Structural and Syntactic Pattern Recognition

Selim Aksoy

Department of Computer Engineering
Bilkent University
saksoy@cs.bilkent.edu.tr

CS 551, Spring 2019

Introduction

- *Statistical pattern recognition* attempts to classify patterns based on a set of extracted features and an underlying statistical model for the generation of these patterns.
- Ideally, this is achieved with a rather straightforward procedure:
  - determine the feature vector,
  - train the system,
  - classify the patterns.
- Unfortunately, there are also many problems where patterns contain structural and relational information that are difficult or impossible to quantify in feature vector form.

Introduction

- *Structural pattern recognition* assumes that pattern structure is quantifiable and extractable so that structural similarity of patterns can be assessed.
- Typically, these approaches formulate hierarchical descriptions of complex patterns built up from simpler primitive elements.

Introduction

- This structure quantification and description are mainly done using:
  - Formal grammars,
  - Relational descriptions (principally graphs).
- Then, recognition and classification are done using:
  - Parsing (for formal grammars),
  - Relational graph matching (for relational descriptions).
- We will study strings, grammatical methods, and graph-theoretic approaches.

Recognition with Strings

- Suppose the patterns are represented as ordered sequences or *strings* of discrete items, as in a sequence of letters in a word or in DNA bases in a gene sequence.
- Pattern classification methods based on such strings of discrete symbols differ in a number of ways from the more commonly used techniques we have discussed earlier.
- Definitions:
  - String elements are called *characters* (or letters, symbols).
  - String representation of a pattern is also called a *word*.
  - A particularly long string is denoted *text*.
  - Any contiguous string that is part of another string is called a *factor* (or substring, segment) of that string.

Recognition with Strings

- Important pattern recognition problems that involve computations on strings include:
  - *String matching:* Given string $\mathbf{x}$ and text, determine whether $\mathbf{x}$ is a factor of text, and if so, where it appears.
  - *String edit distance:* Given two strings $\mathbf{x}$ and $\mathbf{y}$, compute the minimum number of basic operations — character insertions, deletions and exchanges — needed to transform $\mathbf{x}$ into $\mathbf{y}$.
  - *String matching with errors:* Given string $\mathbf{x}$ and text, find the locations in text where the "distance" of $\mathbf{x}$ to any factor of text is minimal.
  - *String matching with the "don't care" symbol:* This is the same as basic string matching but the special "don't care" symbol can match any other symbol.

String Matching

- The most fundamental and useful operation in string matching is testing whether a candidate string $\mathbf{x}$ is a factor of text.
- The number of characters in text is usually much larger than that in $\mathbf{x}$, i.e., $|\text{text}| \gg |\mathbf{x}|$, where each discrete character is taken from an *alphabet*.
- A *shift*, s, is an offset needed to align the first character of $\mathbf{x}$ with character number $s + 1$ in text.

String Matching

- The basic string matching problem is to find whether there exists a valid shift, i.e., one where there is a perfect match between each character in $\mathbf{x}$ and the corresponding one in text.
- The general string matching problem is to list all valid shifts.
- The most straightforward approach is to test each possible shift in turn.
- More sophisticated methods use heuristics to reduce the number of comparisons.

String Matching

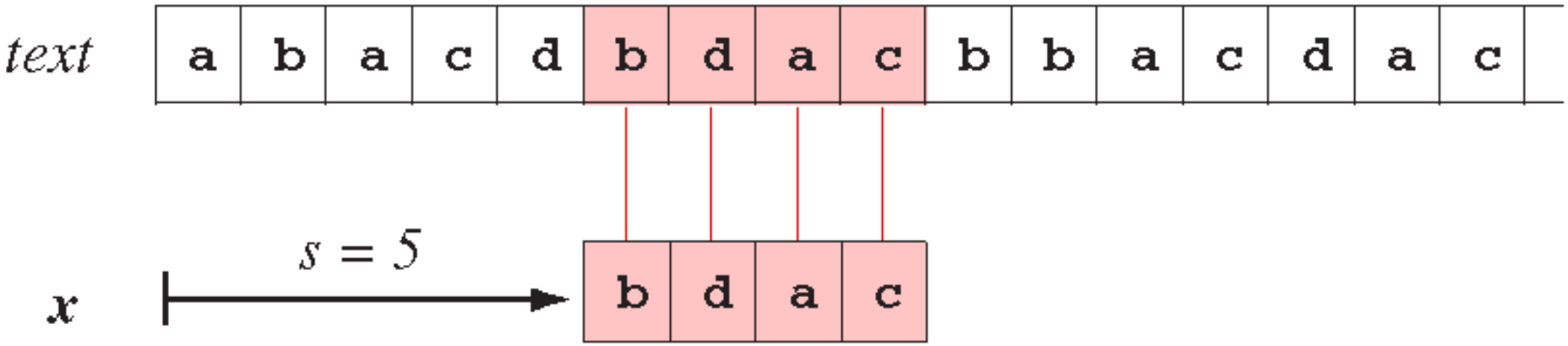


Figure 1: The general string matching problem is to find all shifts s for which the pattern x appears in text. Any such shift is called valid. In this example, $\mathrm{x} =$ "bdac" is indeed a factor of text, and $s = 5$ is the only valid shift.

String Edit Distance

- The fundamental idea underlying pattern recognition using edit distance is based on the nearest neighbor algorithm.
- We store a full training set of strings and their associated category labels.
- During classification, a test string is compared to each stored string, a “distance” is computed, and the string is assigned the category label of the nearest string in the training set.

String Edit Distance

- Edit distance between $\mathbf{x}$ and $\mathbf{y}$ describes how many of the following fundamental operations are required to transform $\mathbf{x}$ and $\mathbf{y}$.
  - *Substitutions:* A character in $\mathbf{x}$ is replaced by the corresponding character in $\mathbf{y}$.
  - *Insertions:* A character in $\mathbf{y}$ is inserted into $\mathbf{x}$, thereby increasing the length of $\mathbf{x}$ by one character.
  - *Deletions:* A character in $\mathbf{x}$ is deleted, thereby decreasing the length of $\mathbf{x}$ by one character.

String Edit Distance

$\mathbf{x}$	excused	source string
	exhused	substitute h for c
	exhaused	insert a
	exhausted	insert t
$\mathbf{y}$	exhausted	target string

Figure 2: Transformation of $\mathbf{x}$ = "excused" to $\mathbf{y}$ = "exhausted" through one substitution and two insertions.

String Matching with Errors

- Given a pattern $\mathbf{x}$ and text, string matching with errors algorithm finds the shift for which the edit distance between $\mathbf{x}$ and a factor of text is minimum.
- The algorithm for the string matching with errors problem is very similar to that for edit distance but some additional heuristics can reduce the computational burden.

