Retrieval by Spatial Configuration

Xin Chen[a], C. P. Lam[b] and W. F. Smyth[c d]

[a]IBM Toronto Laboratory,
8200 Warden Avenue, Markham, Ontario
Canada L6G 1C7
email: axchen@ca.ibm.com

[b]School of Computer and Information Science,
Edith Cowan University, 2 Bradford Street,
Mount Lawley, Western Australia 6050
email: c.lam@ecu.edu.au

[c]Algorithms Research Group,
Department of Computing & Software,
McMaster University, Hamilton, Ontario
Canada L8S 4K1
email: smyth@mcmaster.ca

[d]School of Computing, Curtin University,
Bentley, Western Australia 6102
email: smyth@computing.edu.au

These pages provide you with an example of the layout and style which we wish you to adopt during the preparation of your paper. Your text will be photographically reduced by 20–25%. This is the output from the LaTeX document class you requested.

1. Introduction

Large collections of digital images are being created in all sectors of life as digital information can be distributed easily and cheaply. Increasingly, it is vital to be able to retrieve digital images from collections as many more application areas such as the miltary, geographical imformation systems, entertainment, education and biomedicine are storing information digitally. The traditional approach to searching these collections involved either browsing or keyword indexing. Both of these two methods are labour-intensive, especially for a large collection. Keyword-based image retrieval systems also suffer from problems such as the appropriateness of keywords for annotation which can be non-unique and subjective. Different people can index the same image using different keywords, thus making the retrieval process difficult and imprecise. Content-Based Image Retrieval (CBIR) attempts to retrieve images from an image database by directly matching the contents of a query image with the content of images stored in a database. In designing a CBIR system, key considerations include the content of the image (image features) and its representation, the user interface for the system, and the indexing schemes to be used to allow quick retrieval.

Three factors characterise most current approaches to CBIR in terms of the content of images: low-level image features, level of abstraction and domain dependence. Most of the associated techniques employ image-processing techniques to extract semantic features within the images. Image features used for CBIR fall into two categories: low-level features and high-level features which are often calculated using the low-level features. Low-level features that have been commonly used include color, texture, edges/local orientation, wavelet transform [?] and high-level features may involve curvature, shape [?], spatial locations of objects or regions [?] as well as spatial relationships between objects/regions [?]. Some of the high-level features are also domain-specific: the representation is designed for a specific domain and often requires human intervention. Examples of the use of domain-specific high-level features include faces as in Photobook [?], trademarks [?] and attributed relational graphs [?]. The limitations associated with these approaches are due to the lack of reliable feature-extraction techniques as well as the appropriate level of abstraction to represent the semantic content of the images (i.e., which kind of features to extract and the integration of extracted features to address the user's queries). The issue here relates to use of low-level visual features extracted from images to represent the high-level semantic concepts used by humans.

Two recent surveys on CBIR systems demonstrated that most existing systems predominantly use colour, texture and shape for representing image content and also use these for indexing and retrieval. In the study carried out by Johansson [?], a total of 21 commercial and research prototype CBIR systems were examined and only three (Blobworld, Virage and VisualsEEk) support queries involving spatial relationships (defined in this study as "relative locations between objects or regions"). Example of a query could be : find images where object A is located below and to the right of object P. In a second study [?], 39 CBIR systems were examined, out of which 12 were also covered in the Johansson study. In this study carried out by Veltkamp and Tanase, spatial layout was defined to be "absolute or relative position of colour, texture or shape information" [?]. Only 12 out of the 39 surveyed systems used spatial layout. Most of surveyed systems (31 out of 39) involved the use of some kind of colour features, 25 involved the use of texture and 23 employed shape features [?]. In combining the two surveys, 48 different CBIR systems were evaluated and only about 8 of these support queries relating the spatial configurations of objects/regions in the images. As demonstrated by these two studies, very few of the current CBIR systems attempts to incorporate retrieval based on spatial relationhips between a number of objects (i.e., find images with objects in the spatial configuration indicated in the example image). In addition, existing work involving spatial relationships usually incorporates the identified pairwise spatial relations between two objects into the indexing structure which is subsequently used for retrieval. Petrakis [?] uses the properties of each object as part of the label of a corresponding node in a graph and the pairwise relations become part of the label of an edge in the graph joining two nodes. Thus each image turns into a graph whose number of nodes is equal to the number of identified objects in the image. Lee *et al* [?] computes a signature for each image in the image database using an identifier associated with the objects and the pairwise spatial relations between the identified objects. The signature is used subsequently to aid the retrieval of possible matches.

