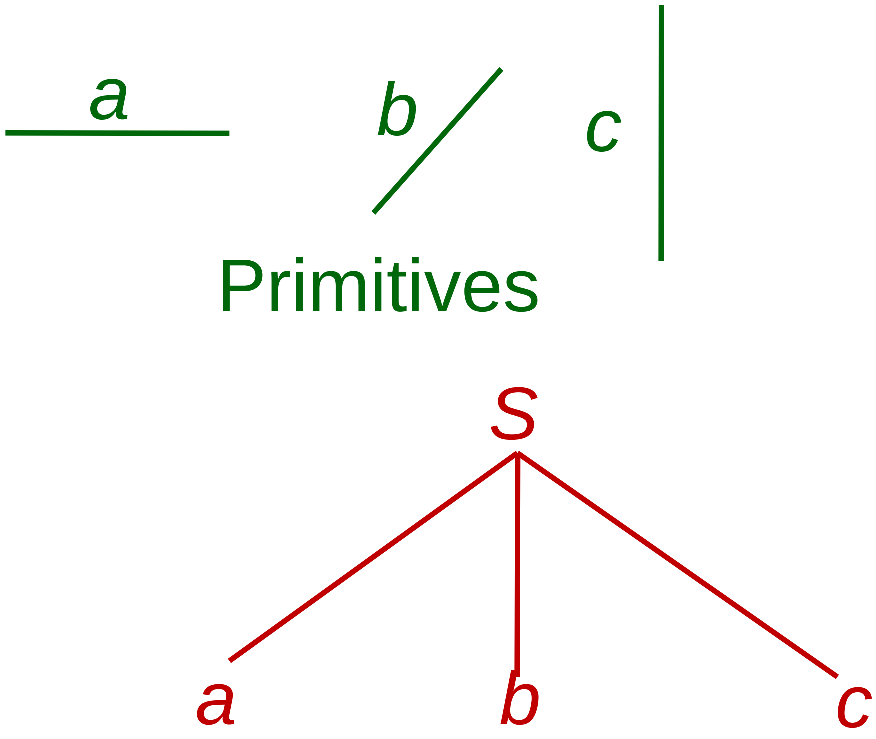
Representation using Tree (1)



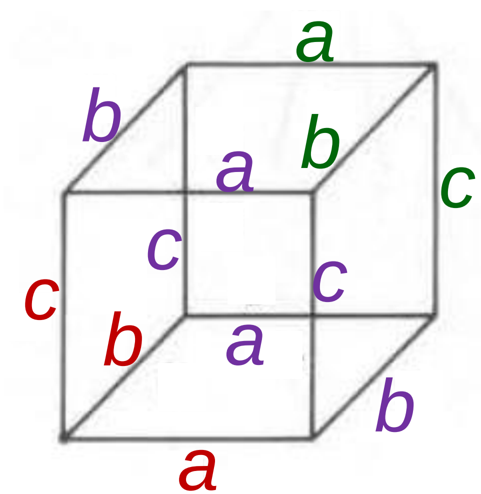


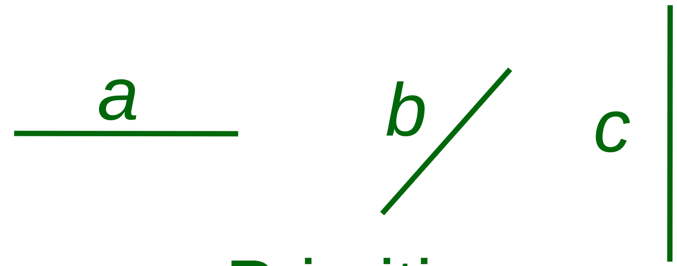
Representation using Tree (1)



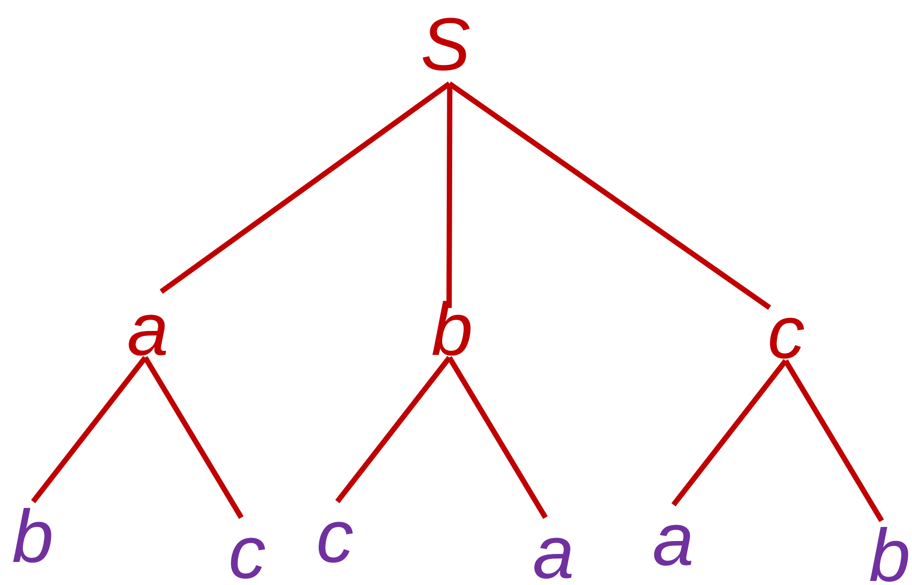


Representation using Tree (1)

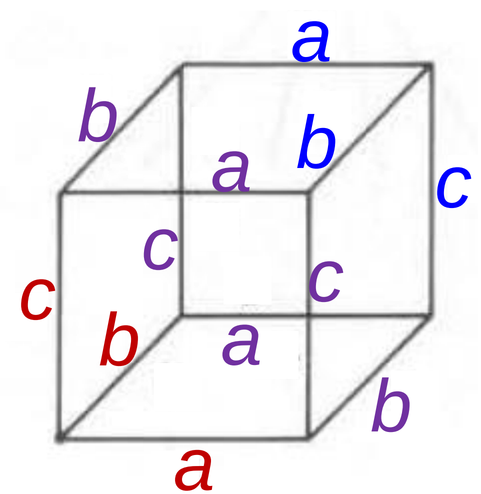


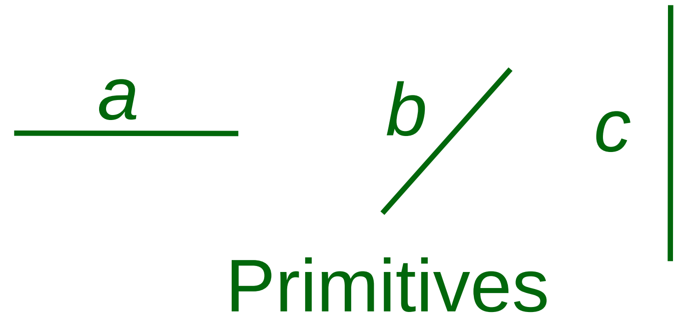


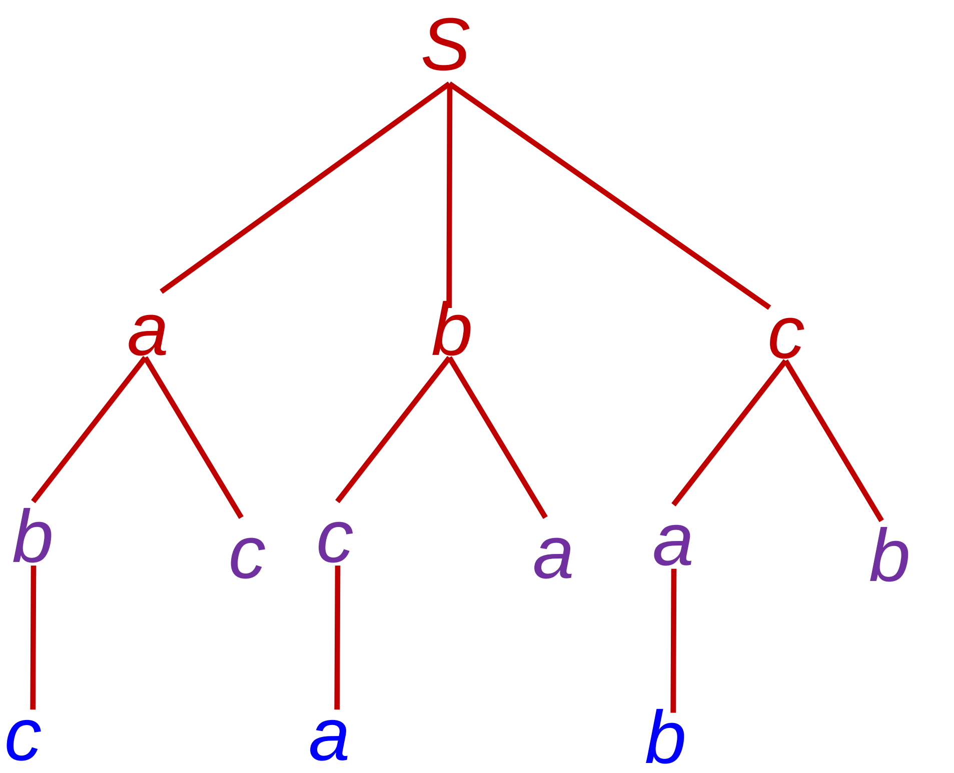
Primitives



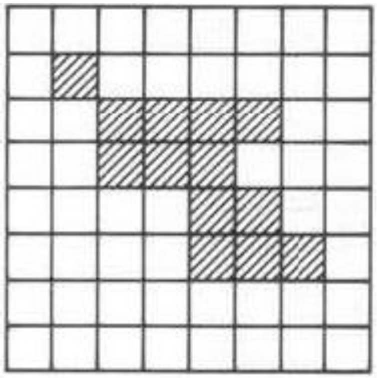
Representation using Tree (1)



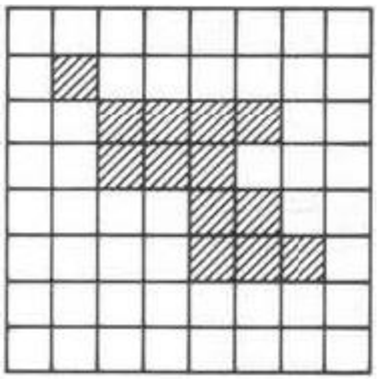


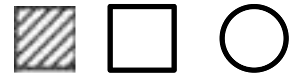


Representation using Tree (2)



Representation using Tree (2)





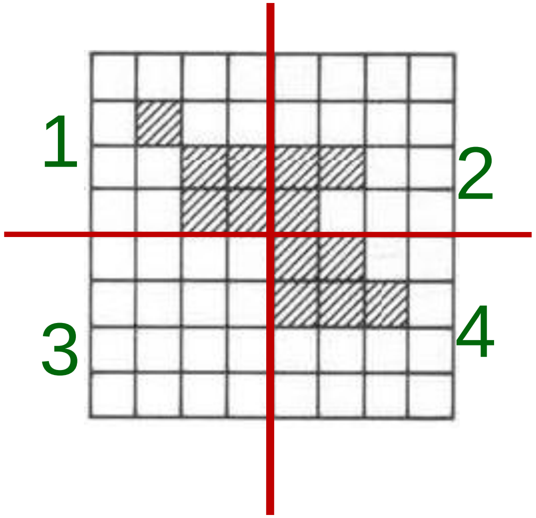
Primitives

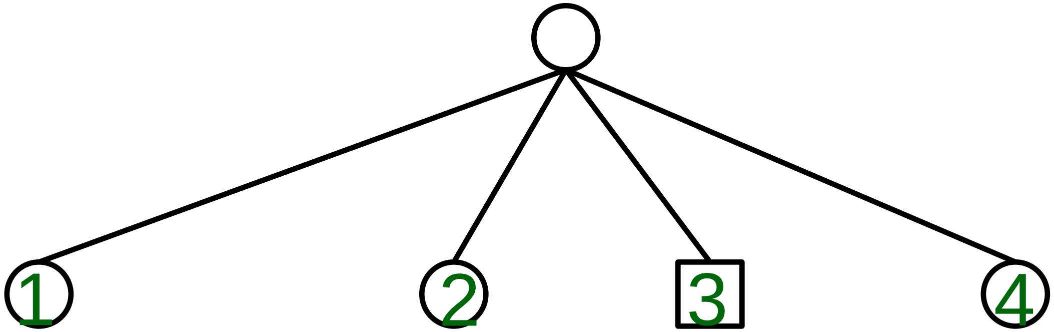
Black region

White region

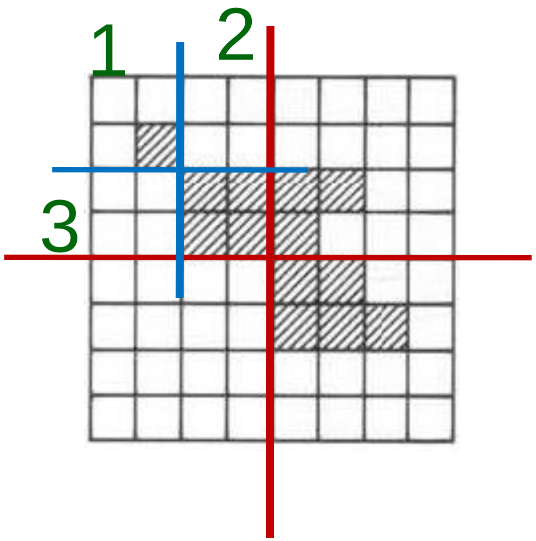
Gray region

Representation using Tree (2)

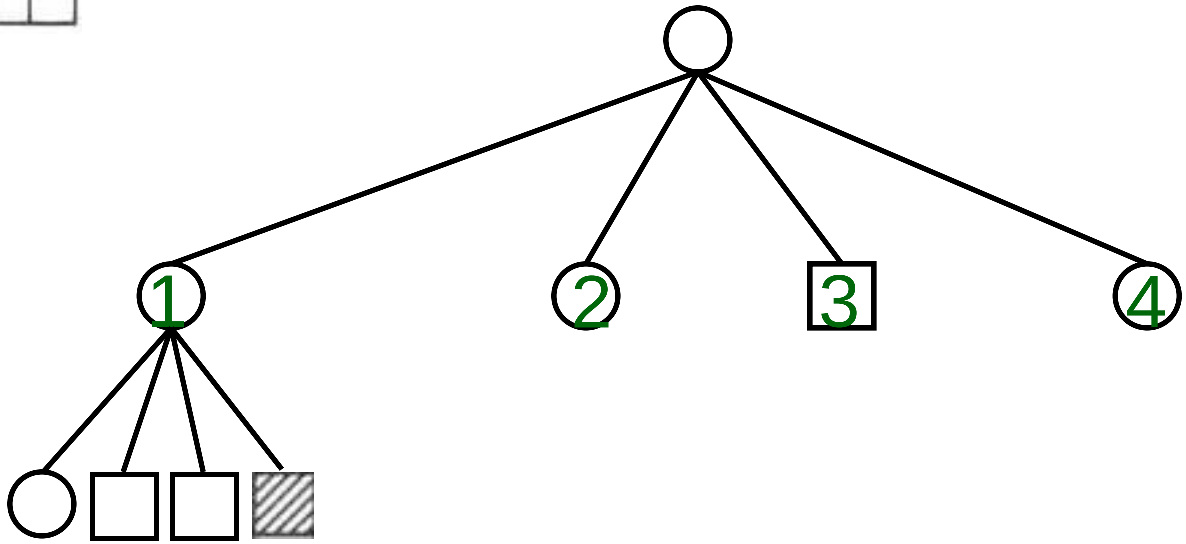




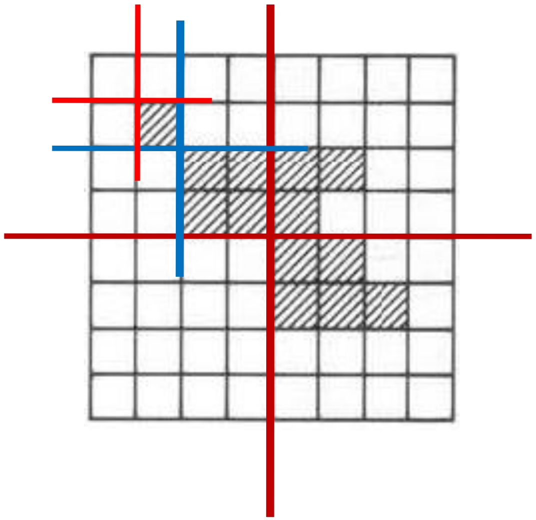
Representation using Tree (2)