String Matching with Errors

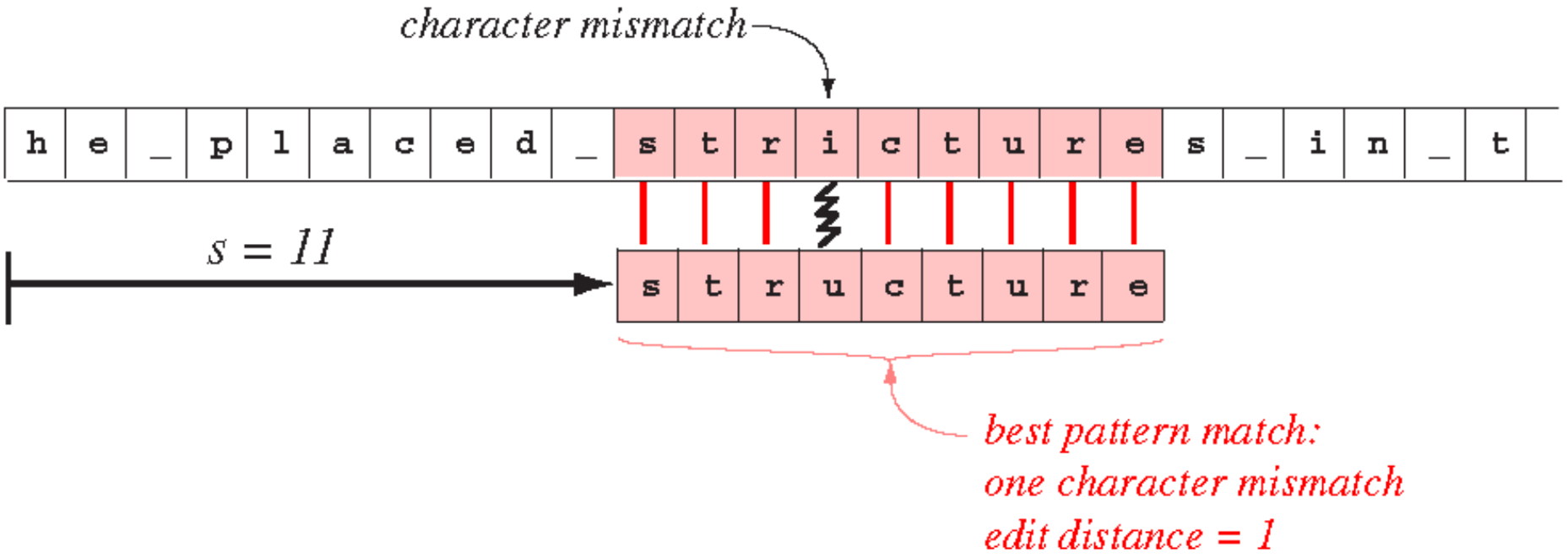


Figure 3: Finding the shift s for which the edit distance between $\mathbf{x}$ and an aligned factor of text is minimum. In this figure, the minimum edit distance is 1, corresponding to the character exchange $u \rightarrow i$, and the shift $s = 11$ is the location.

String Matching with “Don’t Care”

- String matching with the “don’t care” symbol, $\emptyset$, is formally the same as basic string matching, but the $\emptyset$ in either $\mathbf{x}$ or text is said to match any character.
- The straightforward approach is to modify the string matching algorithm to include a condition for matching the “don’t care” symbol.

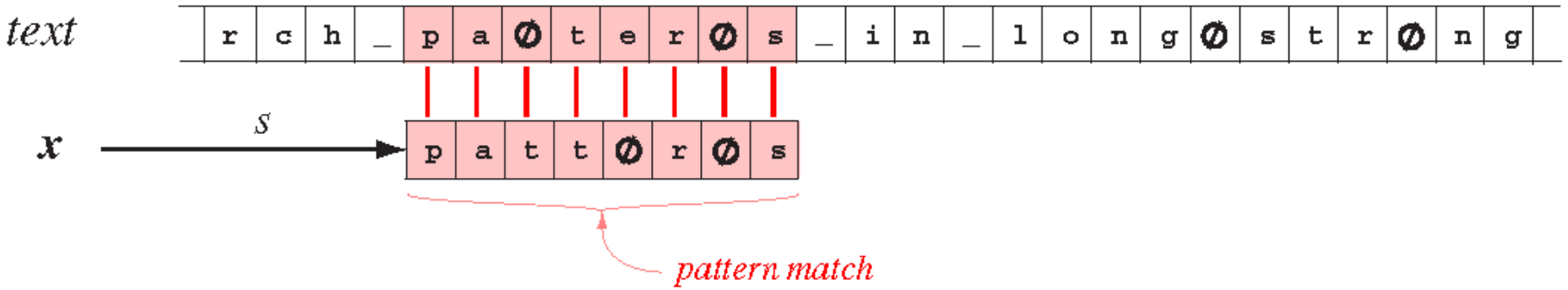


Figure 4: The problem of string matching with the “don’t care” symbol is the same as the basic string matching except that the $\emptyset$ symbol can match any character. The figure shows the only valid shift.

Grammatical Methods

- Grammars provide detailed models that underlie the generation of the sequence of characters in strings.
- For example, strings representing telephone numbers conform to a strict structure.
- Similarly, optical character recognition systems that recognize and interpret mathematical equations can use rules that constrain the arrangement of the symbols.
- In pattern recognition, we are given a *sentence* (a string generated by a set of rules) and a *grammar* (the set of rules), and seek to determine whether the sentence was generated by this grammar.

Grammatical Methods

- Formally, a grammar consists of four components:
  - *Symbols:* Every sentence consists of a string of characters (or primitive symbols, terminal symbols) from an alphabet.
  - *Variables:* These are called the nonterminal symbols (or intermediate symbols, internal symbols).
  - *Root symbol:* It is a special variable, the source from which all sequences are derived.
  - *Productions:* The set of production rules (or rewrite rules) specify how to transform a set of variables and symbols into other variables and symbols.
- For example, if A is a variable and c a terminal symbol, the rewrite rule $cA \rightarrow cc$ means that any time the segment cA appears in a string, it can be replaced by cc.

Grammatical Methods

- The *language* $\mathcal{L}(G)$ generated by a grammar G is the set of all strings (possibly infinite in number) that can be generated by G.

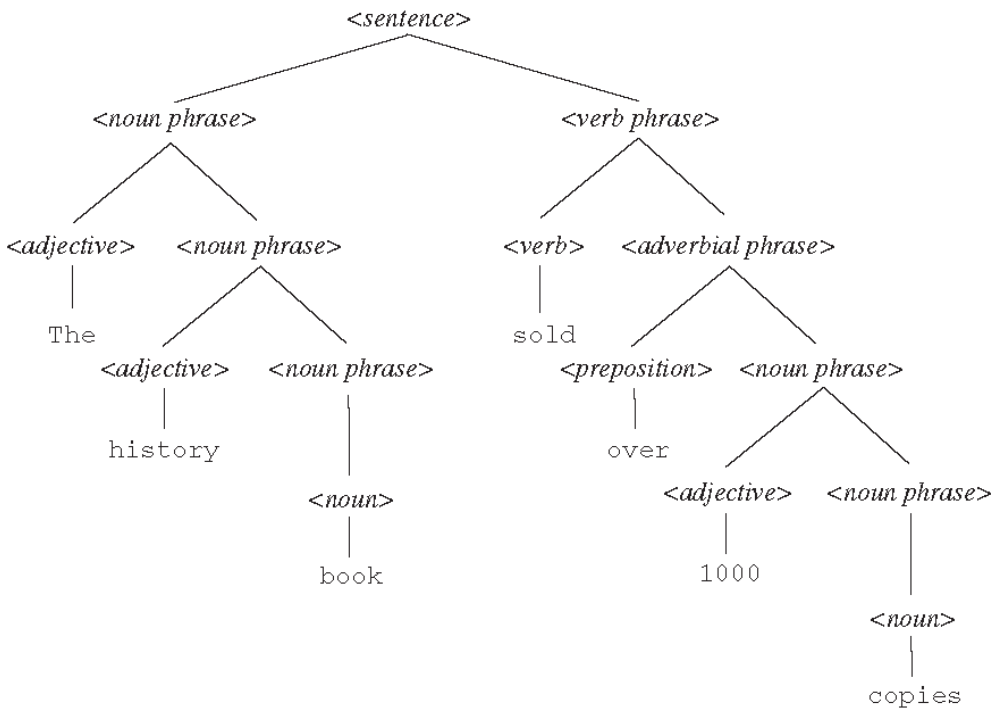


Figure 5: The derivation tree illustrates how a portion of English grammar can transform the root symbol into a particular sentence.