The speed of retrieval for an image database is also important and this process can be extremely slow if the database involved is large and the approach used is sequential scanning. Indexing methods can be applied to speed up this process. From the survey of 39 CBIR systems [?], it was found that about half of them support indexing. Typically, stored images in the database are characterised by fixed-length multidimensional image feature vectors. Searching and retrieval involve comparing the similarity between the feature vectors of the query image with those in the database. Most of these approaches also involve the use of relational databases or a tree indexing structure such as R*-Tree [?], B-Tree [?], SS-Tree [?] or kD-Tree [?]. The performance of most of these approaches degenerates dramatically when the dimensionality of their feature space increases. Other approaches such as the iconic index tree [?] have been employed in CAFIIR to speed up the search on images with faces. This approach is based on using the SOM to simplify the multidimensional feature space to a one dimensional problem. The limitations here mainly relate to the static nature of the index which is constructed when the system is trained using SOM and a large data set. Retraining is required when the content of the image database changes as is in the case of deletion/addition of a class of images. The issue of the evaluation of speed of retrieval has not been addressed in most of the existing CBIR systems where performance has mainly been evaluated in terms of recall and precision [?]

Although many of the systems and techniques discussed above are effective in various application domains, they are not very effective when it comes to retrieval of images in topological image databases where images (aerial and satellite) are very similar in terms of their low-level image content. The most distinguishing characteristics associated with these images are the spatial configuration of the objects within the images and the shapes of these objects. Some existing work on these kinds of images has involved shape retrieval [?].

This paper describes an approach to image retrieval that is based solely on the spatial configuration of the objects or regions in the image. The algorithms retrieve all those "text" images that contain configurations identical to the configuration that occurs in the "pattern" image. The method treats all objects as equivalent and so does not depend on any of their other features; thus the nature of the object does not need to be specified by human intervention. Further, although the method is described in terms of three "classical" approaches to characterizing spatial relationships between pairs of objects, it is not restricted to them: it can be used with *any* system that allows the spatial relationship between any two objects to be specified.

The remaining sections of this paper are organised as follows. Section 2 reviews the standard approaches to characterizing the spatial relationship between pairs of objects. In Section 3 we explain our approach to recognizing collections of objects (that in accordance with pattern-matching usage we call the *pattern*) as "substructures" of larger collections of objects (that we call the *text*). Section 4 describes four algorithms that recognize substructures using different heuristic strategies to increase efficiency; each of these algorithms consists of three main phases — *preprocess*, *filter* and *check*. In Section 5 we summarize experiments conducted on these algorithms designed to evaluate their relative efficiency. Section 6 presents conclusions and describes future work.

2. Characterizing Spatial Relationships

2.1. Matrix Representation

Spatial relationships play an important role in spatial reasoning and spatial query languages. In an image database, users often want to retrieve images that satisfy their request about spatial relationships among objects. To retrieve information as users require from an image database, first we need a definition of spatial relationships between objects. A second requirement is a proper way to describe these relationships; in other words, a notation by which we can transfer an image into a symbolic representation. Third, we need retrieval algorithms that execute efficiently on this notation.

Early in 1987, Chang *et. al.* [?] proposed a new way of representing a symbolic picture by a two-dimensional string, which preserves the objects' spatial relationships embedded in images. Each object in an image is represented by a symbol. A 2D string, which consists of two 1D strings, is obtained by symbolic projections of these symbols along the horizontal and vertical axes. Therefore, an image similarity retrieval problem becomes a 2D subsequence matching problem. Also, Chang *et. al.* defined three kinds of *rank* for each symbol in a 1D string. Then a 2D subsequence match is correspondingly defined as *Type 0*, *Type 1*, and *Type 2*, which provides three different levels of sophistication in matching, according to the ranks of the symbols in the strings.

In order to improve the performance of 2D strings, variations such as 2D E-strings [?], 2D G-strings [?], 2D C-strings [?], 2D B-strings [?], geometry-based θR-strings [?], Attributed Relational Graphs [?] and spatial orientation graphs [?] have been proposed.

In this paper a square matrix is used to represent the spatial relationships among objects in an image. An image is considered to be a collection of objects (note that the word object(s) is used in this paper but the technique applies to regions as well), which pairwise have some kind of spatial relationship with each other. An image containing n objects is represented by a $n \times n$ matrix T, in which every entry $T[i, j]$ is a "letter" drawn from an "alphabet" representing the relationship of object i to object j. Similarly, a query image can also be represented in this way. The problem of determining whether a query image, called the "pattern", is a subimage of another given image, called the "text", becomes a 2D matrix matching problem.

In contrast to the classical 2D pattern matching problem, where the objective is to find all submatrices of the text matrix which match the pattern matrix, the proposed approach searches for all "substructures" of the text matrix that can match to, not only the pattern matrix, but also any permutation of the pattern matrix.