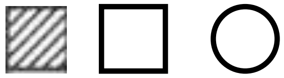


Primitives

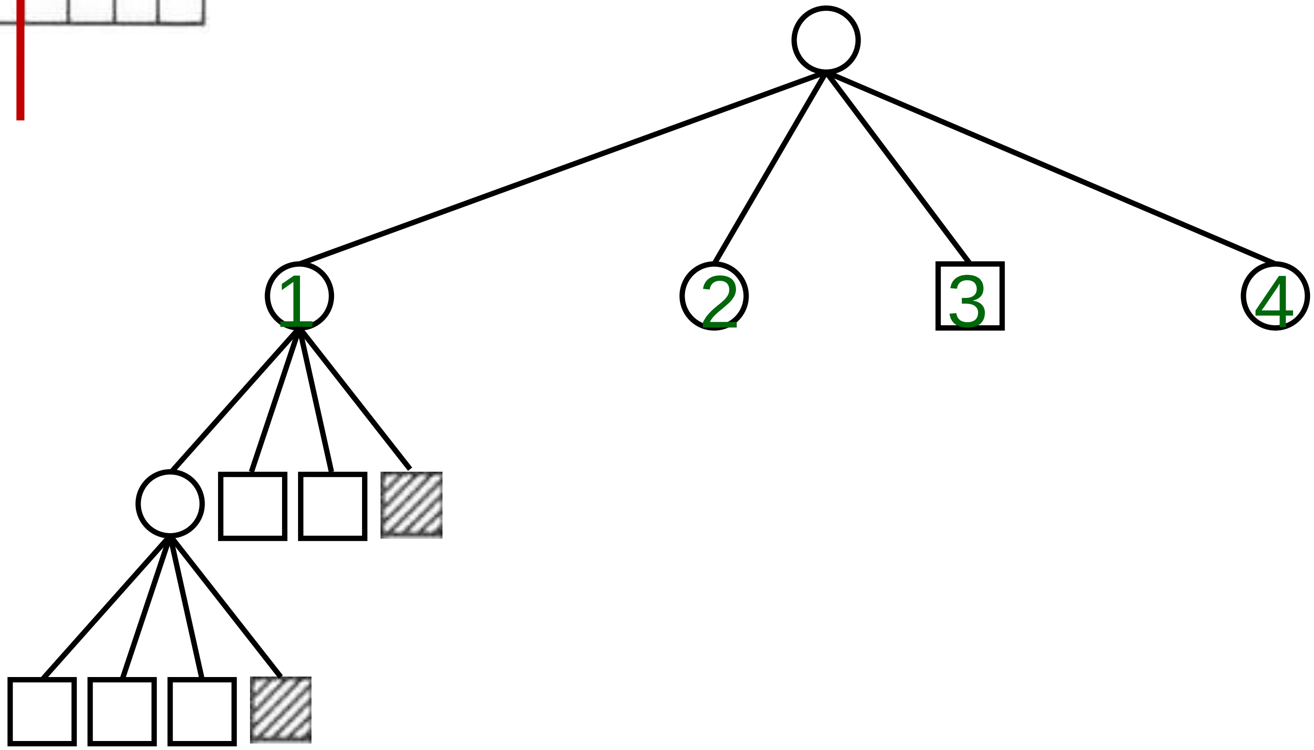


Representation using Tree (2)

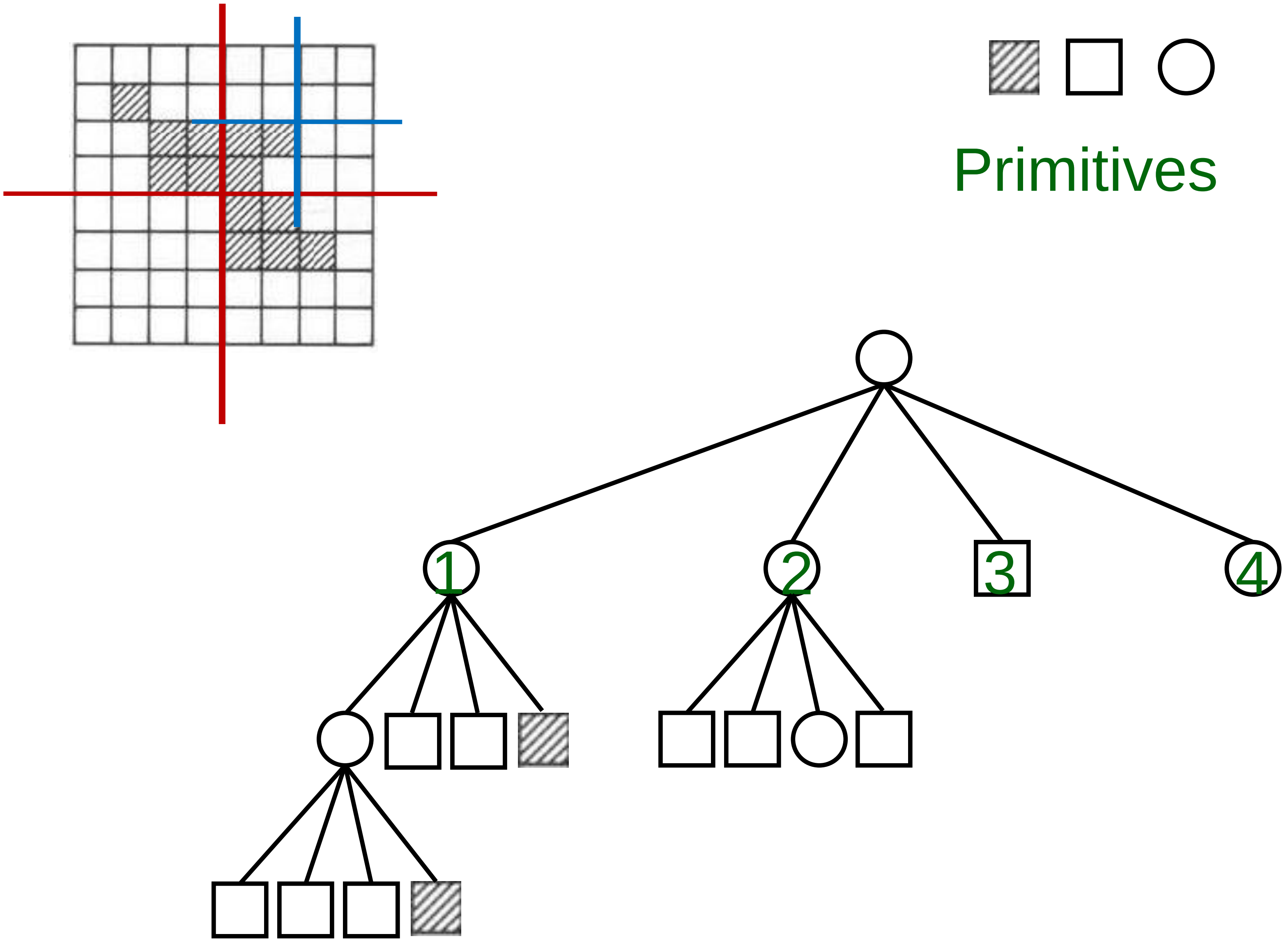




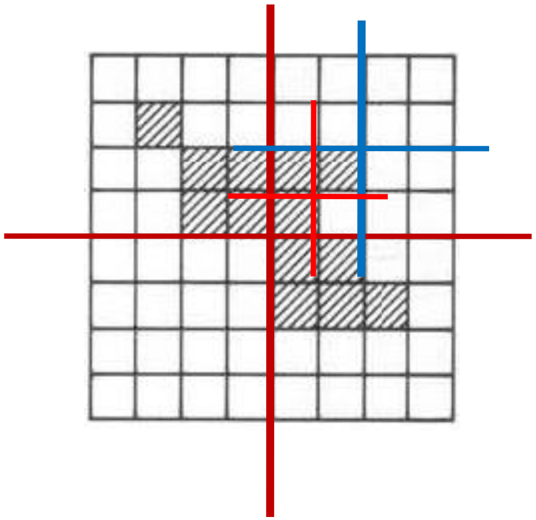
Primitives

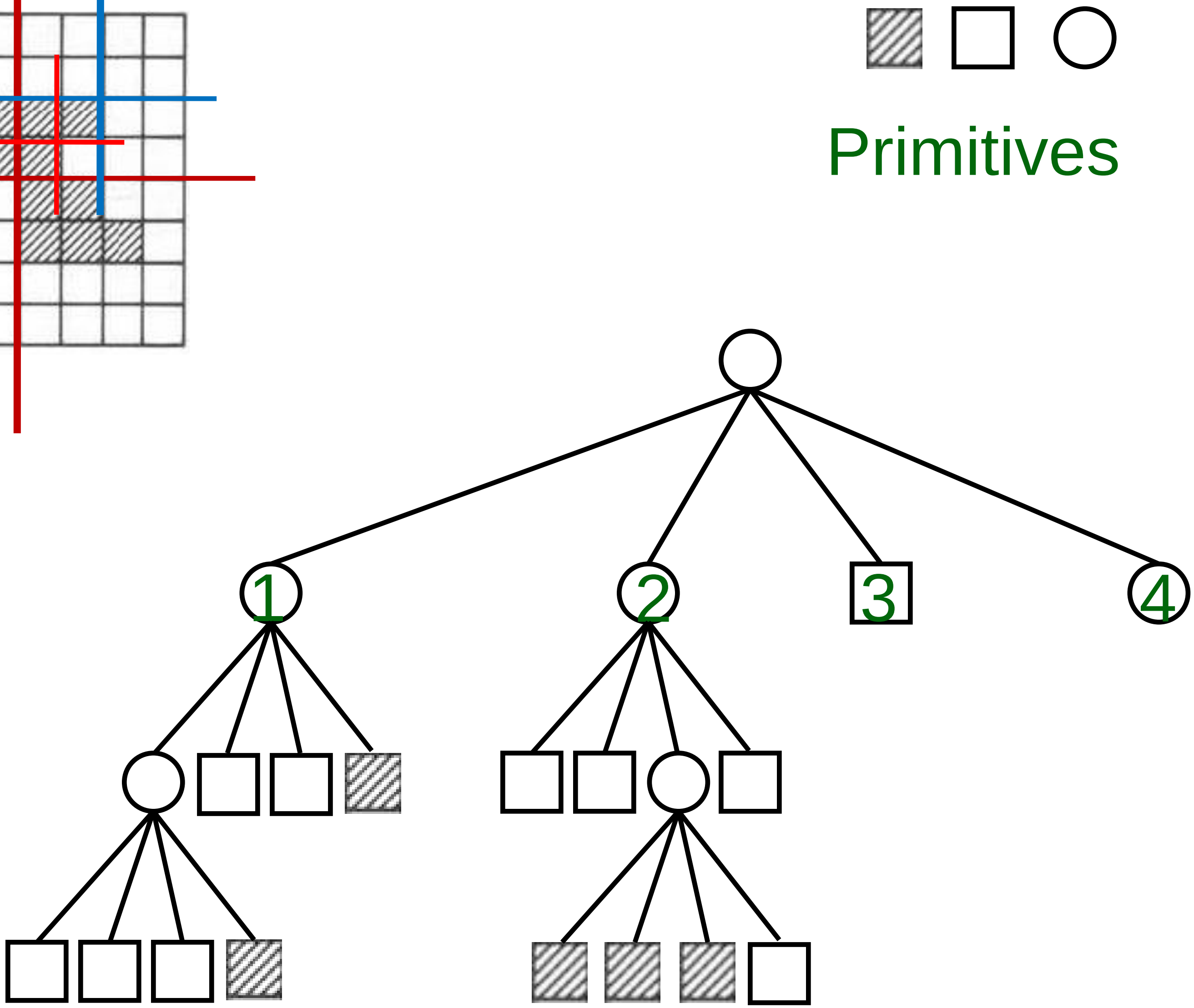


Representation using Tree (2)

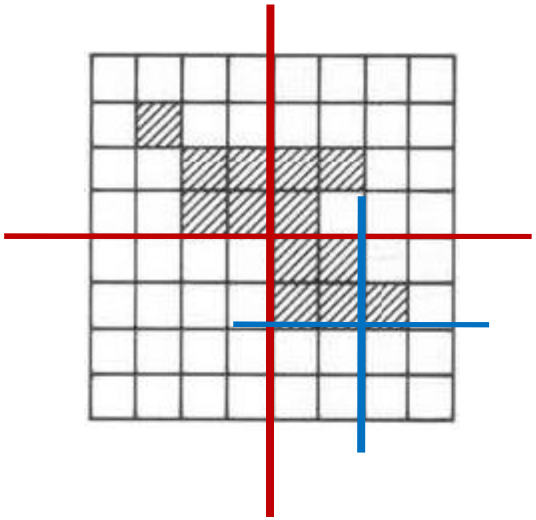


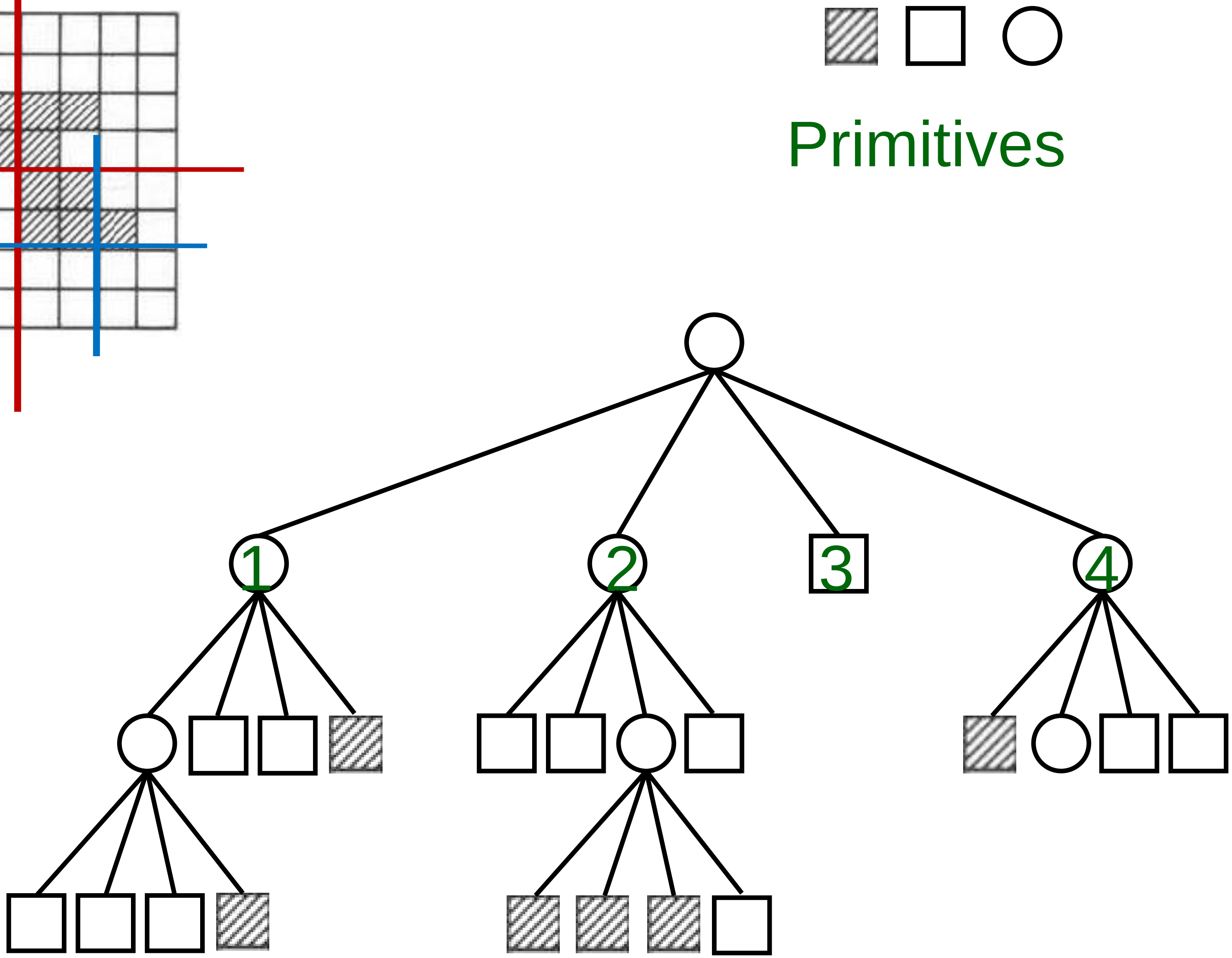
Representation using Tree (2)