Types of String Grammars

- *Type 0: Free or Unrestricted:* Free grammars have no restrictions on the rewrite rules, and thus they provide no constraints or structure on the strings they can produce.
- *Type 1: Context-Sensitive:* A grammar is called context-sensitive if every rewrite rule is of the form

$$\alpha I \beta \rightarrow \alpha x \beta$$

where α and β are any strings made up of intermediate and terminal symbols, I is an intermediate symbol, and x is an intermediate or terminal symbol, i.e., I can be rewritten as x in the context of α on the left and β on the right.

Types of String Grammars

- *Type 2: Context-Free:* A grammar is called context-free if every production rule is of the form

$$I \rightarrow x$$

where I is an intermediate symbol, and x is an intermediate or terminal symbol, i.e., there is no need for a context for the rewriting of I by x.

Types of String Grammars

- *Type 3: Finite State or Regular:* A grammar is called regular if every rewrite rule is of the form

$$\alpha \rightarrow z\beta \qquad \text{or} \qquad \alpha \rightarrow z$$

  where α and β are made up of intermediate symbols and z is a terminal symbol.
- The class of grammars of type i includes all grammars of type $i + 1$.

Recognition Using Grammars

- Suppose we are given a test sentence $\mathbf{x}$ that was generated by one of c different grammars $G_1, G_2, \ldots, G_c$ which can be considered as different models or classes.
- The test sentence $\mathbf{x}$ is classified according to which grammar could have produced it, or equivalently, the language $\mathcal{L}(G_i)$ of which $\mathbf{x}$ is a member.
- *Parsing* is the inverse process that, given a particular $\mathbf{x}$, finds a derivation in G that leads to $\mathbf{x}$.

Recognition Using Grammars

- *Bottom-up parsing* starts with the test sentence $\mathbf{x}$, and seeks to simplify it, so as to represent it as the root symbol.
- The basic approach is to use candidate productions backwards, i.e., find rewrite rules whose right hand side matches part of the current string, and replace that part with a segment that could have produced it.

Recognition Using Grammars

- *Top-down parsing* starts with the root node and successively applies productions with the goal of finding a derivation of the test sentence $\mathbf{x}$.
- Since it is rare that the sentence is derived in the first production attempted, it is necessary to specify some criteria to guide the choice of which rewrite rule to apply.

Pattern Description Using Grammars

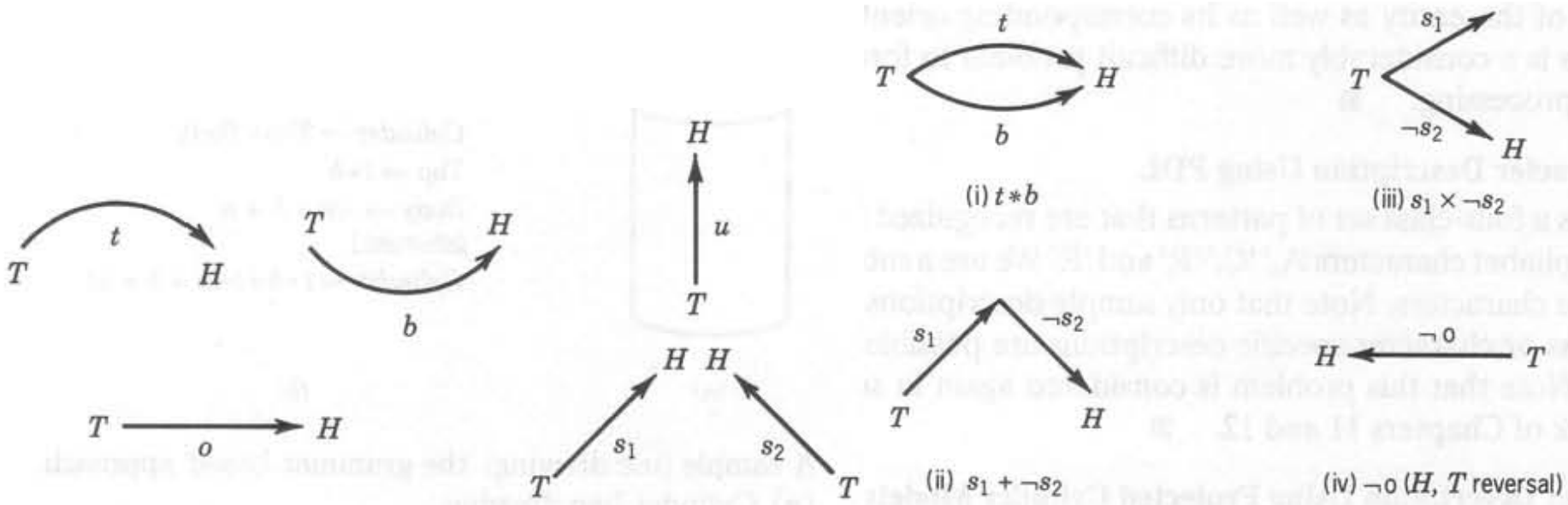
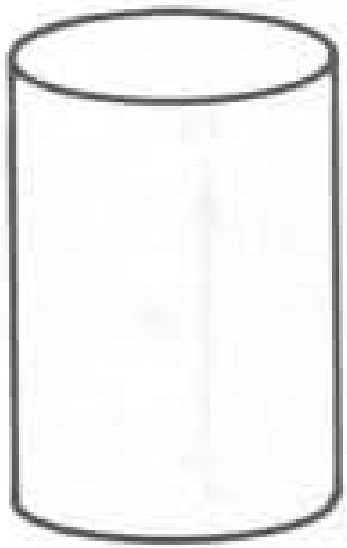


Figure 6: A 2-D line drawing picture description grammar with the set of terminal symbols $\{t, b, u, o, s, *, \neg, +\}$ where $+$ represents head to tail concatenation, $*$ represents head-head and tail-tail attachment, and $\neg$ represents head and tail reversal. H represents heads of lines and T represents the tails. (Schalkoff, Pattern Recognition: Statistical, Structural and Neural Approaches, 1992)

Pattern Description Using Grammars



$$Cylinder \rightarrow Top * Body$$
$$Top \rightarrow t * b$$
$$Body \rightarrow \neg u + b + u$$
(alternate:)
$$Cylinder \rightarrow t * b * (\neg u + b + u)$$

Figure 7: Representation of a cylinder using the line drawing picture description grammar.