As we shall see, this problem is a generalization of the subgraph isomorphism problem, which is known to be NP-complete [?]. Nevertheless, the algorithms we propose here for its solution appear to be very fast in practice, especially for the more sophisticated levels (Types 1 and 2) of matching.

2.2. Spatial Relationships

In this section we introduce our approach to the image similarity retrieval problem using 2D matrices, especially the spatial relationships and orthogonal relations we are going to adopt from the previous 2D string approaches.

Given an image, we set up the x, y coordinates and enclose every object by a Minimum Bounding Rectangle (MBR): the smallest rectangle that can completely enclose an object with sides parallel to the x and y axes. See Figure 1. Thus, every object in an image can be represented by a 4-tuple (*x-begin, x-end, y-begin, y-end*). According to the begin and end boundaries of objects, spatial relationships between two objects are categorized into 13 types in one dimension, as shown in Table 1, and thus a total of 169 types in two dimensions. This is *Type 2* matching.

The Type 2 relationships can be partitioned into five categories — *Disjoint, Edge, Contain, Belong* and *Partial Overlap* — similar to those found in a 2D B-String, except that the begin and end boundaries of the MBRs are employed here instead of the objects' ranks. This is called *Type 0* matching, the least sophisticated and least precise.

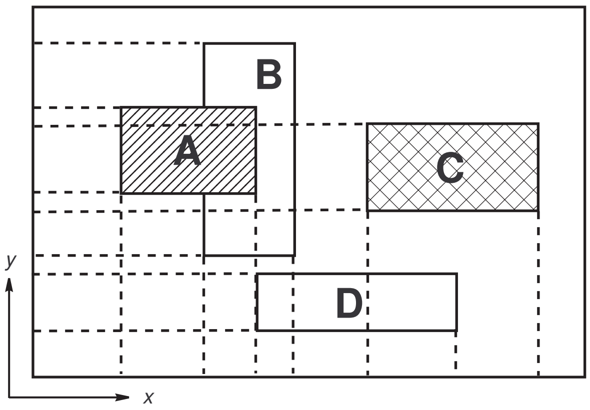


Figure 1. A sample image.

Finally, the four basic orthogonal relationships defined in the 2D B-string representation can be used to define 16 types of "orthogonal" relationships between pairs of objects. These relations are presented in Table 2, where *"E", "W", "N", "S"* denotes *east, west, north,* and *south* respectively, and *"E.W"* denotes the relation "in the *east and west*", *"N.S"* denotes the relation "in the *north and south*", etc. The first type of relation " – " denotes *"not east, not west, not north, and not south"*. This approach yields *Type 1* matching.

We use three distinct sets of symbols (alphabets) to represent the 5 types of relationship categories (Type 0), the 16 types of orthogonal relationships (Type 1), and the 169 types of spatial relationships (Type 2), respectively.

An image containing n objects is represented by a $n \times n$ square matrix $T[0..n-1, 0..n-1]$, in which every entry $T[i,j]$ is a letter drawn from an alphabet representing the relationship of object i to object j. As an example, consider the image in Figure 1. If we use alphabet $\Sigma_0 = \{d, e, c, b, p, -\}$ to represent the five types of spatial relationship categories, where "$-$" denotes the empty relation of one object to itself, then the image can be represented as

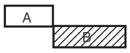
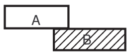
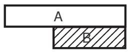
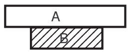
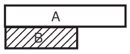
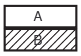
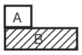
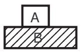
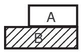
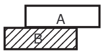
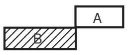
No.	Notation	Condition	Example
(1)	$A < B$	$end(A) < begin(B)$	
(2)	$A\|B$	$end(A) = begin(B)$	
(3)	A/B	$begin(A) < begin(B) < end(A) < end(B)$	
(4)	$A]B$	$begin(A) < begin(B) \wedge end(A) = end(B)$	
(5)	$A\%B$	$begin(A) < begin(B) \wedge end(A) > end(B)$	
(6)	$A[B$	$begin(A) = begin(B) \wedge end(A) > end(B)$	
(7)	$A = B$	$begin(A) = begin(B) \wedge end(A) = end(B)$	
(8)	$B[A$	$begin(A) = begin(B) \wedge end(B) > end(A)$	
(9)	$B\%A$	$begin(B) < begin(A) \wedge end(B) > end(A)$	
(10)	$B]A$	$begin(B) < begin(A) \wedge end(A) = end(B)$	
(11)	B/A	$begin(B) < begin(A) < end(B) < end(A)$	
(12)	$B\|A$	$end(B) = begin(A)$	
(13)	$B < A$	$end(B) < begin(A)$	

Table 1

13 types of spatial relations between two objects A and B in one dimension [?].