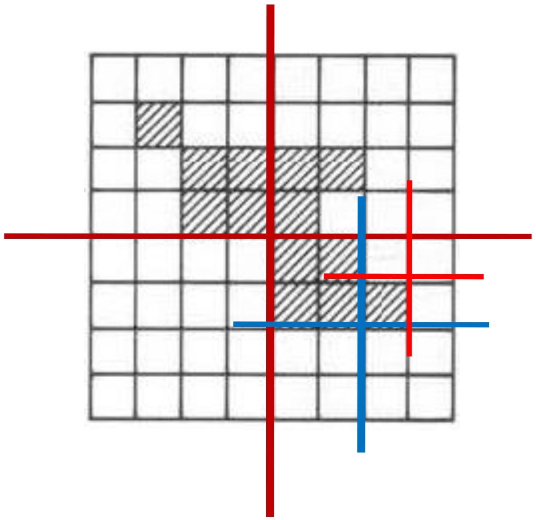


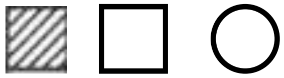
Representation using Tree (2)



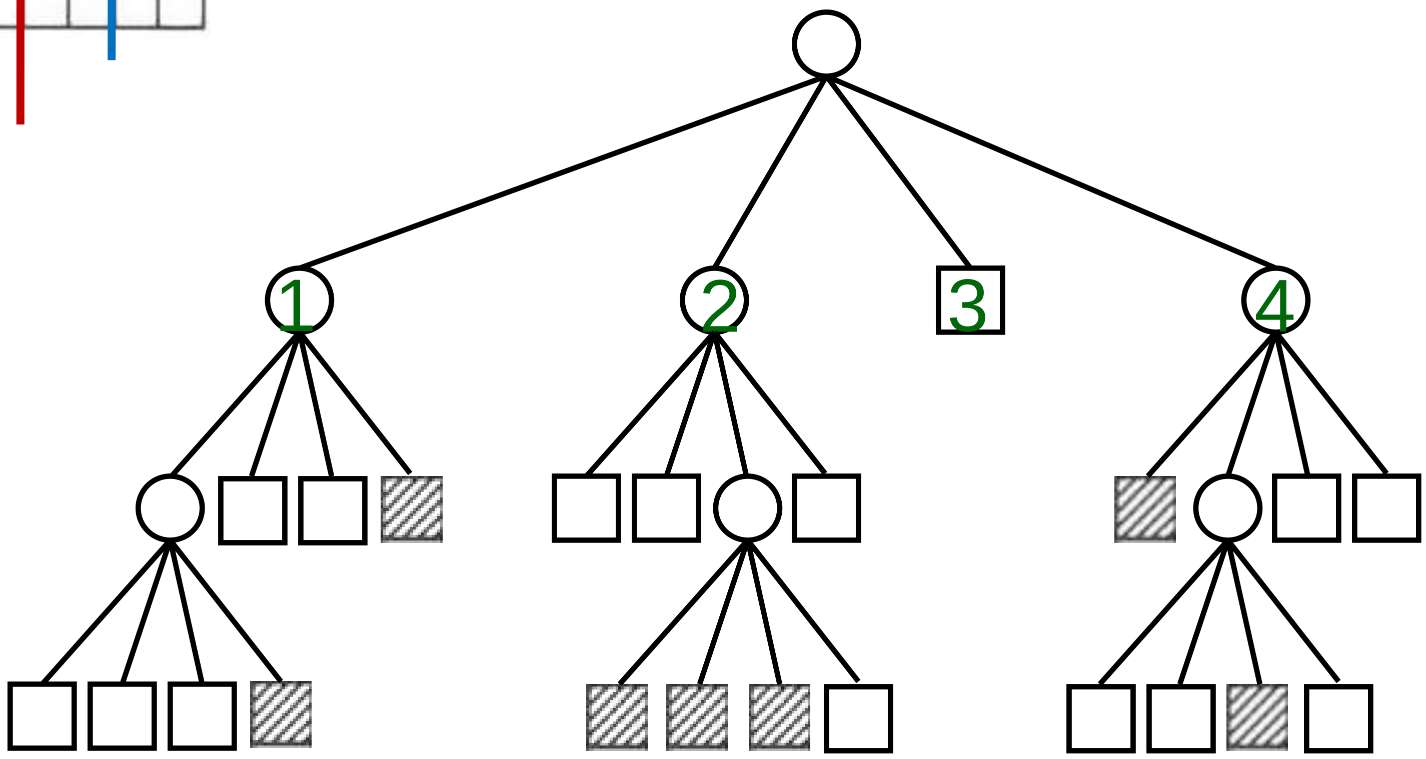


Representation using Tree (2)

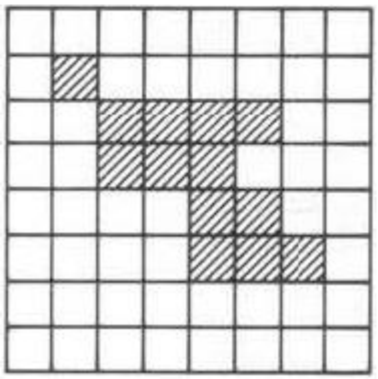


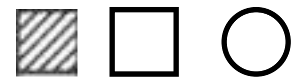


Primitives

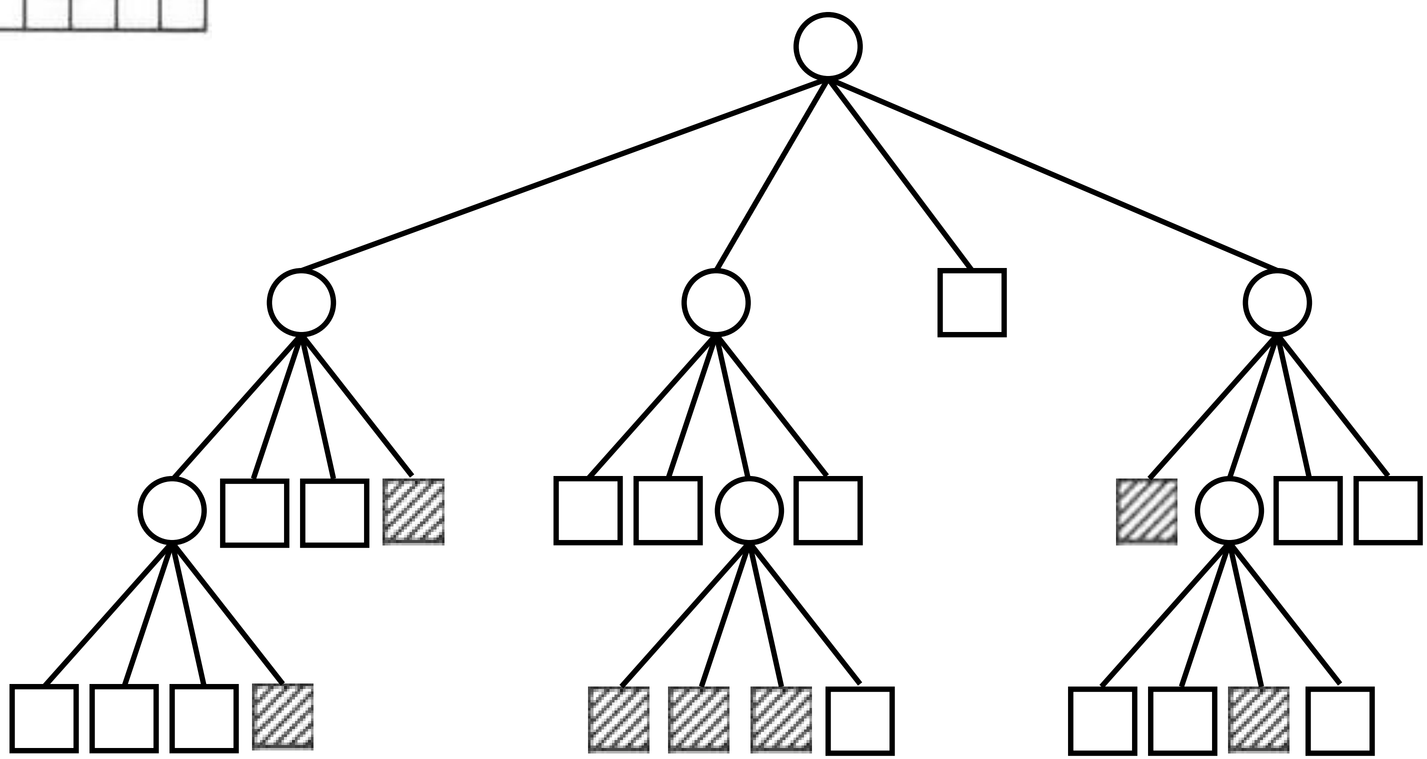


Representation using Tree (2)



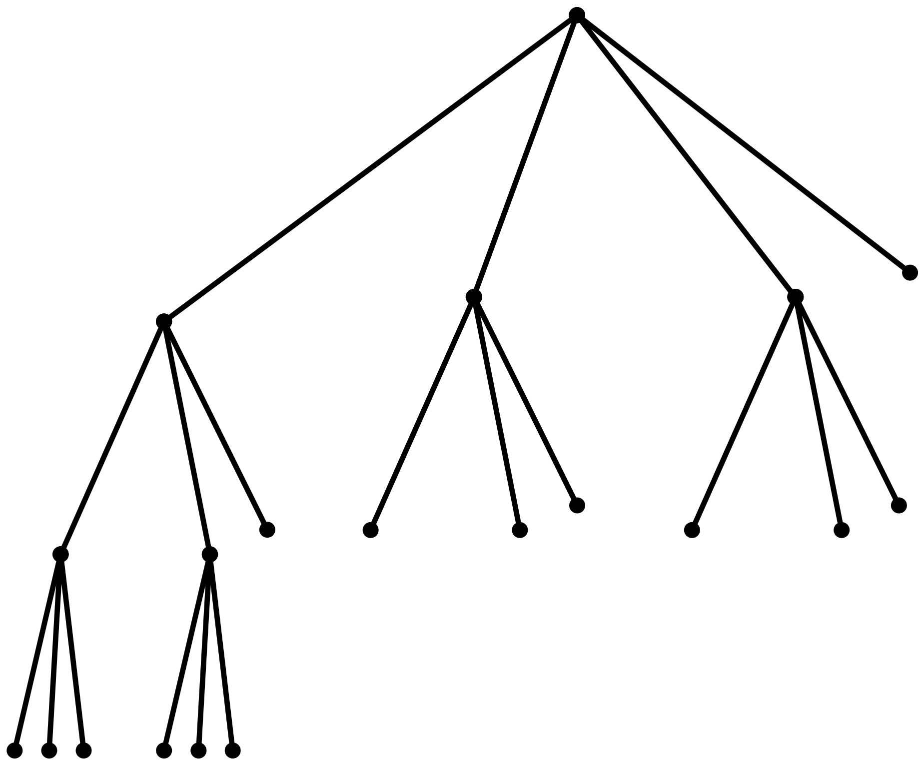


Primitives



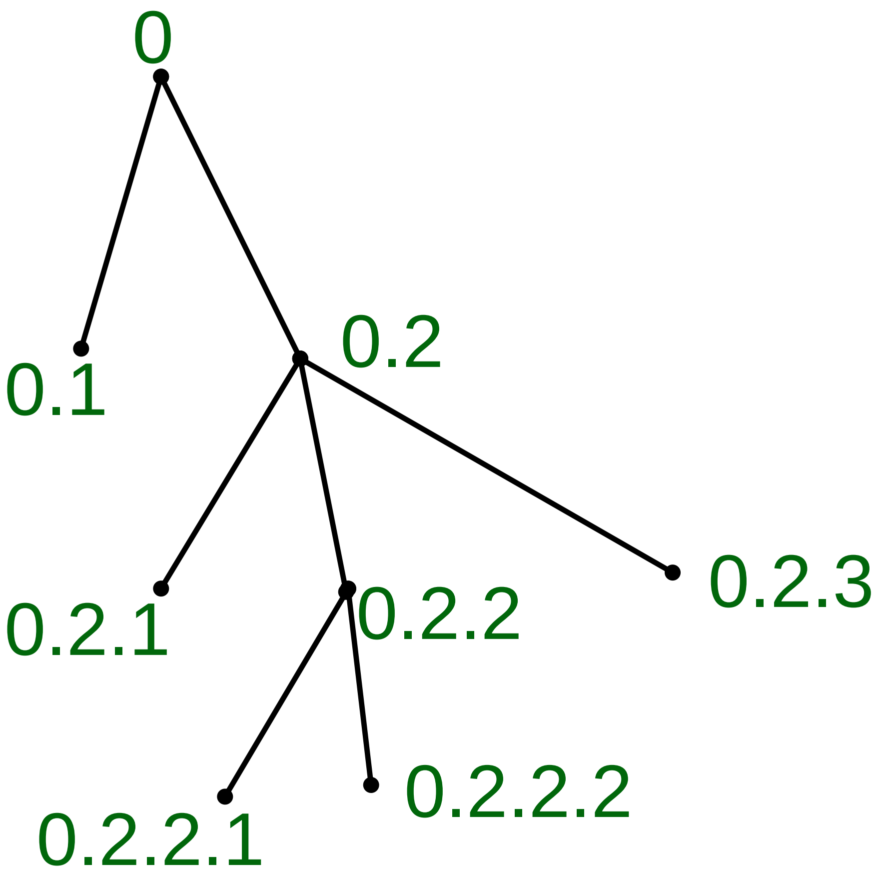
Tree Traversal and Representation

- Many methodologies
  - DFS, BFS, . . .



Tree Traversal and Representation

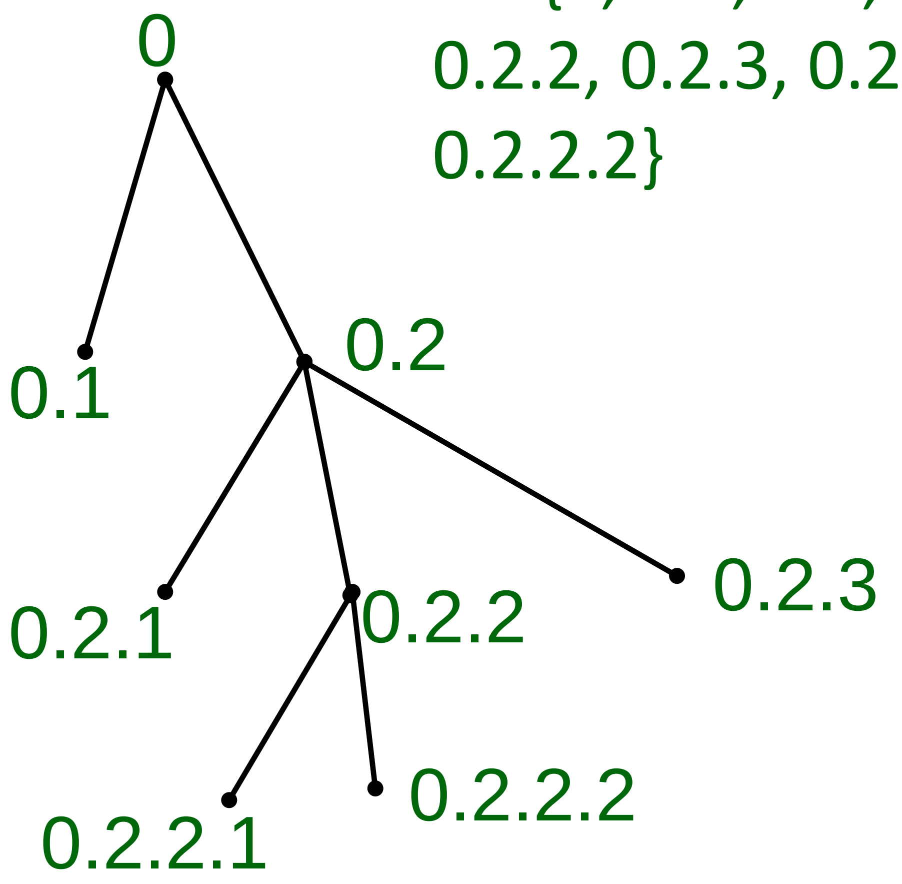
- Many methodologies
  - DFS, BFS, . . .