Pattern Description Using Grammars

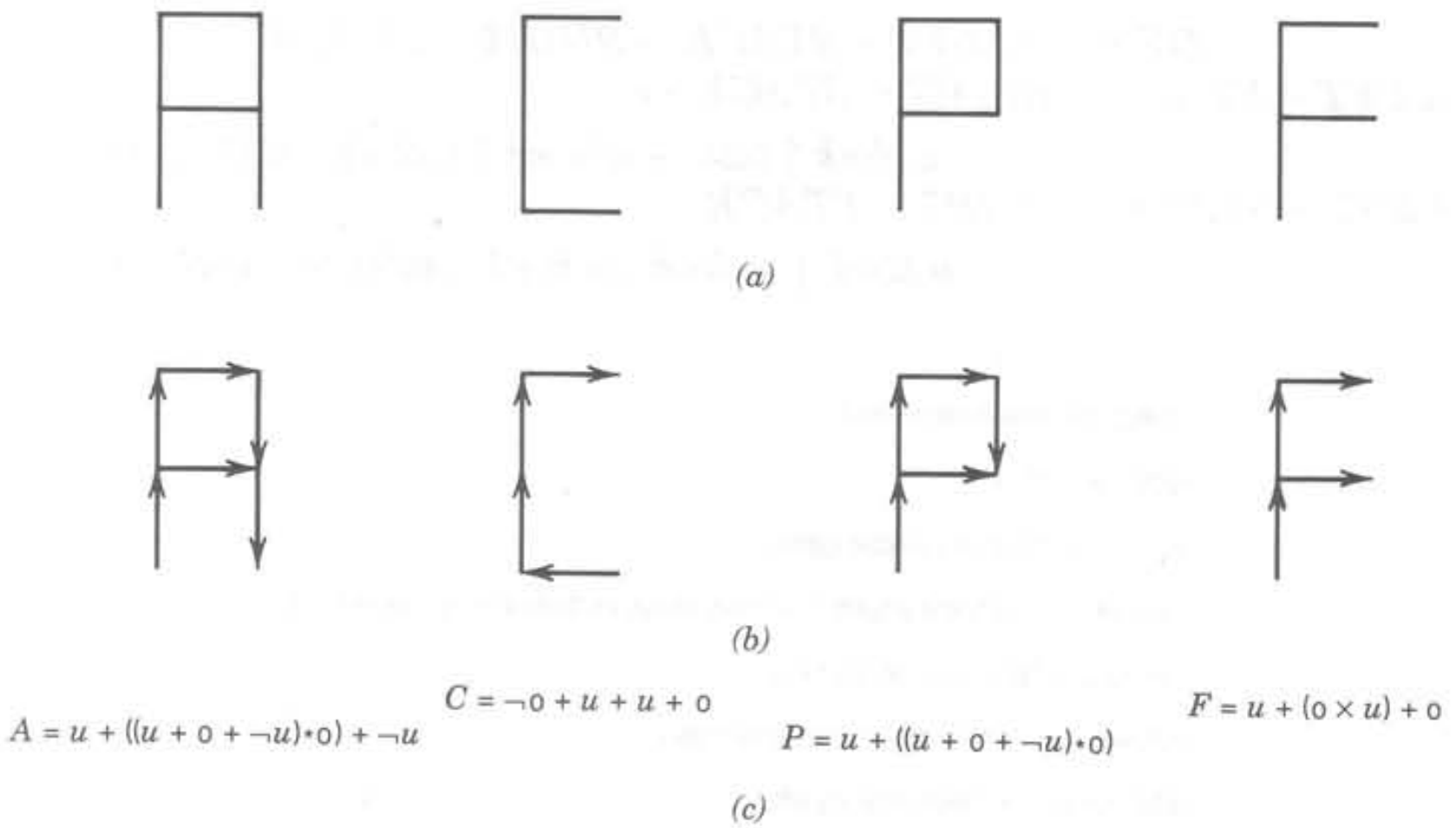


Figure 8: Representation of four characters using the line drawing picture description grammar. (a) Pattern data. (b) Primitive representation and interconnection. (c) Corresponding descriptions.

Pattern Description Using Grammars

- A grammar describing four blocks arranged in 2-block stacks:

$$V_T = \{table, block, +, \uparrow\} \qquad \text{(terminal symbols)}$$

$$V_N = \{DESC, LEFT\_STACK, RIGHT\_STACK\}$$

(non-terminal symbols)

$$S = DESC \in V_N \qquad \text{(root symbol)}$$

$$P = \{DESC \rightarrow LEFT\_STACK + RIGHT\_STACK$$

$$DESC \rightarrow RIGHT\_STACK + LEFT\_STACK$$

$$LEFT\_STACK \rightarrow block \uparrow block \uparrow table$$

$$RIGHT\_STACK \rightarrow block \uparrow block \uparrow table\}$$

(production rules)