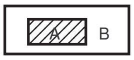
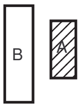
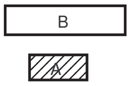
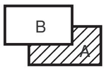
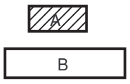
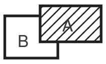
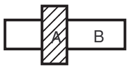
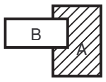
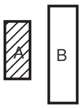
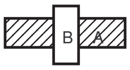
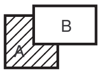
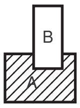
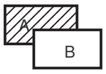
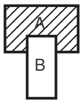
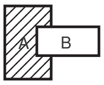
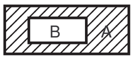
No.	**Relation**	**Example**	**No.**	**Relation**	**Example**
(1)	–		(9)	E	
(2)	S		(10)	$E.S$	
(3)	N		(11)	$E.N$	
(4)	$N.S$		(12)	$E.N.S$	
(5)	W		(13)	$E.W$	
(6)	$W.S$		(14)	$E.W.S$	
(7)	$W.N$		(15)	$E.W.N$	
(8)	$W.N.S$		(16)	$E.W.N.S$	

Table 2

16 types of extended orthogonal relations of object A to object B.

the Type 0 square matrix T_0 shown in Figure 2(a). If we use alphabet $\Sigma_1 = \{1, 2, ..., 16, -\}$ to represent the 16 types of extended orthogonal relations (see Table 2), then the Type 1 square matrix T_1 shown in Figure 2(b) is the representation. And if we use alphabet $\Sigma_2 = \{1, 2, ..., 169, -\}$ to represent the 169 types of spatial relationships, then the image can be represented by the Type 2 square matrix T_2 shown in Figure 2(c), where the entries $T_2[i, j] = (a - 1) \times 13 + b$ while $i \neq j$, and a, $b \in \{1, 2, ..., 13\}$, are one of the 13 spatial relationships (see Table 1) of object i to object j in x and y axes respectively. In T_0, T_1 and T_2, the rows/columns 0, 1, 2, 3 correspond to objects A, B, C and D respectively.

$$\begin{array}{c|cccc|} \hline 0 & - & p & d & d \\ 1 & p & - & d & d \\ 2 & d & d & - & d \\ 3 & d & d & d & - \\ \hline \end{array}$$

(a) T_0

$$\begin{array}{c|cccc|} \hline 0 & - & 5 & 7 & 7 \\ 1 & 12 & - & 8 & 7 \\ 2 & 10 & 9 & - & 11 \\ 3 & 10 & 10 & 6 & - \\ \hline \end{array}$$

(b) T_1

$$\begin{array}{c|cccc|} \hline 0 & - & 35 & 11 & 26 \\ 1 & 135 & - & 5 & 39 \\ 2 & 159 & 165 & - & 143 \\ 3 & 144 & 131 & 27 & - \\ \hline \end{array}$$

(c) T_2

Figure 2. Using a 2D matrix to represent an image.

Using a 2D square matrix to represent an image and preserve the spatial relationships between objects is natural and straightforward. There is no need of any deduction rules, and binary relations between any two objects are easy to retrieve. As we can see from the previous examples, the matrix representation is based on the definition of spatial relationships. Although we use three different alphabets (three different definitions of spatial relationships) as criteria of matching, clearly this approach is not restricted by these definitions. Once the spatial relationships between objects are defined, or in other words, once we have determined which alphabet we are going to use, then we can transform the problem of determining whether a query image, called the "pattern", is a subimage of another given image, called the "text", into a 2D matrix-matching problem.

3. Patterns as Substructures of the Text

This section introduces the special properties of 2D matrix-matching.

3.1. The Problem

The classical string pattern-matching problem is to find all (exact or approximate) occurrences of a pattern P in a text T. In the one-dimensional case, P and T are strings over some alphabet Σ. In our application, P and T are two-dimensional arrays over Σ. We use an $m \times m$ matrix $P = P[0..m-1, 0..m-1]$ to denote a query image (the pattern), and another $n \times n$ matrix $T = T[0..n-1, 0..n-1]$ to denote an image in the image database (the text). The objects in the query image are numbered $0, 1, \ldots, m-1$, those in the database image $0, 1, \ldots, n-1$, and so $P[i, j]$ or $T[i, j]$, $i \neq j$, describes the spatial relationship (in terms of one of the alphabets described above) of object i to object j. For diagonal entries $[i, i]$ we introduce the symbol $-$ to describe the relationship of object i to itself. Of course our objective is to find all matches of P within T.