Tree Traversal and Representation

- Many methodologies
  - DFS, BFS, . . .

- T = {0, 0.1, 0.2, 0.2.1, 0.2.2, 0.2.3, 0.2.2.1, 0.2.2.2}



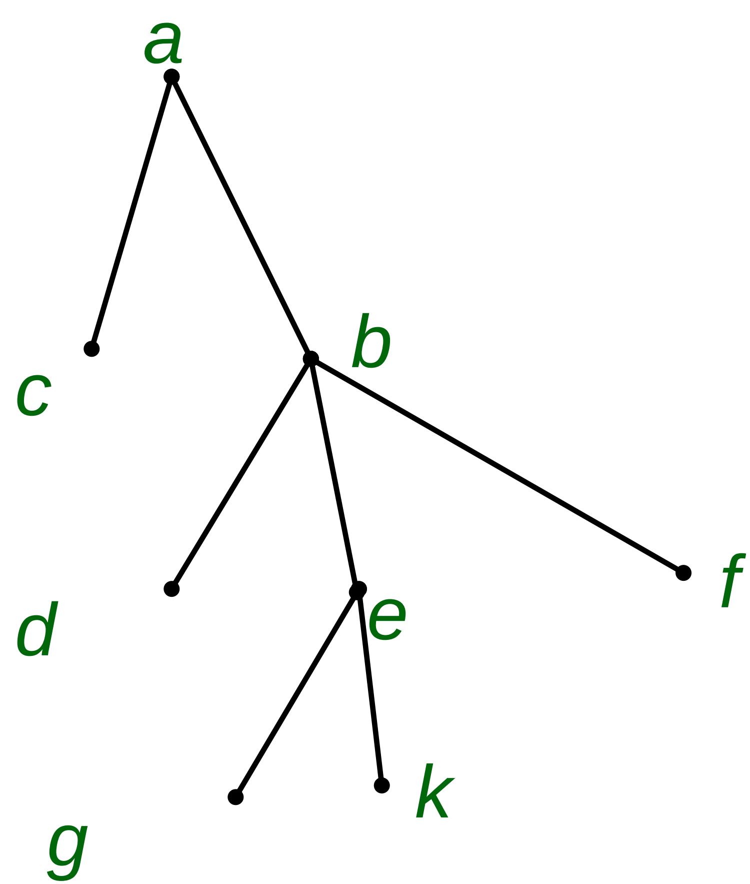
Tree Grammar

- A tree grammar G_T is defined as
  - $G_T = \{V, r, P, S\}$
  - where
    - P = set of productions involving trees
    - $V = V_T \cup V_N$
    - S= starting or root trees, T_V
    - *r = rank of a node*

Tree Grammar

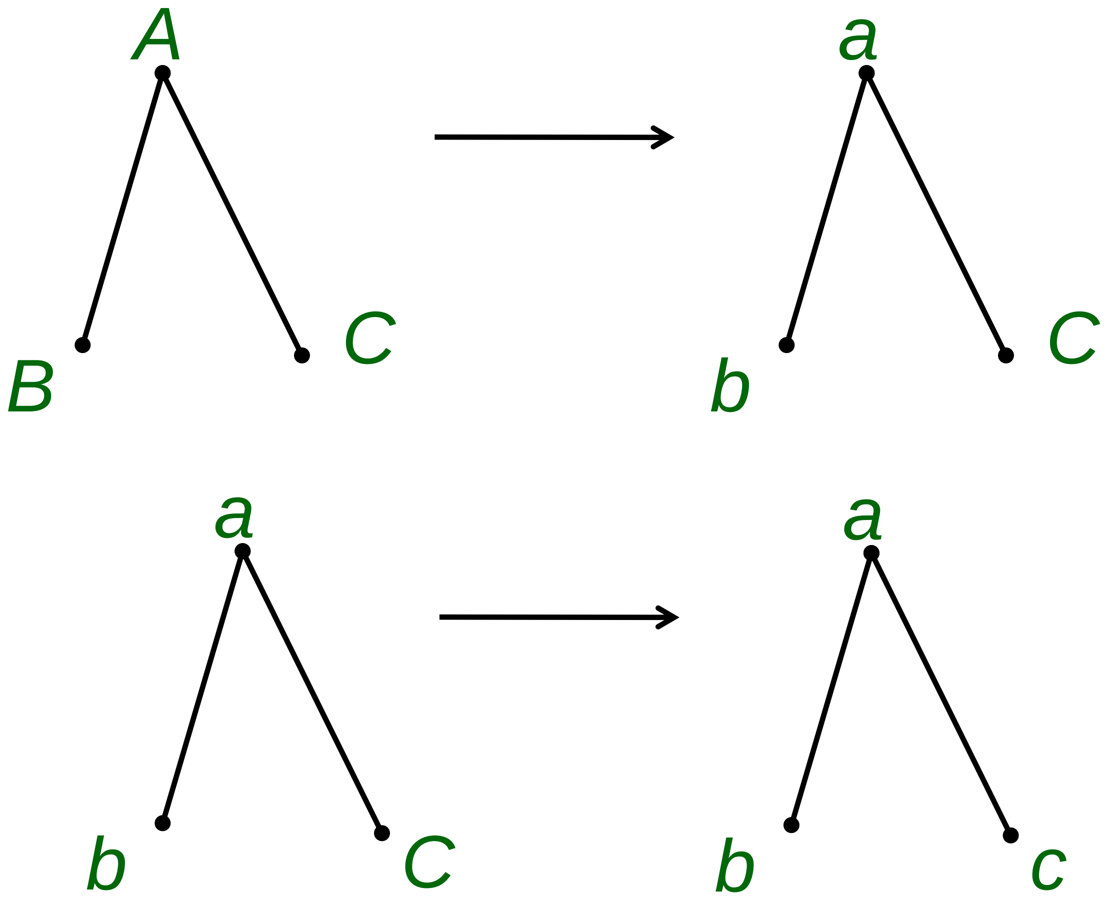
- Rank of a node
  - $r(a) = 2$
  - $r(b) = 3$

  - Therefore, this is out-degree of a node



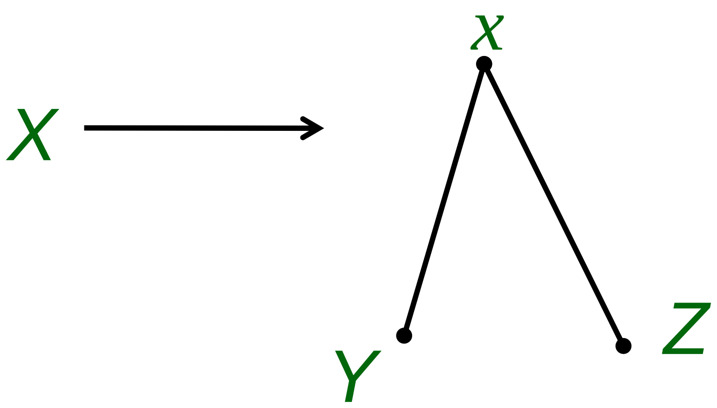
Productions of Tree Grammar

- Preserve the same structure but replace some terminal and/or non-terminals



Productions of Tree Grammar

- Expansive form:
  - $X \text{ -> } xX_1 . . . X_n$



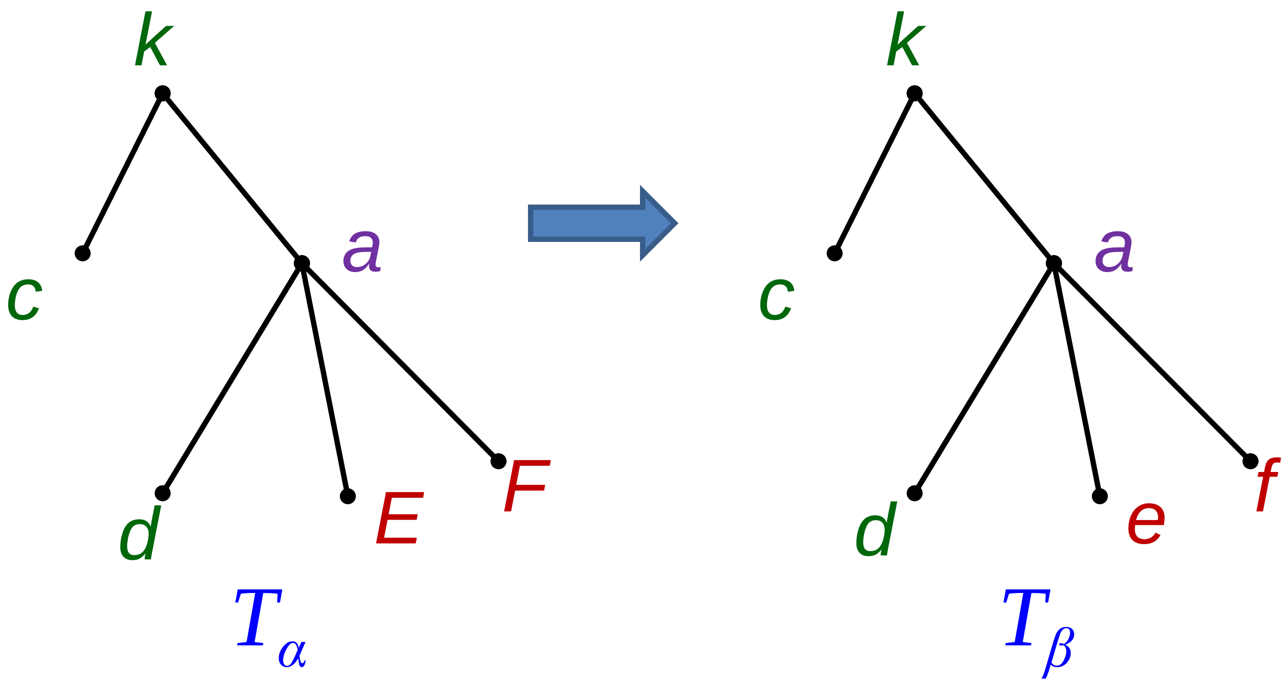
$r(x) = 2$

$A \longrightarrow b \qquad r(b) = 0$

$C \longrightarrow c \qquad r(c) = 0$

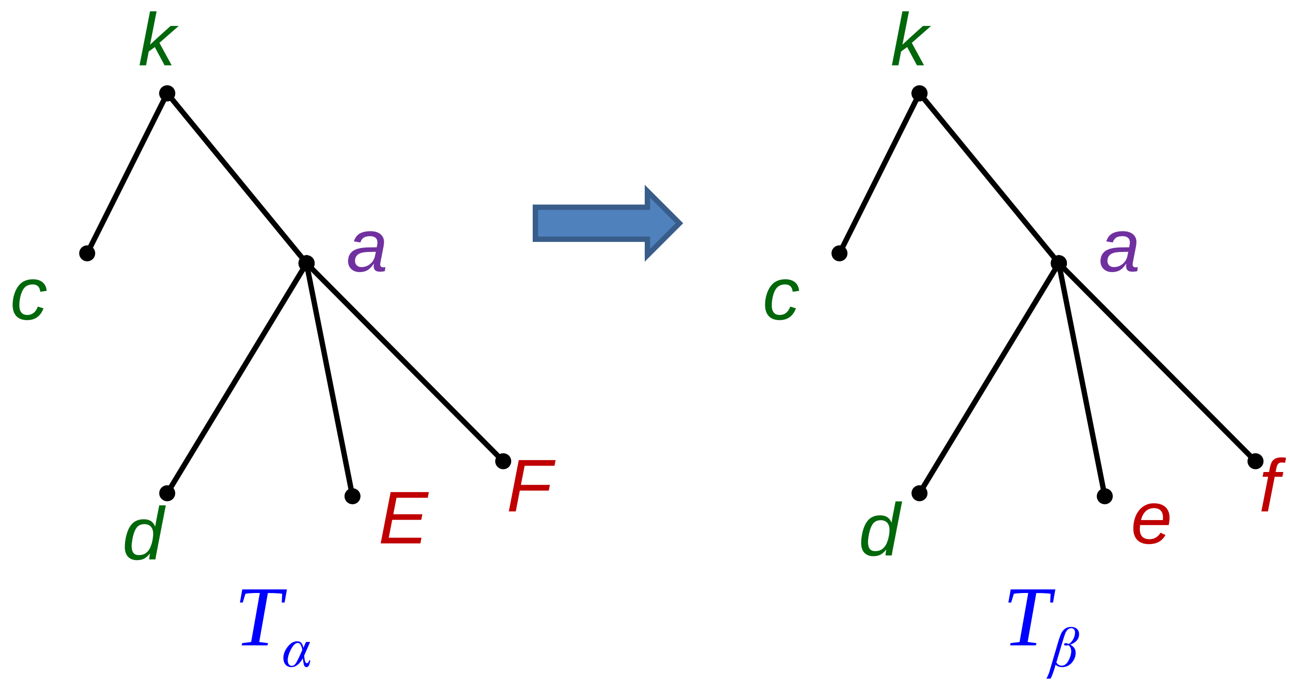
Derivation in a Tree Grammar

- A derivation $T_\alpha \overset{a}{\Rightarrow} T_\beta$ means:

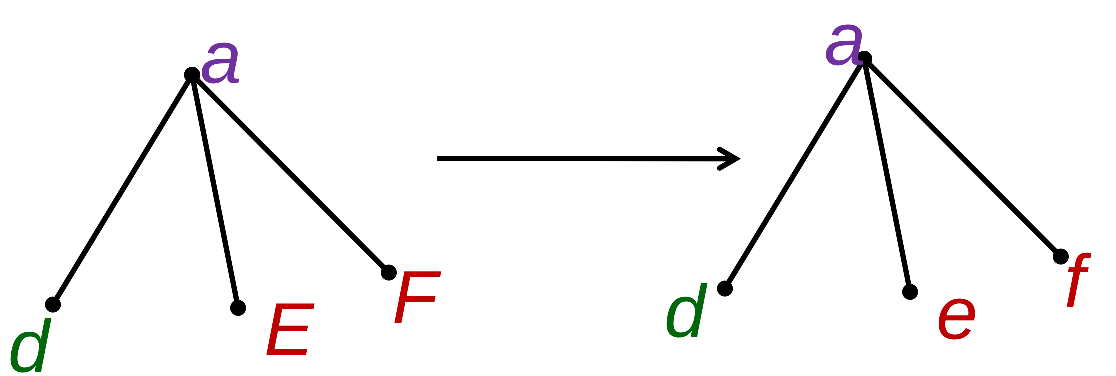


Derivation in a Tree Grammar

- A derivation $T_\alpha \overset{a}{\Rightarrow} T_\beta$ means:



- The production rule applied:



Language of A Tree Grammar

$$L(T_G) = \{T \mid T \in T_{V_T} \cap T_i \Rightarrow T \qquad T_i \in S\}$$

Stochastic Grammar

- Formal grammars allow
  - no error
  - no ambiguity
  - no structural deformation