Pattern Description Using Grammars

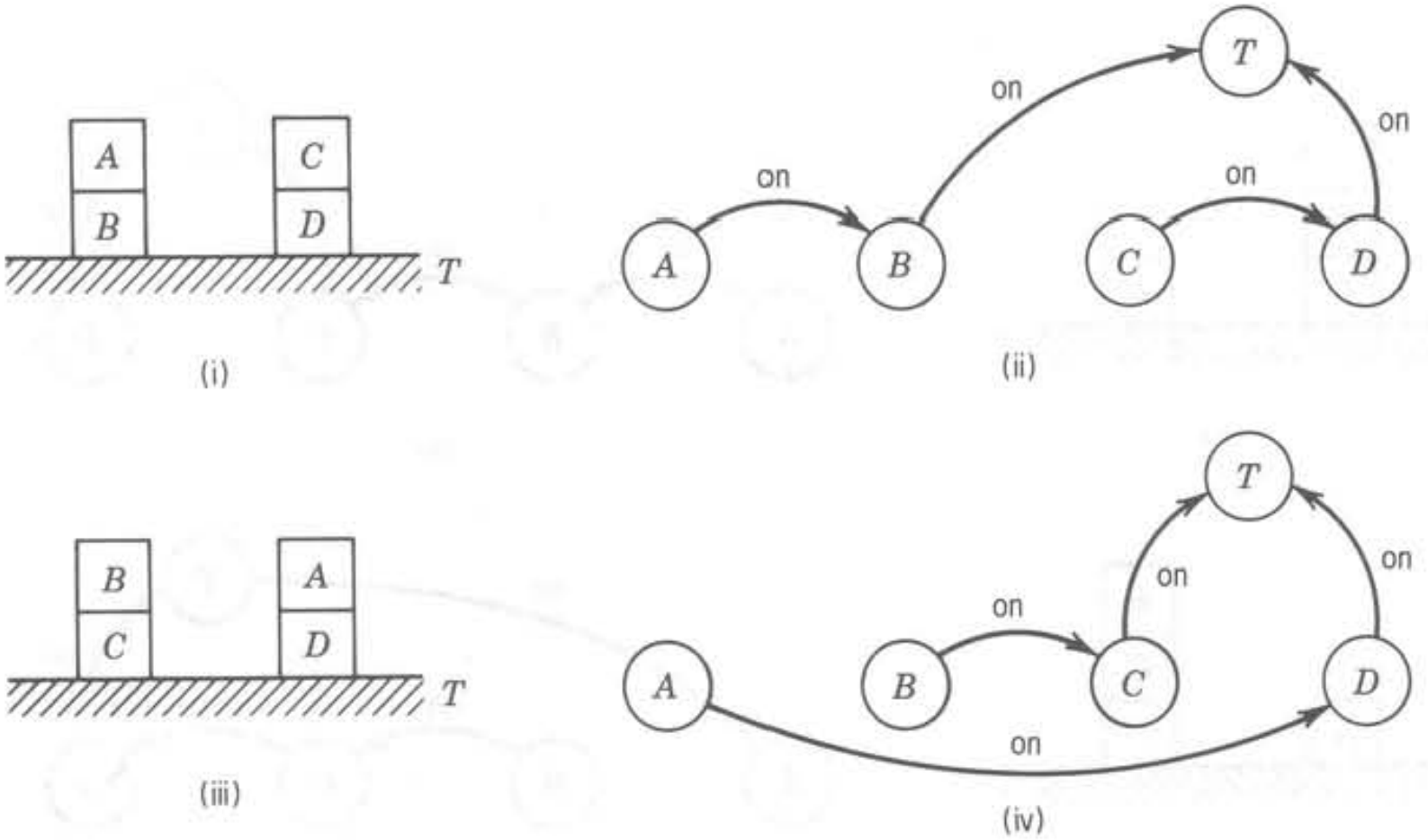


Figure 9: A grammar describing four blocks arranged in 2-block stacks. (i) An example. (ii) Graphical description corresponding to (i). (iii) Another example. (iv) Graphical description corresponding to (iii).

Pattern Description Using Grammars

- A grammar describing four blocks in 3-block and 1-block stacks:

$$V_T = \{table, block, +, \uparrow\} \qquad \text{(terminal symbols)}$$

$$V_N = \{DESC, LEFT\_STACK, RIGHT\_STACK\}$$

$$\text{(non-terminal symbols)}$$

$$S = DESC \in V_N \qquad \text{(root symbol)}$$

$$P = \{DESC \rightarrow LEFT\_STACK + RIGHT\_STACK$$

$$LEFT\_STACK + RIGHT\_STACK \rightarrow$$

$$block \uparrow table + block \uparrow block \uparrow block \uparrow table$$

$$LEFT\_STACK + RIGHT\_STACK \rightarrow$$

$$block \uparrow block \uparrow block \uparrow table + block \uparrow table\}$$

Pattern Description Using Grammars

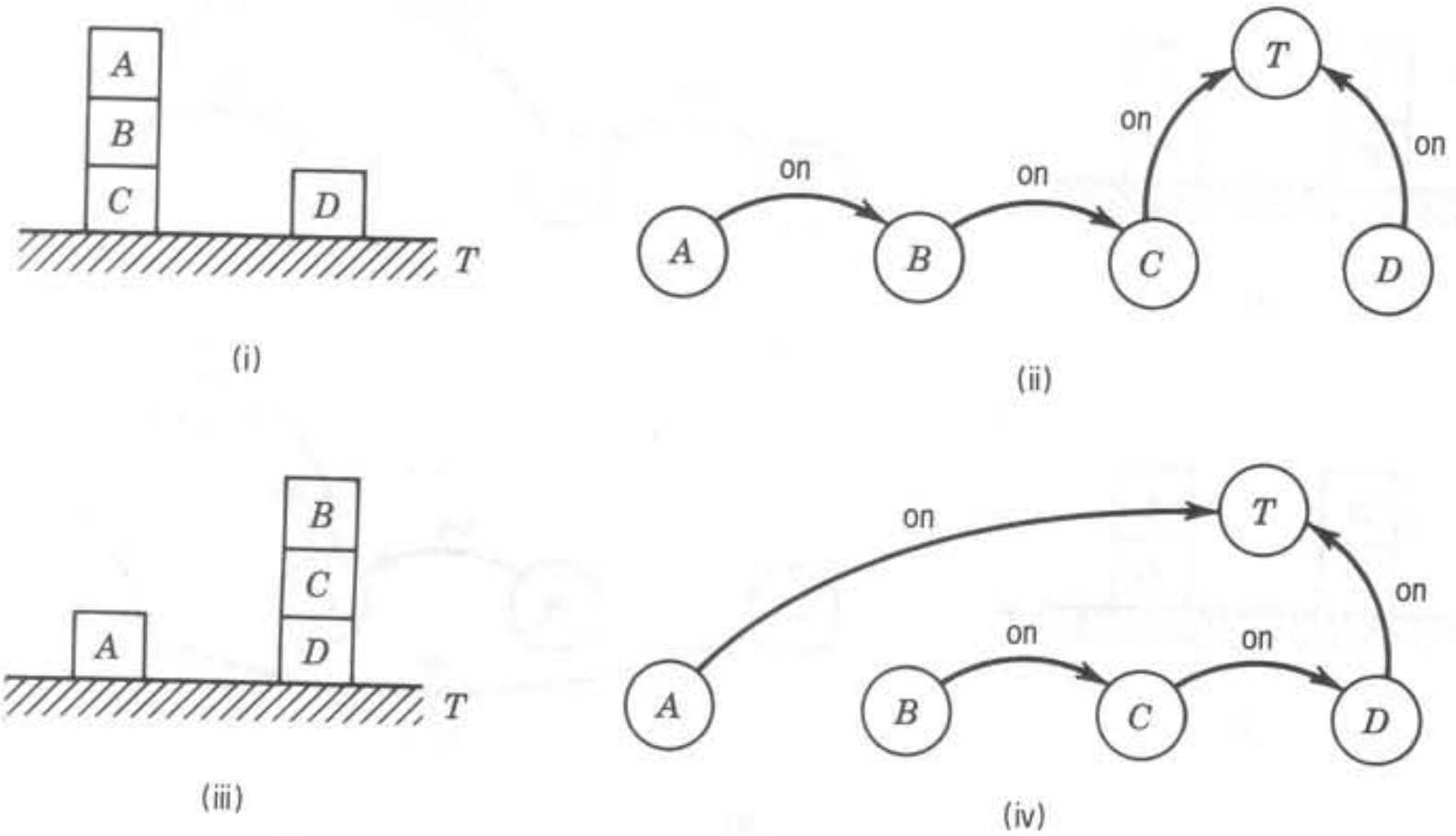


Figure 10: A grammar describing four blocks arranged in 3-block and 1-block stacks. (i) An example. (ii) Graphical description corresponding to (i). (iii) Another example. (iv) Graphical description corresponding to (iii).