Our problem is different from classical 2D pattern-matching in one important way. The objective of classical 2D pattern-matching is to find all submatrices of the text matrix T which are identical to the pattern matrix P. However, as we shall see, our problem is more difficult: a valid match P' in T is not necessarily identical to P — P' could be a permutation of P, and so we are not looking only for a matching submatrix of T, but rather a matching "substructure".

$$P: \quad \begin{array}{c} 0 \\ 1 \\ 2 \end{array} \boxed{\begin{array}{ccc} - & a & b \\ c & - & d \\ e & a & - \end{array}}$$

$$T: \quad \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \boxed{\begin{array}{ccccc} - & a & b & b & a \\ c & - & d & d & e \\ e & a & - & c & d \\ e & a & a & - & b \\ b & c & a & e & - \end{array}}$$

Figure 3. Example of a pattern matrix and a text matrix.

$$P_1: \begin{array}{c} 0 \\ 1 \\ 2 \end{array} \boxed{\begin{array}{ccc} - & a & b \\ c & - & d \\ e & a & - \end{array}} \qquad P_2: \begin{array}{c} 0 \\ 2 \\ 1 \end{array} \boxed{\begin{array}{ccc} - & b & a \\ e & - & a \\ c & d & - \end{array}} \qquad P_3: \begin{array}{c} 1 \\ 0 \\ 2 \end{array} \boxed{\begin{array}{ccc} - & c & d \\ a & - & b \\ a & e & - \end{array}}$$

$$P_4: \begin{array}{c} 1 \\ 2 \\ 0 \end{array} \boxed{\begin{array}{ccc} - & d & c \\ a & - & e \\ a & b & - \end{array}} \qquad P_5: \begin{array}{c} 2 \\ 0 \\ 1 \end{array} \boxed{\begin{array}{ccc} - & e & a \\ b & - & a \\ d & c & - \end{array}} \qquad P_6: \begin{array}{c} 2 \\ 1 \\ 0 \end{array} \boxed{\begin{array}{ccc} - & a & e \\ d & - & c \\ b & a & - \end{array}}$$

Figure 4. Six permutation matrices of P.

3.2. An Example

Suppose we are given a pattern matrix P, representing an image with three objects, and a text matrix T, representing an image with five objects, both defined over an alphabet $\Sigma = \{-, a, b, c, d, e\}$, as shown in Figure 3. We now consider the $3! = 6$ permutations of $m = 3$, and use each permutation to simultaneously permute the rows and columns of P as shown in Figure 4. We can represent the permuted matrices by their permutations: $\{0, 1, 2\}$, $\{0, 2, 1\}$, $\{1, 0, 2\}$, $\{1, 2, 0\}$, $\{2, 0, 1\}$, $\{2, 1, 0\}$.

In general, for a permutation function $\Pi : \{0..m-1\} \to \{0..m-1\}$, we can define the permutation matrix $P' = \Pi(P)$ of P to satisfy $P'[i,j] = P[\Pi(i), \Pi(j)]$ for every $i, j \in 0..m-1$. The following useful properties are immediate:

(Q1) The relationship between permutation matrices is an equivalence relation: reflexive, symmetric and transitive.

(Q2) There are at most $m!$ distinct permutation matrices of P.

(Q3) $\Pi(P)$ preserves the rows (respectively, columns) of P, though in a different order.

(Q4) $\Pi(P)$ preserves the symmetry of every pair $P[i,j], P[j,i]$ with respect to the main diagonal.

This is the key point about our form of 2D matching: since we do not assume that the objects in the pattern and text images have been preprocessed and recognized by image processing techniques (except MBR processing), it follows that the ordering of the entries in each matrix is arbitrary: the objects in P may not correspond directly to those in T — that is, they may appear in a different order. Hence the requirement to find all the permutations of P within T: a valid match is not necessarily a submatrix of T.

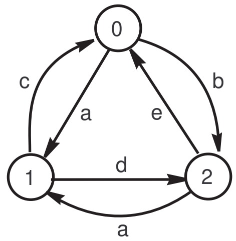
Returning to our example, there is a total of three matches as shown in Figure 5. These three matches can be denoted $012, 013, 234$, identifying the rows (columns) of T within which a match occurs. We consider each of these matches to be valid because each of them is identical to one of the permutation matrices of P.

$$\text{match to } P_1: \quad \begin{array}{c|ccccc|} \cline{2-6} 0 & - & a & b & & \\ 1 & c & - & d & & \\ 2 & e & a & - & & \\ 3 & & & & & \\ 4 & & & & & \\ \cline{2-6} \end{array}$$