Stochastic Grammar

- Formal grammars allow
  - no error
  - no ambiguity
  - no structural deformation
- In formal grammar,

$$L(G_1) \cap L(G_2) = \phi$$

- Often this may be unrealistic

Stochastic Grammar

- Stochastic grammar is defined as

$$G_s = \{V_N, V_T, P_s, S_s\}$$

- The Production P_s is of the form

$$\alpha_i \xrightarrow{p_{ij}} \beta_j$$

Stochastic Grammar

- Stochastic grammar is defined as

$$G_s = \{V_N, V_T, P_s, S_s\}$$

- The Production P_s is of the form

$$\alpha_i \xrightarrow{p_{ij}} \beta_j$$

because, there are multiple probabilistic consequents

Proper Stochastic Grammar

$$\alpha_i \xrightarrow{p_{ij}} \beta_j$$

- A stochastic grammar is proper if

$$\alpha_i \in V_N$$

$$\beta_j \in (V_T \cup V_N)^+$$

- There exist n_i no. of β_j so that

$$\boxed{\sum_{j=1}^{n_i} p_{ij} = 1}$$

Characteristic Stochastic Grammar

- A stochastic grammar is a characteristic grammar
  - when probabilities are removed from the productions

that is, $\alpha_i \xrightarrow{p_{ij}} \beta_j$ replaces $\alpha_i \rightarrow \beta_j$

Classification of Stochastic Grammars

- A stochastic grammar is classified based on its corresponding characteristic grammar:
  - Type 0
  - Type 1
  - Type 2
  - Type 3

Derivation in Stochastic Grammar

- Let there are *m* production rules, p^i, 1 <= *i* <= *m*
- we need to derive:

$$S \Rightarrow x$$

in a number of *n* steps

$$S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \cdots \Rightarrow \alpha_n = x$$

Derivation in Stochastic Grammar

- Let there are m production rules, p^i, 1 <= i <= m

- we need to derive:

$$S \Rightarrow x$$

in a number of n steps

$$S = \alpha_0 \stackrel{t_{0,1}}{\Rightarrow} \alpha_1 \stackrel{t_{1,2}}{\Rightarrow} \alpha_2 \cdots \stackrel{t_{n-1,n}}{\Rightarrow} \alpha_n = x$$

where, t_{ij} is a production rule applied in the step

$$\alpha_i \stackrel{t_{i,j}}{\Rightarrow} \alpha_j$$

Derivation in Stochastic Grammar

- The probability of generating

$$S = \alpha_0 \overset{t_{0,1}}{\Longrightarrow} \alpha_1 \overset{t_{1,2}}{\Longrightarrow} \alpha_2 \cdots \overset{t_{n-1,n}}{\Longrightarrow} \alpha_n = x$$

is

$$P(t_{0,1} \cap t_{1,2} \cap \cdots \cap t_{n-1,n})$$

Derivation in Stochastic Grammar

- The probability of generating

$$S = \alpha_0 \overset{t_{0,1}}{\Rightarrow} \alpha_1 \overset{t_{1,2}}{\Rightarrow} \alpha_2 \cdots \overset{t_{n-1,n}}{\Rightarrow} \alpha_n = x$$

is

$$P(t_{0,1} \cap t_{1,2} \cap \cdots \cap t_{n-1,n})$$
$$= P(t_{0,1}, t_{1,2}, \cdots, t_{n-2,n-1}, t_{n-1,n})$$
$$= P(t_{n-1,n}, t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1})$$

Derivation in Stochastic Grammar

$$P(t_{n-1,n}, t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1})$$

$$= P(t_{n-1,n} \mid t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1}) \times P(t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1})$$

Derivation in Stochastic Grammar

$$P(t_{n-1,n}, t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1})$$

$$= P(t_{n-1,n} \mid t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1}) \times P(t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1})$$

$$= P(t_{n-1,n}) \times P(t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1})$$

- Assumes that the probability of applying the next production is independent of previously applied productions

Derivation in Stochastic Grammar

$$P(t_{n-1,n}, t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1})$$

$$= P(t_{n-1,n} \mid t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1}) \times P(t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1})$$

$$= P(t_{n-1,n}) \times P(t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1})$$

$$= P(t_{n-1,n}) \times P(t_{n-2,n-1}) \times P(t_{n-3,n-2} \cdots, t_{1,2}, t_{0,1})$$

- Applying the same approach

Derivation in Stochastic Grammar

$$P(t_{n-1,n}, t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1})$$

$$= P(t_{n-1,n} \mid t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1}) \times P(t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1})$$

$$= P(t_{n-1,n}) \times P(t_{n-2,n-1}, \cdots, t_{1,2}, t_{0,1})$$

$$= P(t_{n-1,n}) \times P(t_{n-2,n-1}) \times P(t_{n-3,n-2} \cdots, t_{1,2}, t_{0,1})$$

$$= \prod_{q=1}^{n} P(t_{q-1,q})$$

Stochastic Language

- A stochastic language is of the form

$$L(G_s) = \{(x, p(x) \mid x \in V_T^+, S_s \overset{p_j}{\Rightarrow} x, j = 1,2,\cdots k, \text{and } p(x) = \sum_{j=1}^{k} p_j\}$$

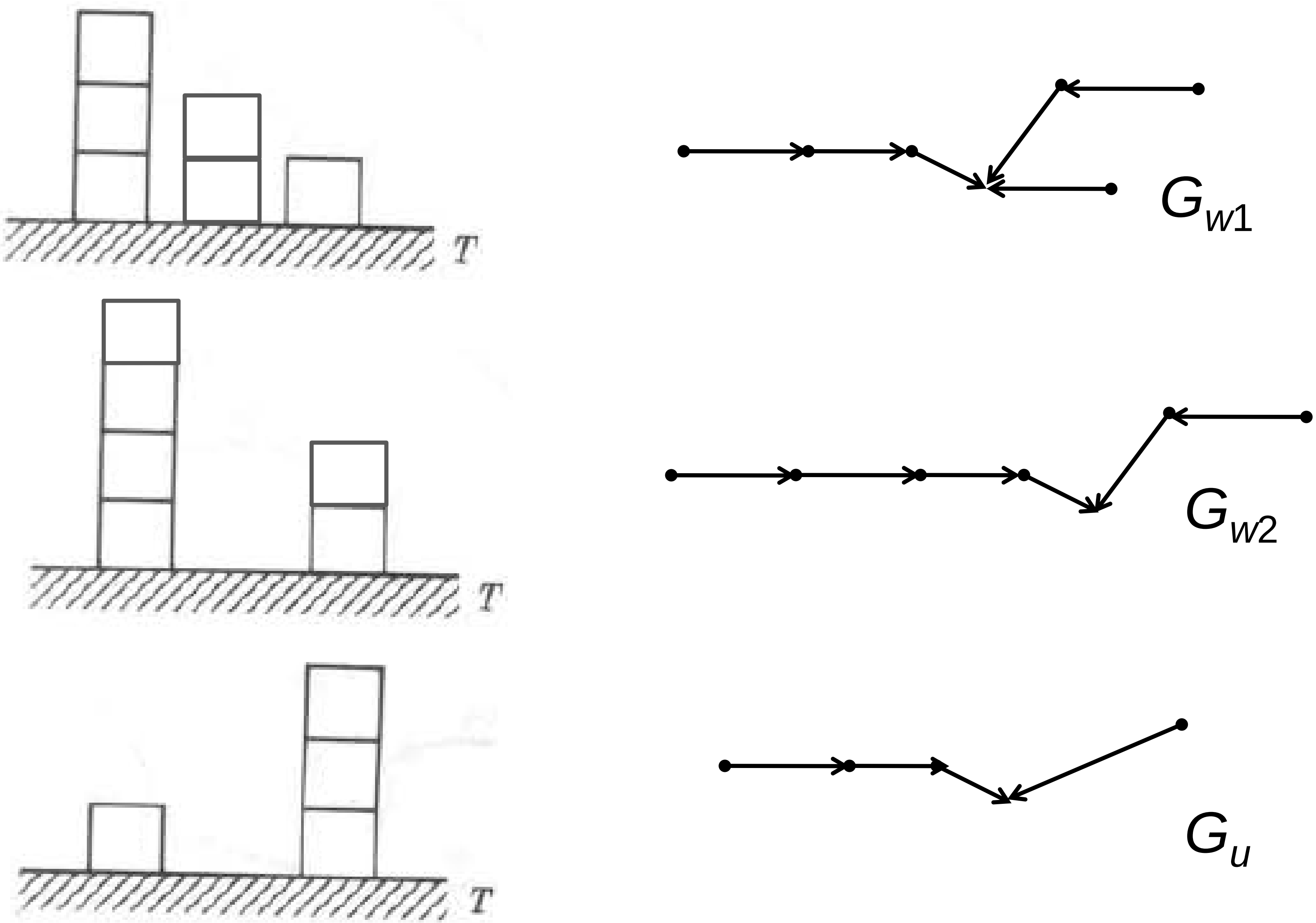
Syntactic Pattern Recognition using Graph Theory

Graphical Approaches to SyntPR

- Graphical alternatives to represent structures or relational information

- Natural extensions of higher dimensional grammars

Graphical Description of Patterns



Graph Representation

- A graph $G = \{N, R\}$ is an ordered pair represented using:
  - a set of nodes (vertices), N,
  - a set of edges (arcs), $R \subseteq N \times N$

Graph Representation

- A sub-graph $G_s = \{N_s, R_s\}$ is itself a graph, where
  - $N_s \subseteq N$
  - $R_s \subseteq R$, however, R_s consists of edgese that connects only the nodes in N_s.

Graph Representation

- A graph is *connected* if there is a path between all pairs of its nodes.
- A graph is *complete* if there is an edge between all pairs of its nodes.

Other definitions related to Graph

- In *directional graphs* (digraphs), edges have directional significance, i.e., $(a, b) \in R$ means there is an edge from node a to node b.
- When the direction of edges in a graph is not important, i.e., specification of either (a, b) or $(b, a) \in R$ is acceptable, the graph is an *undirected graph*.

Other definitions related to Graph

- A *relational graph* represents a particular relation graphically using arrows to show this relation between the elements as a directed graph.

Classification Task by Comparing Relational Graph Descriptions

Naïve approach:

- Represent each class as a *prototypical relational graph*
- Convert unknown pattern to *relational graph*
- Compare with the library of prototypes

Classification Task by Comparing Relational Graph Descriptions

- The observed data rarely matches a stored relational representation "exactly",

- We need other approaches
  - graph similarity should be measured.

Classification Task by Comparing Relational Graph Descriptions

- One approach is to check whether the observed data match a “portion” of a relational model.
  - Case 1: Any relation not present in both graphs is a failure.
  - Case 2: Any single match of a relation is a success.
  - A realistic strategy is somewhere in between these extremes.

Comparing Relational Graph Descriptions

- We define some terms for graph similarity
  - Use graph *homomorphism* and/or *isomorphism*

Graph Homomorphism

- Consider two graphs $G_1 = \{N_1, R_1\}$ and $G_2 = \{N_2, R_2\}$.
- A *homomorphism* from G_1 to G_2 is a function f from N_1 to N_2:

such that

$$(v_1, w_1) \in R_1 \Rightarrow (f(v_1), f(w_1)) \in R_2$$

Graph Isomorphism

- A stricter term

► A stricter test is that of *isomorphism*, where f is required to be 1:1 and onto:

$$(v_1, w_1) \in R_1 \Leftrightarrow (f(v_1), f(w_1)) \in R_2$$

► Isomorphism simply states that relabeling of nodes yields the same graph structure.

Graph Isomorphism Test: *Adjaceny Matrix*

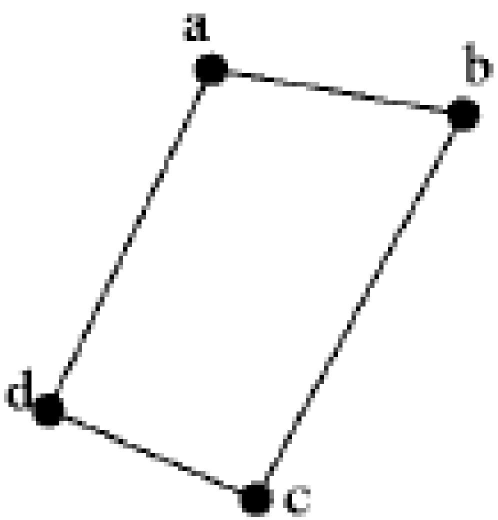
- A digraph G with p nodes can be converted to an *adjacency matrix*:
  - Number each node by an index $\{1, \ldots, p\}$.
  - Represent the existence or absence of an edge as

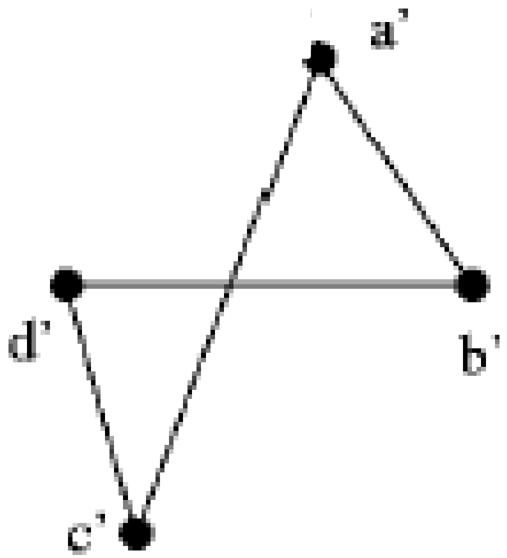
$$\text{Adj}(i, j) = \begin{cases} 1 & \text{if } G \text{ contains an edge from node } i \text{ to node } j, \\ 0 & \text{otherwise.} \end{cases}$$

Graph Isomorphism Test

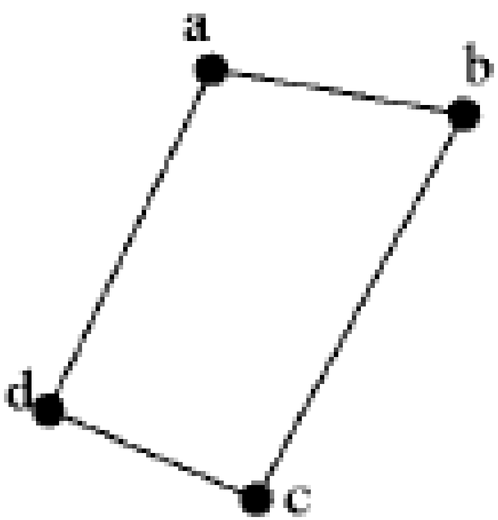
- Given two graphs G_1 and G_2 each with p nodes, to determine isomorphism:
  - Label the nodes of each graph with labels $1, \ldots, p$.
  - Form the adjacency matrices M_1 and M_2 for both graphs.
  - If $M_1 = M_2$, G_1 and G_2 are isomorphic.
  - Otherwise, consider all the $p!$ possible labelings on G_2.