Pattern Description Using Grammars

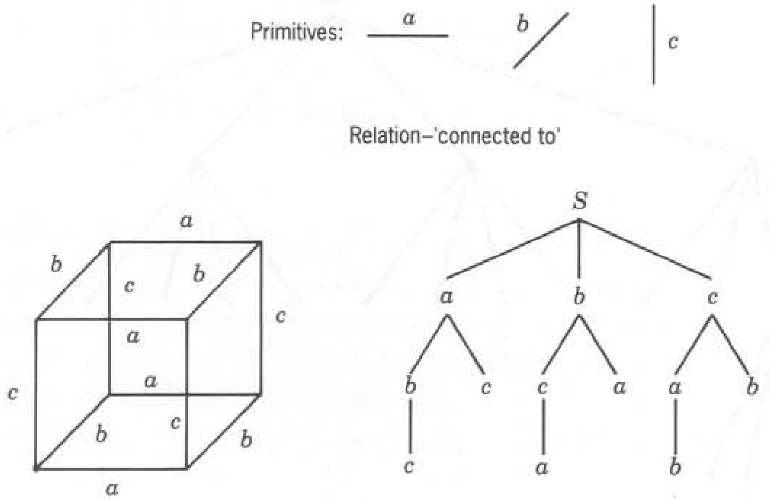


Figure 11: Tree grammar-based representation of a cube.

Pattern Description Using Grammars

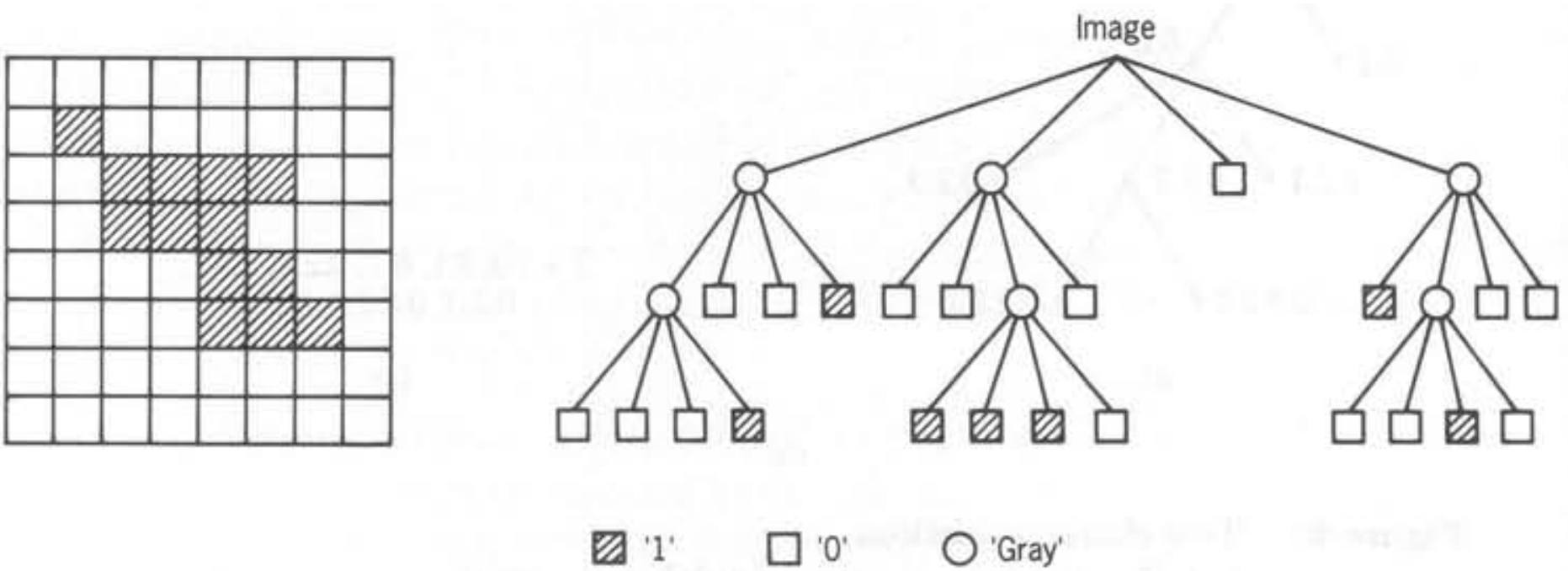


Figure 12: Quadtree representation of an 8×8 binary image.

Graph-Theoretic Methods — Definitions

- Graphical alternatives for structural representations are natural extensions of higher dimensional grammars because graphs are valuable tools for representing relational information.
- A *graph* $G = \{N, R\}$ is an ordered pair represented using:
  - a set of *nodes* (vertices), N,
  - a set of *edges* (arcs), $R \subseteq N \times N$.
- A *subgraph* of G is itself a graph $G_s = \{N_s, R_s\}$ where $N_s \subseteq N$ and R_s consists of edges in R that connect only the nodes in N_s.

Graph-Theoretic Methods — Definitions

- 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.
- A *relation* from set A to set B is a subset of $A \times B$.
- It is usually shown using a function $f : A \rightarrow B$ or $b = f(a)$.

Graph-Theoretic Methods — Definitions

- For example, the relation “lies on” can contain:

$$R = \{(\text{floor}, \text{foundation}), (\text{rug}, \text{floor}), (\text{chair}, \text{rug}), (\text{person}, \text{chair})\}.$$

- Note that relations have directions, i.e., the order in which an entity appears in the pair is significant.
- Higher-order relations can be shown as ordered n-tuples that can also be viewed as ordered pairs of an $(n-1)$-tuple and a single element, e.g., $(((A \times B) \times C) \times D)$.

Graph-Theoretic Methods — Definitions

- 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*.

Graph-Theoretic Methods — Definitions

- A *relational graph* represents a particular relation graphically using arrows to show this relation between the elements as a directed graph.
- A *tree* is a finite acyclic (containing no closed loops or paths or cycles) digraph.

Comparing Relational Graph Descriptions

- One way to recognize structure using graphs is to let each pattern structural class be represented by a prototypical relational graph.
- An unknown input pattern is then converted into a structural representation in the form of a graph, and this graph is then compared with the relational graphs for each class.

Comparing Relational Graph Descriptions

- The observed data rarely matches a stored relational representation “exactly”, hence, *graph similarity* should be measured.
- 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.

Graph Isomorphism

- 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

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

- 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:

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

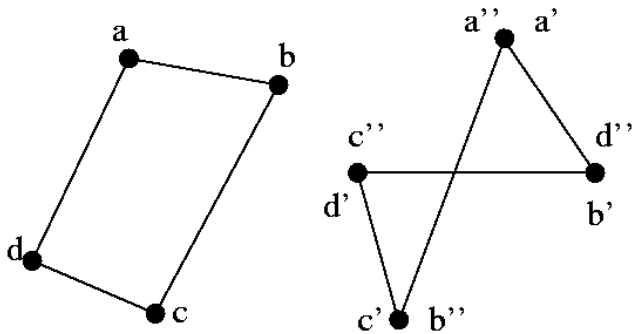
Graph Isomorphism

- 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.
- 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



	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

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

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

Figure 13: An example of isomorphism of two undirected graphs with $p = 4$.

Graph Isomorphism

- Unfortunately, determining graph isomorphism is computationally expensive (NP complete).
- Furthermore, it is also not a practical similarity measure because it allows only exact matches but not all existing relations for a given class are observed in practical problems.
- 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.

Extensions to Graph Matching

- 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.

Extensions to Graph Matching

- Common transformations include:
  - Node insertion,
  - Node deletion,
  - Node splitting,
  - Node merging,
  - Edge insertion,
  - Edge deletion.
- Note that computational complexity can still be high and it may be difficult to design a distance measure that can distinguish structural deformations between different classes.