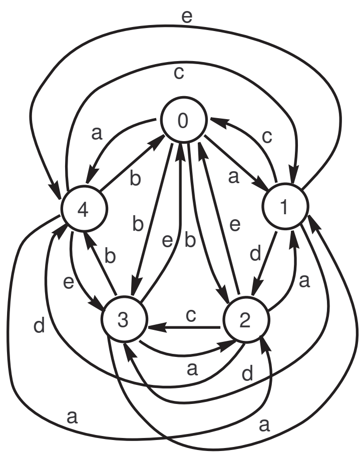
$$\text{match to } P_1: \quad \begin{array}{c|ccccc|} \cline{2-6} 0 & - & a & & b & \\ 1 & c & - & & d & \\ 2 & & & & & \\ 3 & e & a & & - & \\ 4 & & & & & \\ \cline{2-6} \end{array}$$

$$\text{match to } P_3: \quad \begin{array}{c|ccccc|} \cline{2-6} 0 & & & & & \\ 1 & & & & & \\ 2 & & & - & c & d \\ 3 & & & a & - & b \\ 4 & & & a & e & - \\ \cline{2-6} \end{array}$$

Figure 5. Three substructures of T that match P.

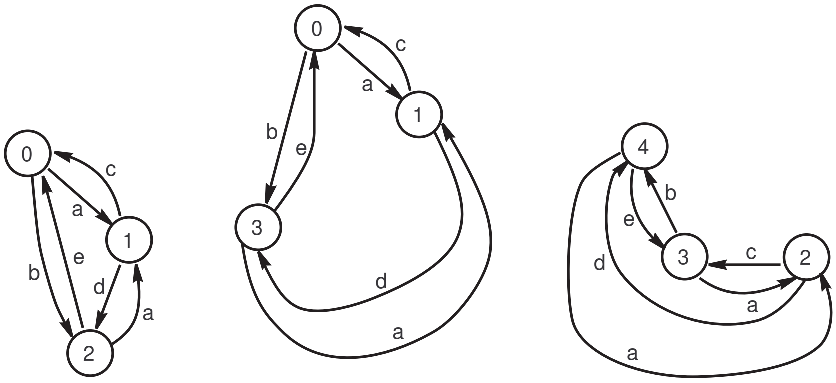


(a) Pattern P



(b) Text T

Figure 6. Graph representations of P and T.



(a) Match 1 (b) Match 2 (c) Match 3

Figure 7. Three subgraphs of T which are isomorphic to P.

Definition 3.1 Given a matrix $T[0..n-1, 0..n-1]$ and a sequence $I = \{i_1, i_2, ..., i_m\}$ of $m \leq n$ strictly increasing nonnegative integers, $i_m \leq n-1$, the intersections of the m rows and m columns of T specified by I form a new matrix $P'[0..m-1, 0..m-1]$ called a *substructure* of T.

Definition 3.2 A substructure P' of T is a *match* for P if there exists a permutation function Π such that $P' = \Pi(P)$.

We see from Definition 3.1 that every submatrix of T is also a substructure of T, but of course the converse is not true. Our matching problem then becomes a search for substructures of T that are permutations of P.

In fact, we see that the 2D matrix-matching problem is a generalization of the subgraph isomorphism problem, determined by replacing an alphabet of two letters (1 for an edge, 0 for no edge) by the larger alphabets of sizes 5, 16 or 169. Even though the subgraph isomorphism problem is known to be NP-complete [**?**], it appears that in practice our generalization of it leads to very efficient algorithms — due to the increase in alphabet size, it turns out that the likelihood of false matches is small, occurring only in pathological cases that should not arise in practice. In the case that the number of objects in the text and in the pattern is the same, our problem reduces to a generalization of the graph isomorphism problem, whose complexity has not yet been established [**?**].

Returning to our example, we show the graph representations of the pattern P and text T in Figure 6. And the three subgraphs of T which are isomorphic to P, i.e. the three matches, are shown in Figure 7, where the permutations from P to them are 0 1 2, 0 1 3, and 3 2 4 respectively.

In the following section, we will introduce the approaches and search strategies that could apply to our 2D matrix-matching problem. We always assume that the sizes of the pattern and the text are m and n respectively. Our algorithms consist of three stages. In each stage, there are one or more strategic options. We believe that in applications, the text T often contains relatively fewer occurrences of the pattern P. Therefore, a very simple elimination idea plays an important role in the first two stages.

4. Description of the Algorithms

In this section we first describe the three main stages of each algorithm (Preprocessing, Filtering, Checking), then outline each of the four algorithms in terms of the strategy each one uses in these stages.

4.1. Stage I: Preprocess

In this stage we compute a signature vector for each row (or column) of P and T, then use the signatures to eliminate search possibilities. By the end of this stage, we will have m lists. The elements in list i ($i \in 0..m-1$) are the row numbers of T that could have one or more matches to row i of P.