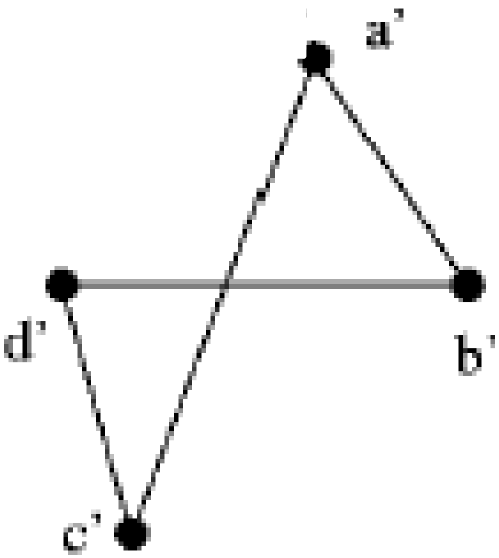
Graph Isomorphism Test





Graph Isomorphism Test

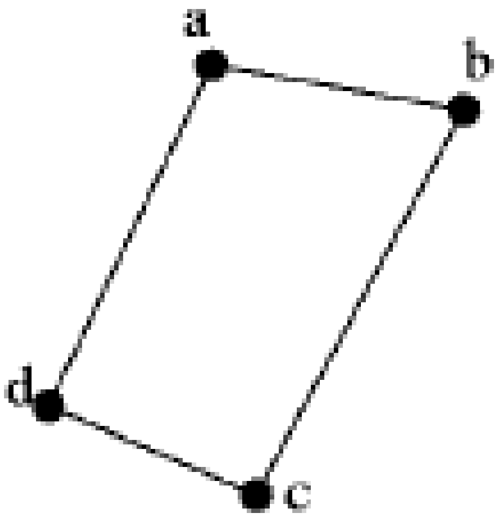




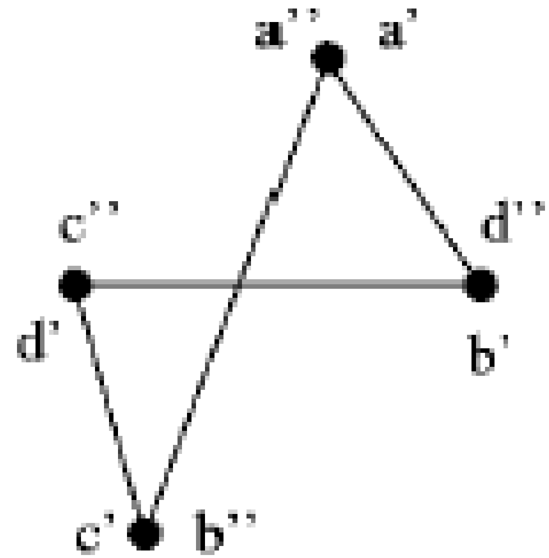
	a	b	c	d
a	0	1	0	1
b	1	0	1	0
c	0	1	0	1
d	1	0	1	0

	a'	b'	c'	d'
a'	0	1	1	0
b'	1	0	0	1
c'	1	0	0	1
d'	0	1	1	0

Graph Isomorphism Test



$$f(a') = a''$$
$$f(b') = d''$$
$$f(c') = b''$$
$$f(d') = c''$$



	a	b	c	d
a	0	1	0	1
b	1	0	1	0
c	0	1	0	1
d	1	0	1	0

	a"	b"	c"	d"
a"	0	1	0	1
b"	1	0	1	0
c"	0	1	0	1
d"	1	0	1	0

Graph Isomorphism Test

- Computationally expensive
- Allows only exact measure, not realistic
  - Many relations may not be observed in practical examples

Graph Isomorphism Test: Quick Reject

- Check for following properties
  - Number of nodes
  - Number of edges

Graph Isomorphism Test: Quick Reject

- These are useful, too:
  - in-degree *i* of a node
  - out-degree *j* of a node
  - degree *k* of a node (for undirected graph)
  - closed path of length *l*

Alternate Graph Isomorphism Test: Sub-isomorphism

- G_1 and G_2 are called *subisomorphic* if a subgraph of G_1 is isomorphic to a subgraph of G_2.

- Clearly, this is a less restrictive structural match than that of isomorphism.

Alternate Graph Isomorphism Test: Sub-isomorphism

- G_1 and G_2 are called *subisomorphic* if a subgraph of G_1 is isomorphic to a subgraph of G_2.
- Clearly, this is a less restrictive structural match than that of isomorphism.
- However, determining subisomorphism is also computationally expensive.

Extension to Graph Matching Approach

- To allow structural deformations, numerous extensions to graph matching have been proposed.
  - Extract features from graphs G_1 and G_2 to form feature vectors x_1 and x_2, respectively, and use statistical pattern recognition techniques to compare x_1 and x_2.
  - Use a matching metric as the minimum number of transformations necessary to transform G_1 into G_2.

Extension to Graph Matching Approach

- Common transformations include:
  - Node insertion,
  - Node deletion,
  - Node splitting,
  - Node merging,
  - Edge insertion,
  - Edge deletion.

Extension to Graph Matching Approach

- $\boxed{\text{Common transformations}}$ include:
  - Node insertion,
  - Node deletion,
  - Node splitting,
  - Node merging,
  - Edge insertion,
  - Edge deletion.

- Computation is still high
- Difficult to design a suitable distance measure to distinguish structural deformations

Extension to Graph Matching Approach

- An *attributed graph* $G = \{N, P, R\}$ is a 3-tuple where
  - N is a set of nodes,
  - P is a set of properties of these nodes,
  - R is a set of relations between nodes.

Matching Through Attributed Graph

- Let $p_q^i(n)$ denote the value of the q'th property of node n of graph G_i.

- Nodes $n_1 \in N_1$ and $n_2 \in N_2$ are said to form an assignment (n_1, n_2) if

$$p_q^1(n_1) \sim p_q^2(n_2)$$

where "$\sim$" denotes similarity.

Matching Through Attributed Graph

- Let $r^i_j(n_x, n_y)$ denote the j'th relation involving nodes $n_x, n_y \in N_i$.

- Two assignments (n_1, n_2) and (n'_1, n'_2) are considered *compatible* if

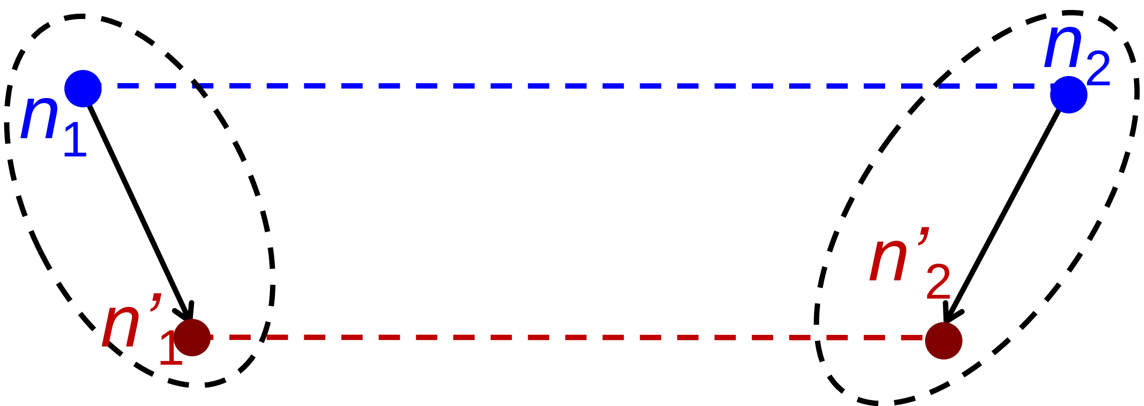
$$r^1_j(n_1, n'_1) \sim r^2_j(n_2, n'_2) \quad \forall j.$$

Matching Through Attributed Graph

- Let $r^i_j(n_x, n_y)$ denote the j'th relation involving nodes $n_x, n_y \in N_i$.

- Two assignments (n_1, n_2) and (n'_1, n'_2) are considered *compatible* if

$$r^1_j(n_1, n'_1) \sim r^2_j(n_2, n'_2) \quad \forall j.$$



Matching Through Attributed Graph

- Let $r_j^i(n_x, n_y)$ denote the j'th relation involving nodes $n_x, n_y \in N_i$.

- Two assignments (n_1, n_2) and (n_1', n_2') are considered *compatible* if

$$r_j^1(n_1, n_1') \sim r_j^2(n_2, n_2') \quad \forall j.$$

- Two attributed graphs G_1 and G_2 are isomorphic if there exists a set of 1:1 assignments of nodes in G_1 to nodes in G_2 such that all assignments are compatible.

Matching Through Attributed Graph

- A strategy for measuring the similarity between two attributed graphs is to find node pairings using the cliques of a match graph.

G_1 G_2

↓

Match Graph

↓

Maximal Clique

↓

Calculate Measure of Graph Similarity

Matching Through Attributed Graph

- A *match graph* is formed from two graphs G_1 and G_2 as follows:
  - Nodes of the match graph are assignments from G_1 to G_2.
  - An edge in the match graph exists between two nodes if the corresponding assignments are compatible.

- A *clique* of a graph is a totally connected subgraph.
- A *maximal clique* is not included in any other clique.

Approaches in Matching Through Attributed Graph

- Find
  - the *attributed graphs* from the patterns
  - the *match graphs* from the attributed graph
  - the *maximum clique* from the match graph
  - the *similarity*

Recursive Procedure to Find Cliques

Input:

- *X*: an initial clique (possibly empty)
- *Y*: the graph

Output:

- The set of all maximal cliques

Recursive Procedure to Find Cliques

Procedure *clique* (X, Y)

  Form $Y - X$;

  If a node y in $Y - X$ is connected to all nodes of X,

  Then return *cliques* ($X \cup \{y\}, Y$) U *cliques* ($X, Y - \{y\}$)

  Else return X

End

Recursive Procedure to Find Cliques

Procedure *clique* (X, Y)

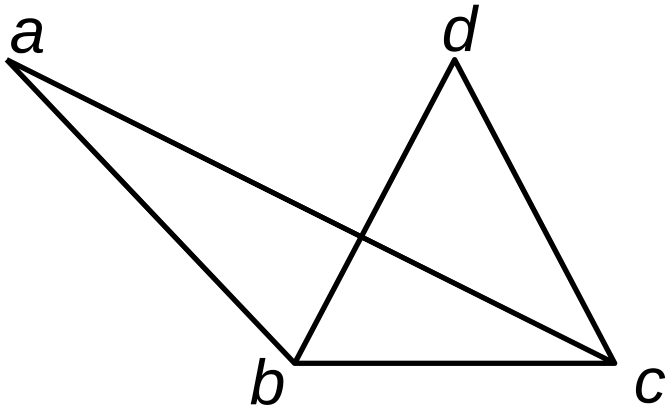
Form $Y - X$;

If a node y in $Y - X$ is connected to all nodes of X,

Then return *cliques* ($X \cup \{y\}, Y$) $\cup$ *cliques* ($X, Y - \{y\}$)

Else return X

End



Recursive Procedure to Find Cliques

Procedure *clique* (X, Y)

Form $Y - X$;

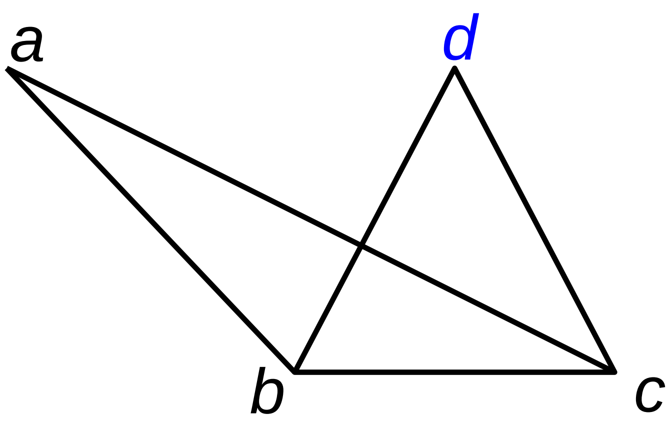
If a node y in $Y - X$ is connected to all nodes of X,

Then return *cliques* ($X \cup \{y\}, Y$) $\cup$ *cliques* ($X, Y-\{y\}$)

Else return X

End

Find *clique* (d, Y)?



Recursive Procedure to Find Cliques

Procedure *clique* (X, Y)

Form $Y - X$;

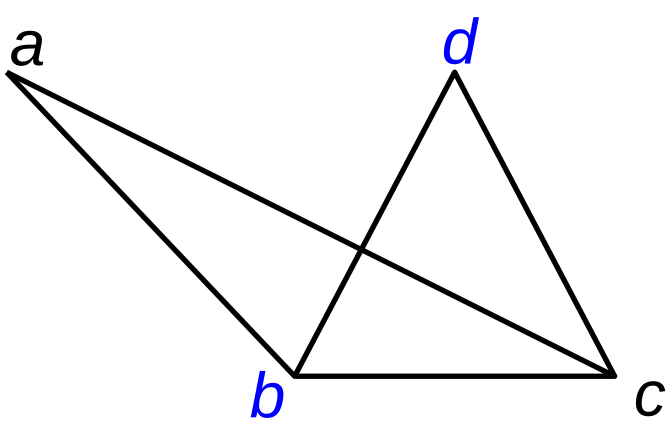
If a node y in $Y - X$ is connected to all nodes of X,

Then return *cliques* ($X \cup \{y\}, Y$) U *cliques* ($X, Y - \{y\}$)

Else return X

End

clique (d, Y) = *clique* ($\{d, b\}$, Y) U *clique* (d, $\{a, c, d\}$)



Recursive Procedure to Find Cliques

Procedure *clique* (X, Y)

Form $Y - X$;

If a node y in $Y - X$ is connected to all nodes of X,

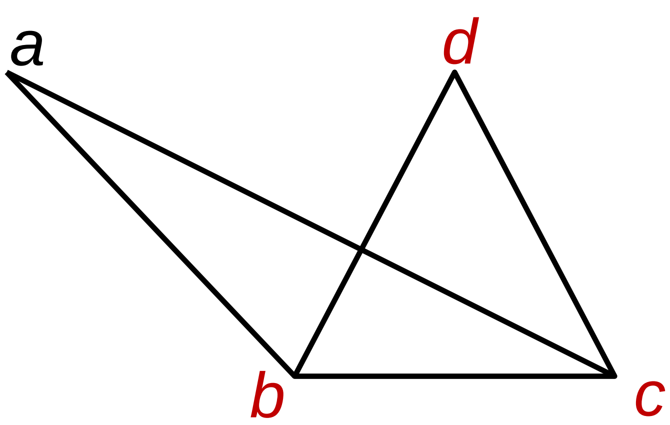
Then return *cliques* ($X \cup \{y\}, Y$) U *cliques* ($X, Y-\{y\}$)

Else return X

End

clique (d, Y) = *clique* ($\{d, b\}$, Y) U *clique* (d, $\{a, c, d\}$)

clique ($\{d, b\}$, Y) = *clique* ($\{d, b, c\}$, Y) U *clique* ($\{d, b\}$, $\{a, b, d\}$)



Recursive Procedure to Find Cliques

Procedure *clique* (*X*, *Y*)

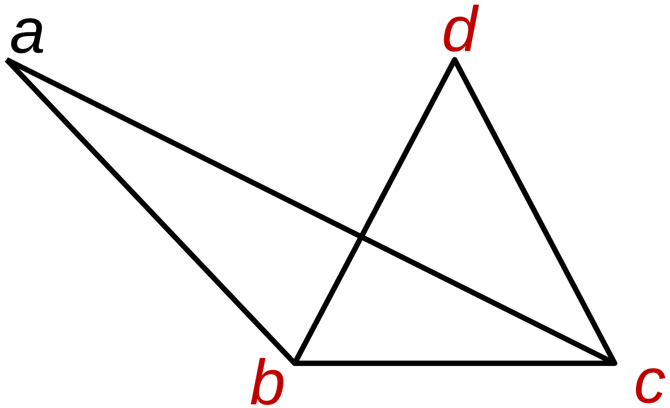
Form *Y* – *X*;

If a node *y* in *Y* – *X* is connected to all nodes of *X*,

Then return *cliques* (*X* U {*y*},*Y*) U *cliques* (*X*,*Y*-{*y*})

Else return *X*

End



clique (*d*, *Y*) = *clique* ({*d*, *b*}, *Y*) U *clique* (*d*, {*a*, *c*, *d*})

clique ({*d*, *b*}, *Y*) = *clique* ({*d*, *b*, *c*}, *Y*) U *clique* ({*d*, *b*}, {*a*, *b*, *d*})

⇓

{*d*, *b*}

Recursive Procedure to Find Cliques

Procedure *clique* (*X*, *Y*)

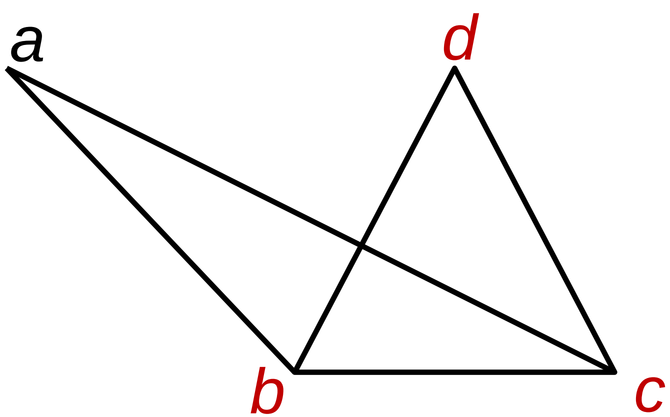
Form *Y* – *X*;