Attributed Relational Graphs

- In addition to representing pattern structure, the representation may be extended to include numerical and symbolic attributes of pattern primitives.
- 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.

Attributed Relational Graphs

- Let $p_q^i(n)$ denote the value of the q'th property of node n of graph G_i.
- Nodes $n_i \in N_i$ and $n_2 \in N_2$ are said to form an *agreement* (n_1, n_2) if

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

where "$\sim$" denotes similarity.

Attributed Relational Graphs

- 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.

Comparing Attributed Graph Descriptions

- A strategy for measuring the similarity between two attributed graphs is to find node pairings using the cliques of a match graph.
- A *clique* of a graph is a totally connected subgraph.
- A *maximal clique* is not included in any other clique.
- 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.
- The maximal cliques of the match graph provide starting points for good candidate node pairings between two graphs.

Comparing Attributed Graph Descriptions

- Another similarity measure between two attributed graphs is the *editing distance* which is defined as the minimum cost taken over all sequences of operations (error corrections) that transform one graph to the other.
- These operations are defined as substitution, insertion and deletion.

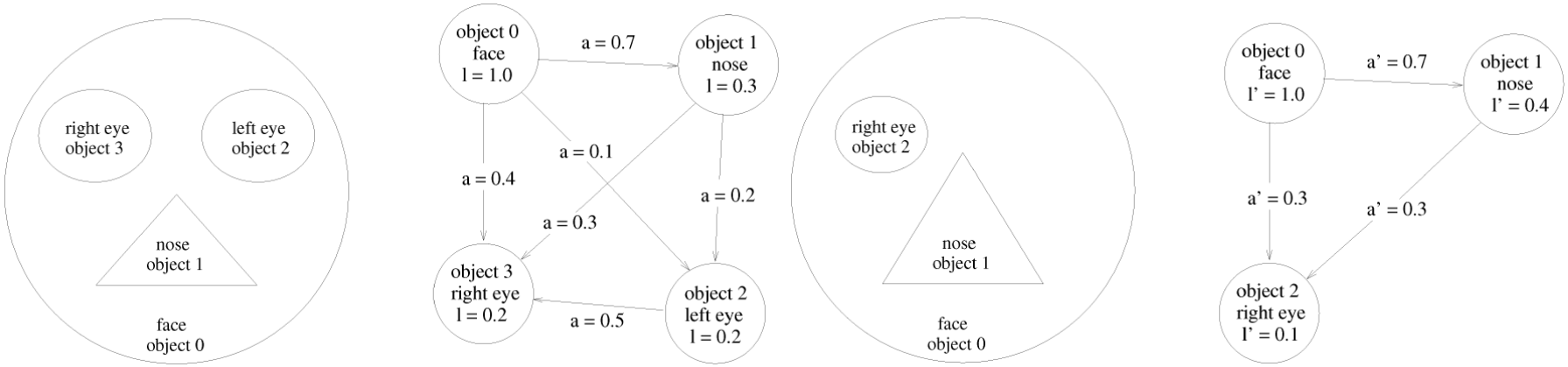
Comparing Attributed Graph Descriptions

- Let G_1 and G_2 be two graphs where each node and edge are assigned labels with an additional confidence value for each label.
- Let f_1 be the confidence value of the label of a node in the first graph and f_2 be the confidence value of a node with the same label in the second graph.
- The cost of node substitution is $|f_1 - f_2|$ and the cost of node insertion or deletion is f_1.

Comparing Attributed Graph Descriptions

- Let g_1 be the confidence value of the label of an edge in the first graph and g_2 be the corresponding value for an edge in the second graph.
- The cost of edge substitution is $|g_1 - g_2|$ and the cost of edge insertion or deletion is g_1.
- The computation of the distance between two attributed graphs involves not only finding a sequence of error corrections that transforms one graph to the other, but also finding the one that yields the minimum total cost.

Comparing Attributed Graph Descriptions



The editing distance between these two graphs is computed as:

- node substitution:
  - object 0 in (b) with object 0 in (a) with cost $|1.0 - 1.0| = 0$
  - object 1 in (b) with object 1 in (a) with cost $|0.4 - 0.3| = 0.1$
  - object 2 in (b) with object 3 in (a) with cost $|0.1 - 0.2| = 0.1$
- edge substitution for these nodes:
  - $|0.7 - 0.7| = 0$, $|0.3 - 0.4| = 0.1$, $|0.3 - 0.3| = 0$
- node deletion: object 2 in (a) with cost 0.2
- edge deletion for this node: 0.5, 0.2, 0.1

Total cost of matching is $0.1 + 0.1 + 0.1 + 0.2 + 0.5 + 0.2 + 0.1 = 1.3$.

(Taken from Petrakis et al. "ImageMap: An Image Indexing Method Based on Spatial Similarity," IEEE Trans. on Knowledge and Data Engineering, 14(5):979–987, 2002.)