Definition 4.1 Given an $m \times m$ matrix P, and an alphabet Σ of size σ, the *row signature* of P is a matrix $SigP_r[0..m-1, 0..\sigma-1]$, where the entry $SigP_r[i,j]$ is the total number of occurrences of the jth element of Σ in row i of P.

$SigT_r$ is analogously defined. For example, if $\Sigma = \{-, a, b, c, d, e\}$, and we have pattern matrix P and text matrix T as shown in Figure 8, then we will have the row signatures $SigP_r$ and $SigT_r$, also shown in Figure 8. We can similarly define *column signatures* $SigP_c$ and $SigT_c$, but we will deal mainly with row signatures and simply denote them $SigP$ and $SigT$.

Provided that we can map the letters of the alphabet onto the integers $0, 1, \ldots, \sigma-1$ (as in our application we can), the calculation of row signatures for P and T is straightforward. Note that, by (Q3), we know that for all $m!$ permutation matrices of P, the row signatures will be the same as that of P, except in a different order as a result of the permutations. So here we can use $SigP[0..m-1, \sigma-1]$ to denote all the row signatures without specifying which permutation matrix it actually corresponds to.

To compute $SigP$ will require $\Theta(m\sigma)$ time for initialization plus $\Theta(m^2)$ time for the m^2 elements of P. With analogous time requirements for $SigT$, the total time required to compute the signatures of P and T will be

$$\Theta(m\sigma) + \Theta(m^2) + \Theta(n\sigma) + \Theta(n^2) = O\big(\max\{n, \sigma\}n\big). \quad (1)$$

$$P: \begin{array}{c} 0 \\ 1 \\ 2 \end{array} \begin{array}{|ccc|} \hline - & a & b \\ c & - & d \\ e & a & - \\ \hline \end{array}$$

$$T: \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{|ccccc|} \hline - & a & b & b & a \\ c & - & d & d & e \\ e & a & - & c & d \\ e & a & a & - & b \\ b & c & a & e & - \\ \hline \end{array}$$

$$SigP[0..2, 0..5]: \begin{array}{c} 0 \\ 1 \\ 2 \end{array} \begin{array}{|cccccc|} \hline 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

$$SigT[0..4, 0..5]: \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{|cccccc|} \hline 1 & 2 & 2 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 2 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ \hline \end{array}$$

Figure 8. Example of a pattern matrix P, a text matrix T and their row signatures.

We now use the signatures to eliminate search possibilities by comparing the signatures of each row i of $SigP$ with each row j of $SigT$: if there exists an entry in position k of row i of $SigP$ greater than $SigT[j,k]$, we can conclude that row i of P cannot match with row j of T; that is, row i of P is not a subsequence of row j of T. In this way we can eliminate some rows that do not contain a match. Initially, we have m empty lists. At the end of this step, elements in list i ($i \in 0..m-1$) are the row numbers of T that could possibly have one or more matches to row i of P.

Example

Continuing the example of Figure 8, if we pick row 0 of $SigP$, which is ${}_0\,\boxed{1\ 1\ 1\ 0\ 0\ 0}$, and compare it with every row of $SigT$

$$\begin{array}{rl} {\scriptstyle 0} & \boxed{1\ 2\ 2\ 0\ 0\ 0} \\ {\scriptstyle 1} & \boxed{1\ 0\ 0\ 1\ 2\ 1} \\ {\scriptstyle 2} & \boxed{1\ 1\ 0\ 1\ 1\ 1} \\ {\scriptstyle 3} & \boxed{1\ 2\ 1\ 0\ 0\ 1} \\ {\scriptstyle 4} & \boxed{1\ 1\ 1\ 1\ 0\ 1}, \end{array}$$

then we know that possible matches of row 0 of P can occur in row 0, 3 and 4 of T, because all entries in these rows of $SigT$ are not less than the corresponding entries of row 0 of $SigP$. And there is no possible match of row 0 of P with row 1 and 2 of T. In other words, we know that row 0 of P contains exactly one a and one b, while rows 0, 3 and 4 of T contain at least one a and one b, so there are possible matches. And row 1 and 2 of T do not contain at least one a and one b, so there are no possible matches.

Continuing our example, after we compare every row of $SigP$ to every row of $SigT$, at the end of this step, we will have three lists:

$$\text{list } 0 = \{0,3,4\};\ \text{list } 1 = \{1,2\};\ \text{list } 2 = \{2,3,4\}. \tag{2}$$

Elements in lists $0,1,2$ are the row numbers of T that could possibly have one or more matches to row $0,1,2$ of P, respectively.