If a node *y* in *Y* – *X* is connected to all nodes of *X*,

Then return *cliques* (*X* U {*y*},*Y*) U *cliques* (*X*,*Y*-{*y*})

Else return *X*

End



clique (*d*, *Y*) = *clique* ({*d*, *b*}, *Y*) U *clique* (*d*, {*a*, *c*, *d*})

clique ({*d*, *b*}, *Y*) = *clique* ({*d*, *b*, *c*}, *Y*) U {*d*, *b*}

{*d*, *b*, *c*}

Recursive Procedure to Find Cliques

Procedure *clique* (*X*, *Y*)

Form *Y* – *X*;

If a node *y* in *Y* – *X* is connected to all nodes of *X*,

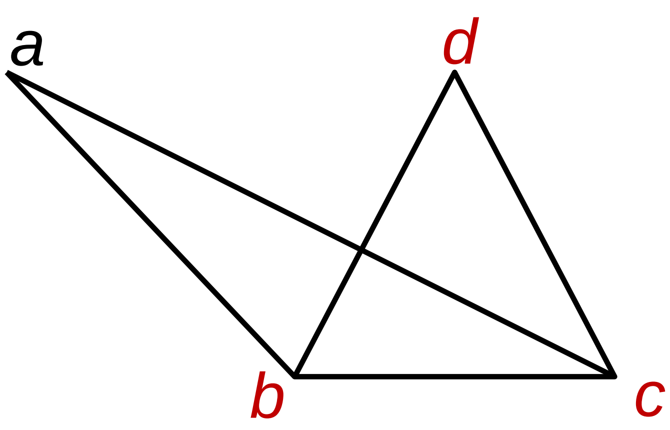
Then return *cliques* (*X* U {*y*},*Y*) U *cliques* (*X*,*Y*-{*y*})

Else return *X*

End

clique (*d*, *Y*) = *clique* ({*d*, *b*}, *Y*) U *clique* (*d*, {*a*, *c*, *d*})

clique ({*d*, *b*}, *Y*) = {*d*, *b*, *c*} U {*d*, *b*}



Recursive Procedure to Find Cliques

Procedure *clique* (X, Y)

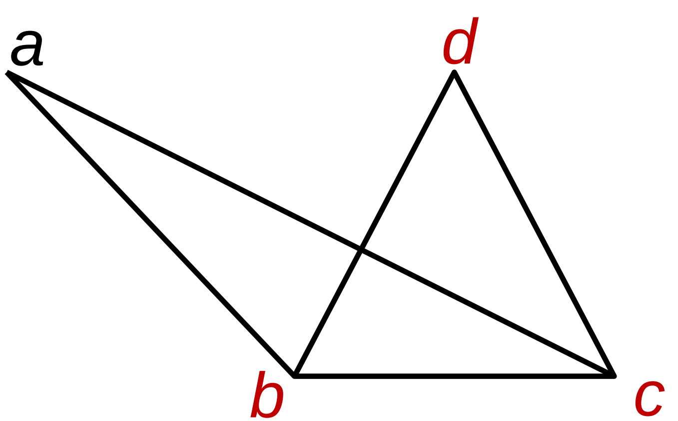
Form $Y - X$;

If a node y in $Y - X$ is connected to all nodes of X,

Then return *cliques* ($X \cup \{y\}, Y$) U *cliques* ($X, Y-\{y\}$)

Else return X

End



clique (d, Y) = *clique* ($\{d, b\}$, Y) U *clique* (d, $\{a, c, d\}$)

clique ($\{d, b\}$, Y) = $\{d, b, c\}$ U $\{d, b\}$

= $\{d, b, c\}$

Recursive Procedure to Find Cliques

Procedure *clique* (X, Y)

Form $Y - X$;

If a node y in $Y - X$ is connected to all nodes of X,

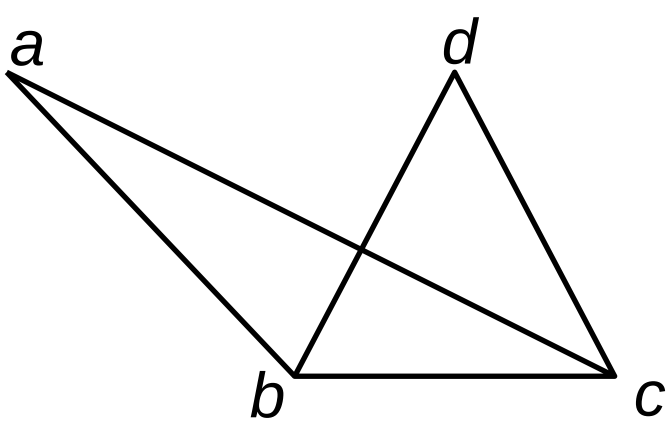
Then return *cliques* ($X \cup \{y\}, Y$) U *cliques* ($X, Y-\{y\}$)

Else return X

End

clique (d, Y) = $\{d, b, c\} \cup \{d, b\} \cup$

clique (d, $\{a, c, d\}$)



Recursive Procedure to Find Cliques

Procedure *clique* (X, Y)

Form $Y - X$;

If a node y in $Y - X$ is connected to all nodes of X,

Then return *cliques* ($X \cup \{y\}, Y$) $\cup$ *cliques* ($X, Y - \{y\}$)

Else return X

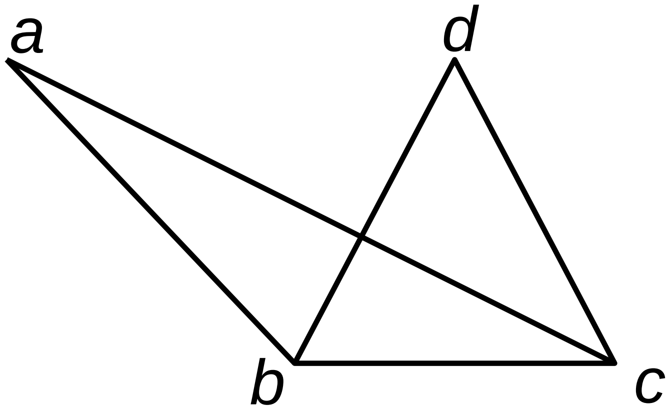
End

clique (d, Y) = $\{d, b, c\} \cup \{d, b\} \cup$ *clique* (d, $\{a, c, d\}$)

clique (d, $\{a, c, d\}$)

= *clique* ($\{d, c\}$, $\{a, c, d\}$)

$\cup$ *clique* (d, $\{a, d\}$)



Recursive Procedure to Find Cliques

Procedure *clique* (X, Y)

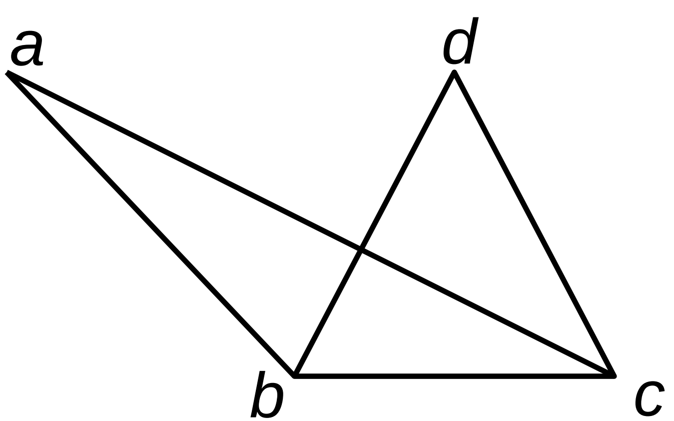
Form $Y - X$;

If a node y in $Y - X$ is connected to all nodes of X,

Then return *cliques* ($X \cup \{y\}, Y$) $\cup$ *cliques* ($X, Y - \{y\}$)

Else return X

End



clique (d, Y) = $\{d, b, c\} \cup \{d, b\} \cup$ *clique* (d, $\{a, c, d\}$)

clique (d, $\{a, c, d\}$)

= *clique* ($\{d, c\}$, $\{a, c, d\}$)

$\cup$ *clique* (d, $\{a, d\}$)

$=\{d, c\} \cup \{d\}$

Recursive Procedure to Find Cliques

Procedure *clique* (X, Y)

  Form $Y - X$;

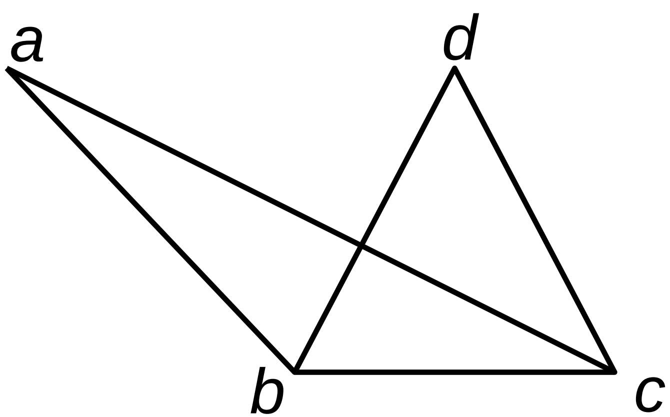
  If a node y in $Y - X$ is connected to all nodes of X,

  Then return *cliques* ($X \cup \{y\}, Y$) $\cup$ *cliques* ($X, Y-\{y\}$)

  Else return X

End

clique (d, Y) = $\{d, b, c\} \cup \{d, b\}$
$\cup \{d, c\} \cup \{d\}$



Recursive Procedure to Find Cliques

Procedure *clique* (*X*, *Y*)

Form *Y* – *X*;

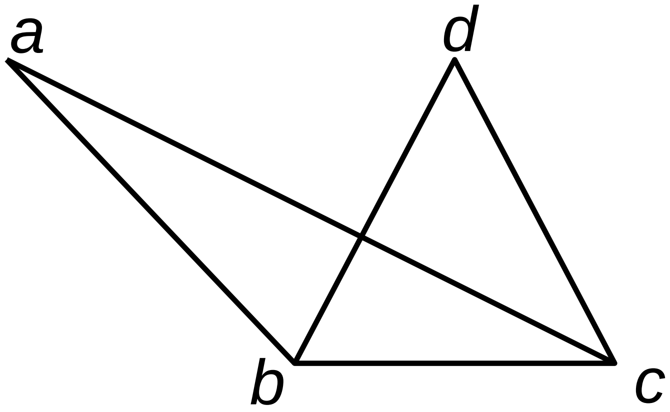
If a node *y* in *Y* – *X* is connected to all nodes of *X*,

Then return *cliques* (*X* U {*y*},*Y*) U *cliques* (*X*,*Y*-{*y*})

Else return *X*

End

The maximal clique is = {*d*, *b*, *c*}



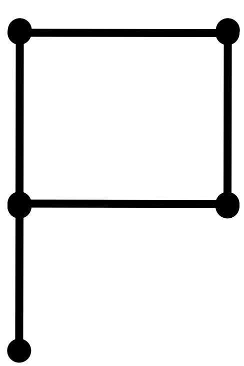
Example: Pattern Matching Using Attributed Graph