Comparing Attributed Graph Descriptions

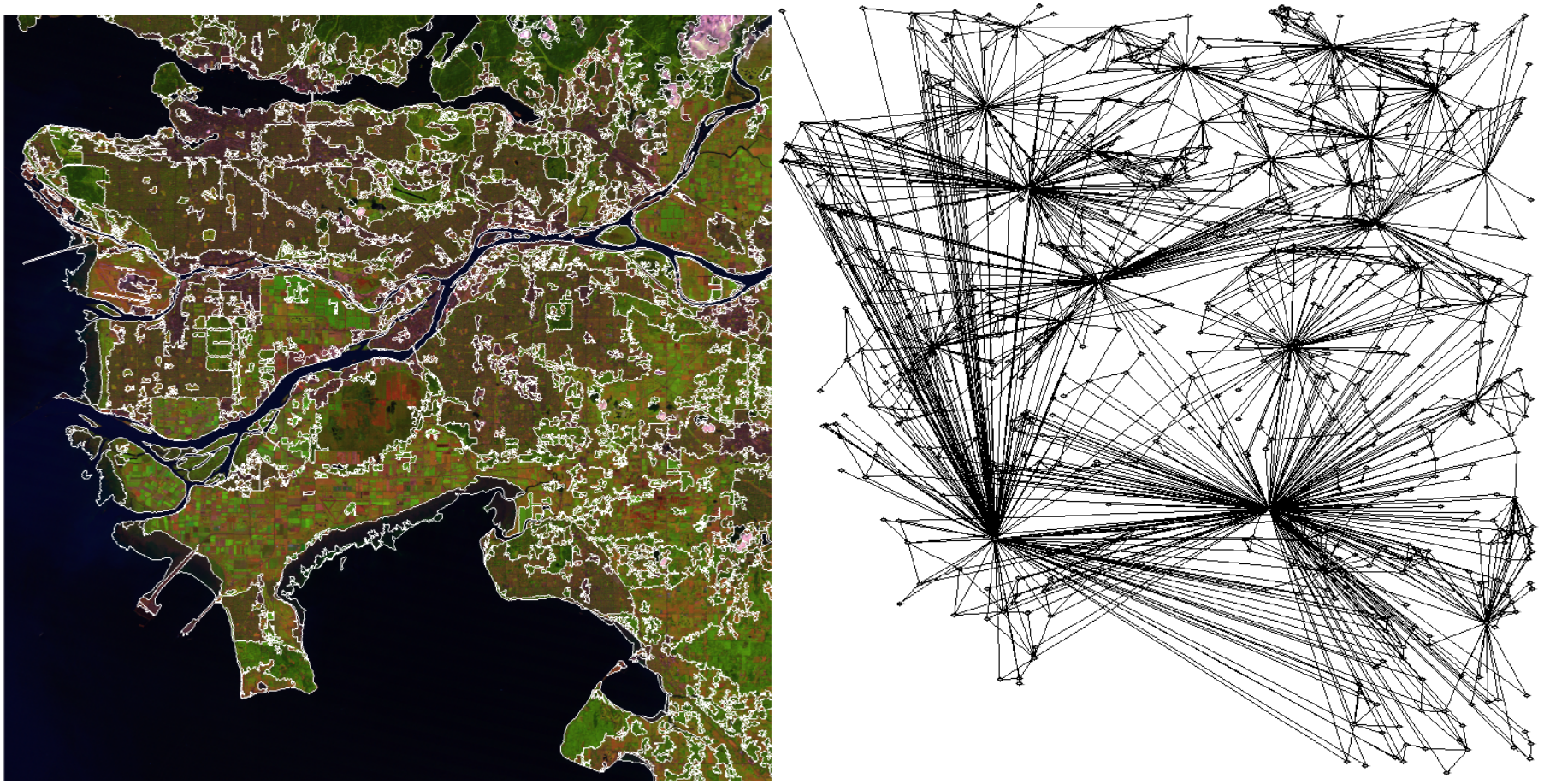
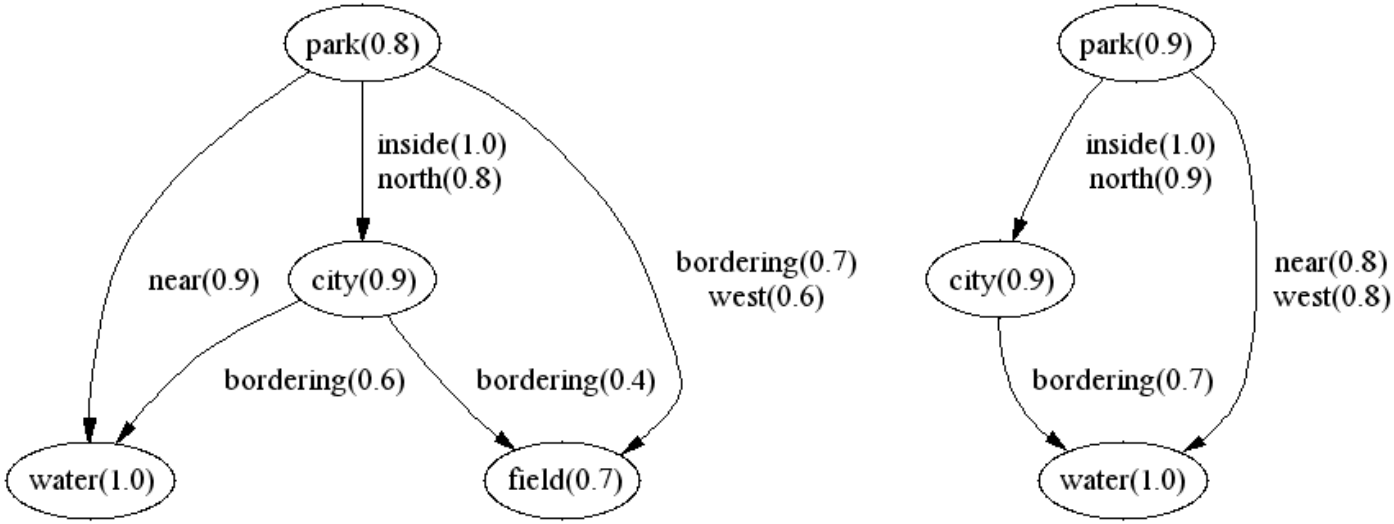


Figure 14: An example image scene and its attributed graph. Nodes correspond to image regions marked with white boundaries and edges correspond to the spatial relationships between these regions.

Comparing Attributed Graph Descriptions

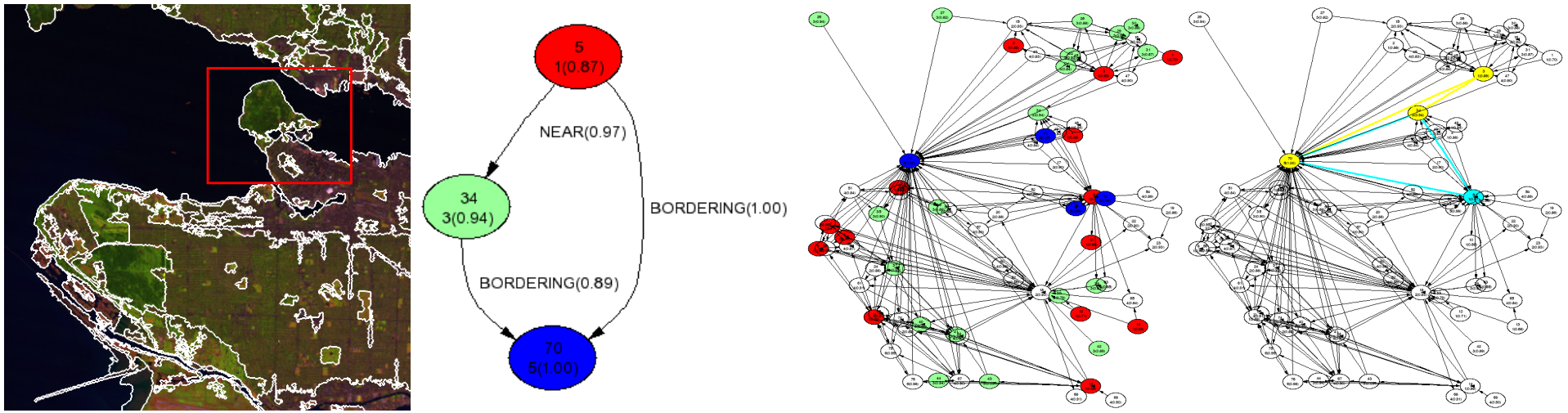


The editing distance between these two graphs is computed as:

- node substitution:
  - park in (a) with park in (b) with cost $|0.8 - 0.9| = 0.1$
  - city in (a) with city in (b) with cost $|0.9 - 0.9| = 0$
  - water in (a) with water in (b) with cost $|1.0 - 1.0| = 0$
- edge substitution for these nodes:
  - $|0.9 - 0.8| + 0.8 = 0.9$, $|1.0 - 1.0| + |0.8 - 0.9| = 0.1$, $|0.6 - 0.7| = 0.1$,
- node insertion: field in (a) with cost 0.7
- edge insertion for this node: 0.4, 0.7, 0.6

Total cost of matching is 3.6.

Comparing Attributed Graph Descriptions



(a) (b) (c) (d)

Figure 15: Scene matching using attributed graphs. (a) Query scene marked using the red rectangle. (b) Query graph: red is city, green is park, blue is water. (c) Nodes with labels similar to those in the query. (d) Subgraphs matching to the query.