P T

Example: Pattern Matching Using Attributed Graph

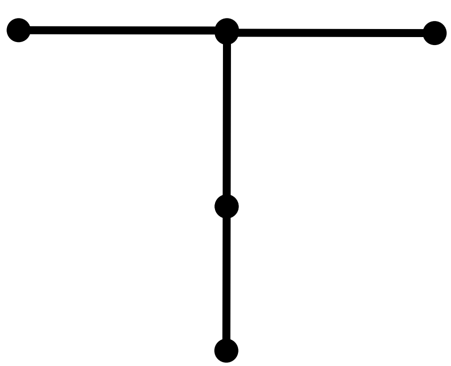
P

Reference



T

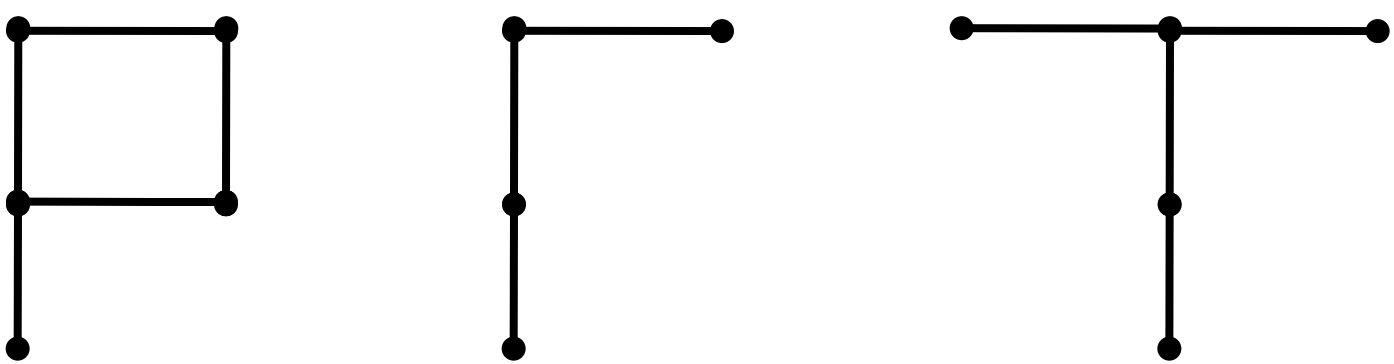
Reference



Example: Pattern Matching Using Attributed Graph

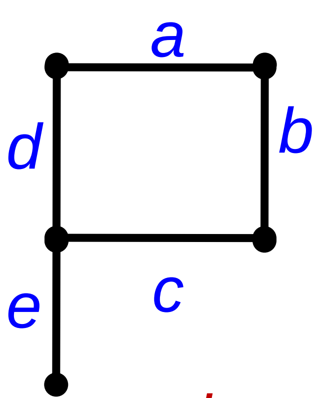
P T

Reference Test Reference

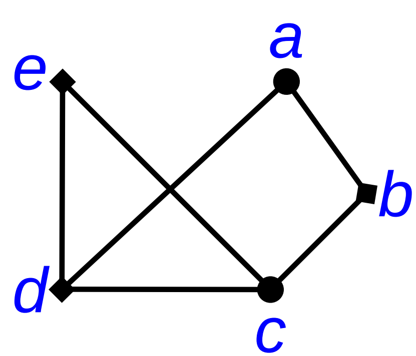


Example: Pattern Matching Using Attributed Graph

Find Attributed Graph for all patterns



I_1



AG_{I_1}

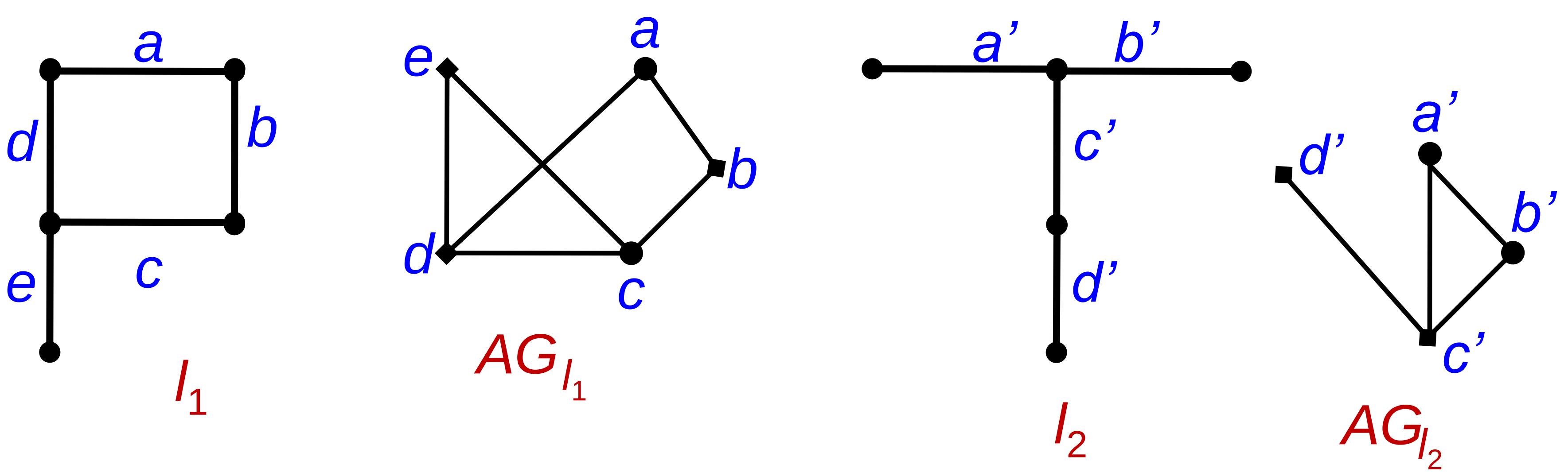
● Horizontal line

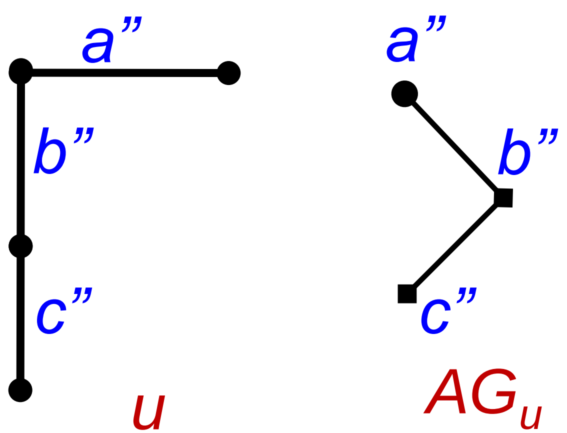
◊ Vertical line

Relation: —— connected

Example: Pattern Matching Using Attributed Graph

Find Attributed Graph for all patterns





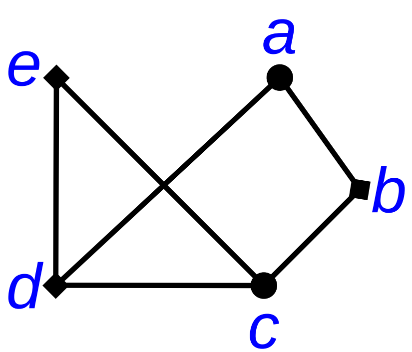
● Horizontal line

◊ Vertical line

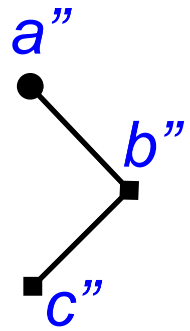
Relation: —— connected

Example: Pattern Matching Using Attributed Graph

Find Matched Graph between AGl_1 and AG_u



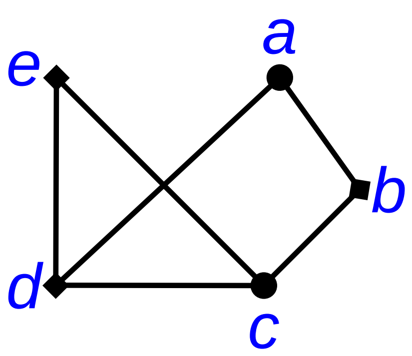
AG_{l_1}



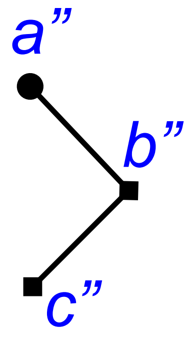
AG_u

Example: Pattern Matching Using Attributed Graph

Find Matched Graph between AGl_1 and AG_u



AG_{l_1}



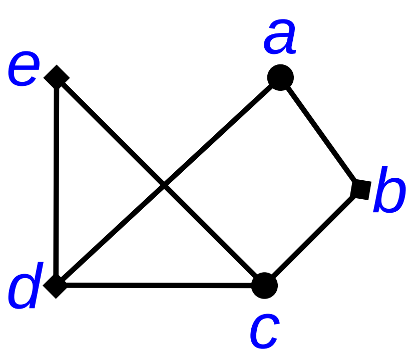
AG_u

Find all assignments between AGl_1 and AG_u:

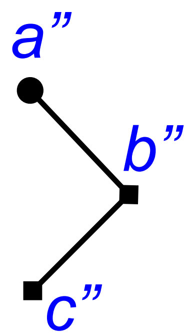
(a, a”), (b, b”), (b, c”), (c, a”), (d, b”), (d, c”), (e, b”), (e, c”)

Example: Pattern Matching Using Attributed Graph

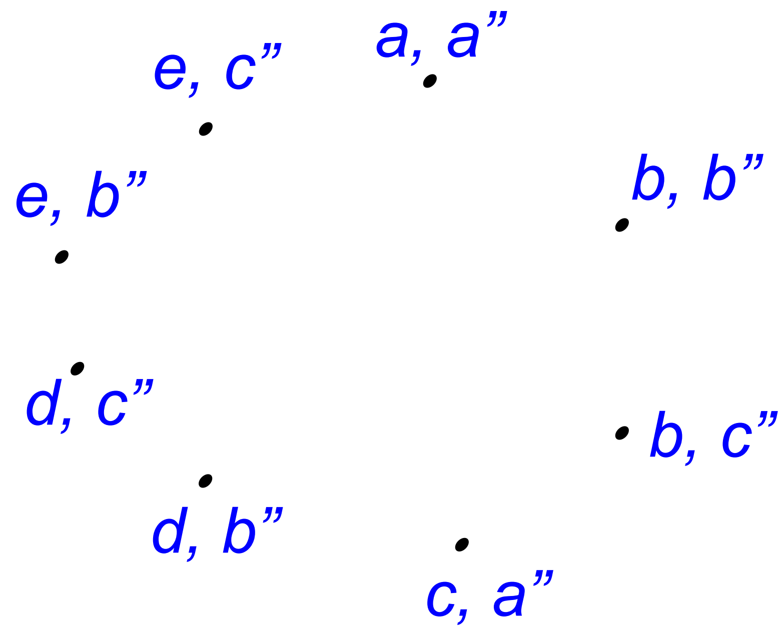
Find Matched Graph between AGl_1 and AG_u



AG_{l_1}



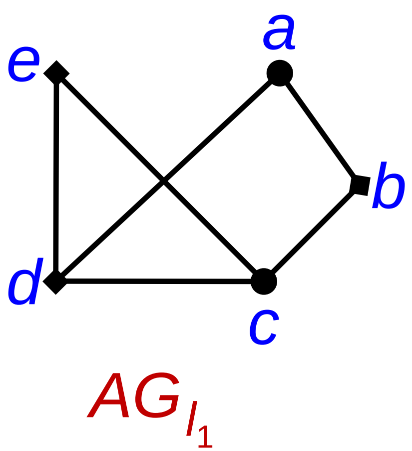
AG_u

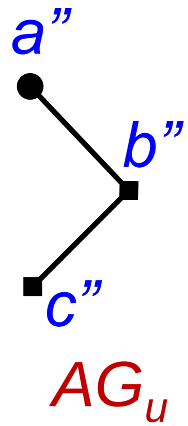


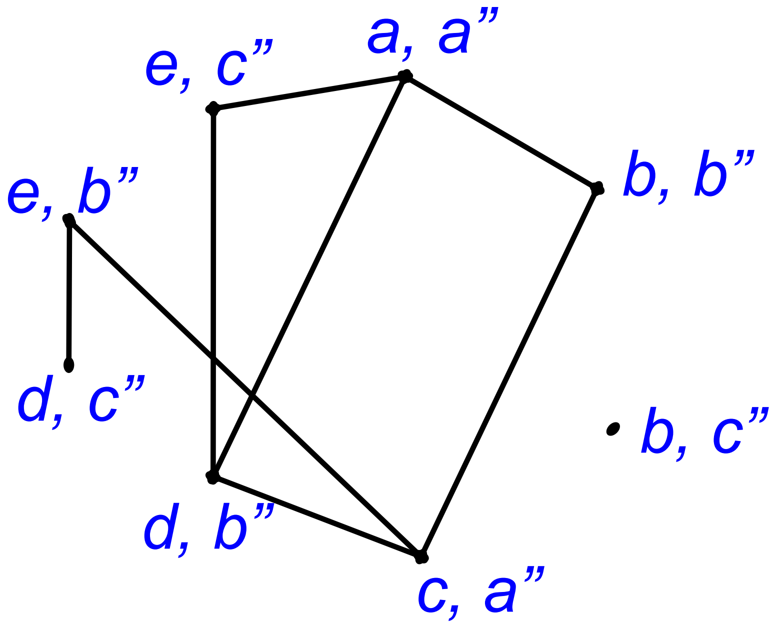
Find all assignments between AGl_1 and AG_u

Example: Pattern Matching Using Attributed Graph

Find Matched Graph between AGl_1 and AG_u



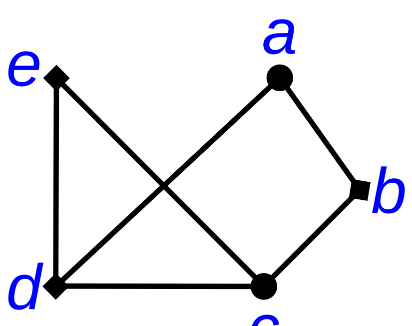




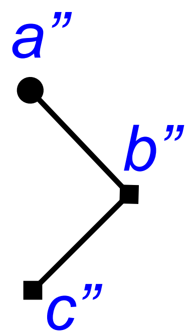
Connect the compatible assignments

Example: Pattern Matching Using Attributed Graph

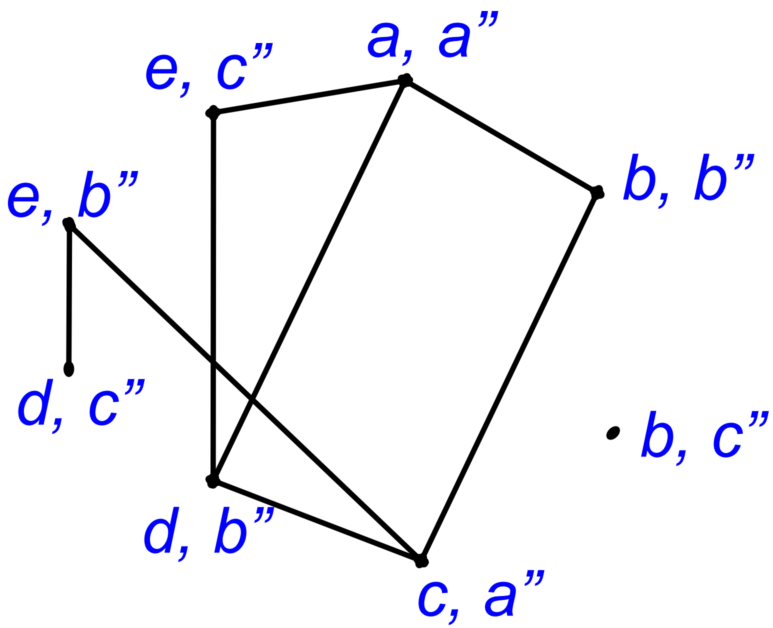
Find Matched Graph between AGl_1 and AG_u

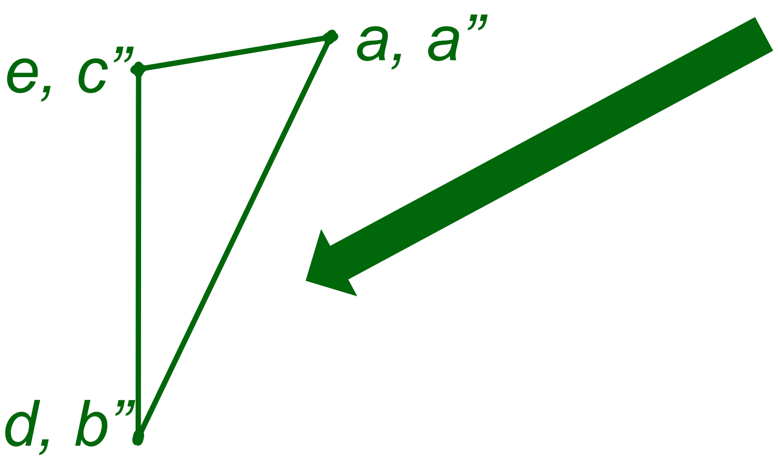


AG_{l_1}



AG_u

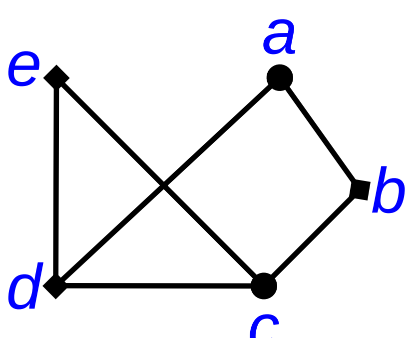




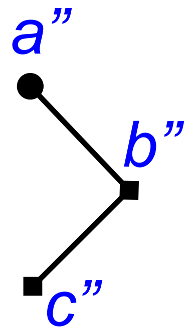
Find the maximal clique:
{(*a*, *a*”),(*d*, *b*”), (*e*, *c*”)}

Example: Pattern Matching Using Attributed Graph

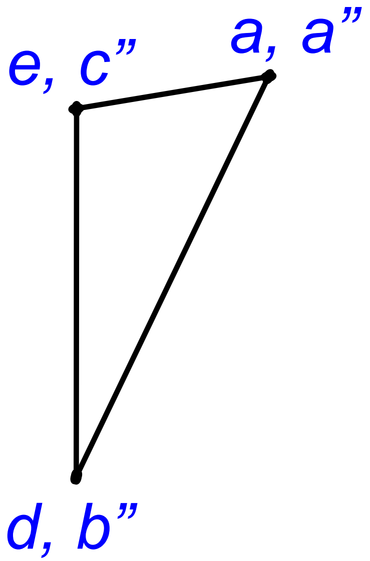
Find Matched Graph between AGl_1 and AG_u

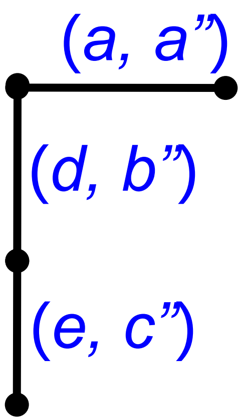


AG_{l_1}



AG_u

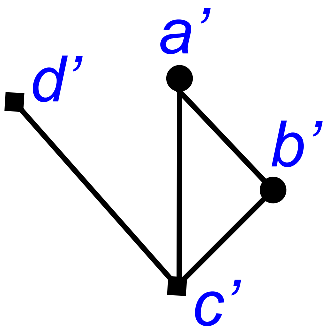




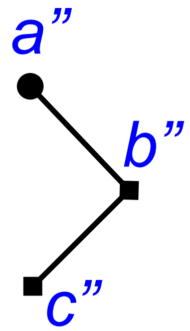
Visual Representation

Example: Pattern Matching Using Attributed Graph

Find Matched Graph between AGl_1 and AG_u



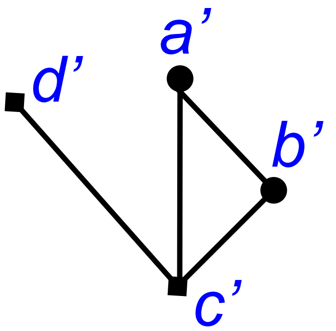
AG_{l_2}



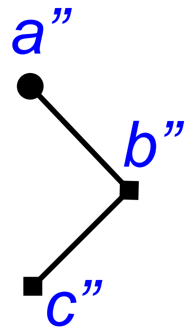
AG_u

Example: Pattern Matching Using Attributed Graph

Find Matched Graph between AGl_1 and AG_u



AG_{l_2}



AG_u

Find all assignments between AGl_2 and AG_u:

(*a*', *a*"), (*b*', *a*"), (*c*', *b*"), (*c*', *c*"), (*d*', *b*"), (*d*', *c*")