The time complexity for comparing $SigP$ with $SigT$ will be $\Theta(m)$ for list initialization, plus $O(\sigma mn)$ for comparing $SigP$ with $SigT$ and adding row numbers to m lists. Thus this step requires time $\Theta(m) + O(\sigma mn) = O(\sigma mn)$. In view of (1), the overall execution time for Stage I is therefore

$$O\big(\max\{n, \sigma m\} n\big), \tag{3}$$

while the storage of up to m nonempty lists of length at most n of course requires $O(mn)$ space.

4.2. Stage II: Filter

If the lists output by Stage I are all empty, there is no match. Otherwise, we make use of one of two strategies to eliminate as many potential matches as possible from the nonempty lists:

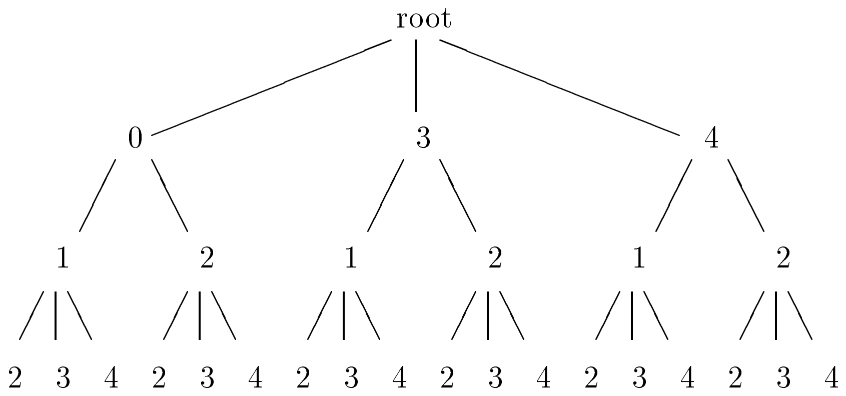


Figure 9. A tree shows all possible row sequences

- Select feasible row sequences: use compatible row numbers to select feasible row sequences of T that correspond to rows of P.

- Select feasible column sequences: use the row i of P that is compatible with the fewest number of rows of T to identify all feasible column sequences of T.

To save space, we do not save a set of feasible row sequences or column sequences in memory. Once we get a feasible row or column sequence, we immediately pass it to the next stage.

4.2.1. Select Feasible Row Sequences

In the m lists we find every ***feasible*** row sequence; that is, every sequence of distinct integers $(row_0, row_1, ..., row_{m-1})$ such that for $i \in 0..m-1$, $row_i \in$ list i.

Example

Using the three lists (2) obtained at the end of Stage I, there are $3 \times 2 \times 3 = 18$ possibilities we need to check. We may represent the ***possible*** row sequences by paths in a rooted tree. Using a dummy node as root (level 0), the level 1 nodes will be the t_0 nodes of list 0; then at level 2 the t_1 nodes of list 1 will be duplicated t_0 times, and so on until level $m-1$ is reached. Thus, as shown in Figure 9, a path from the root to one of the leaves will represent one of the possible row sequences; for example, $(0, 1, 2)$, $(3, 1, 4)$. The number of leaves is the number of possibilities.

But according to the definition of a feasible row sequence, the integers on each path must be distinct. Thus, eliminating paths with duplicate entries, we obtain the reduced tree shown in Figure 10, now with only 11 feasible paths: (0,1,2), (0,1,3), (0,1,4), (0,2,3), (0,2,4), (3,1,2), (3,1,4), (3,2,4), (4,1,2), (4,1,3), (4,2,3).

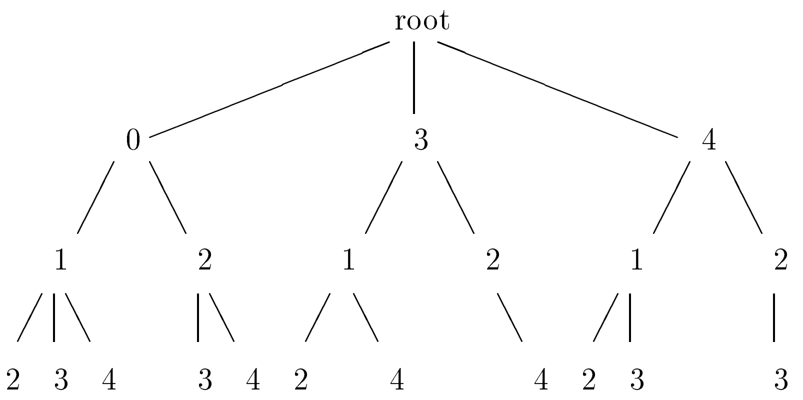


Figure 10. A pruned tree shows all feasible row sequences

Outline of Algorithm

For each of up to n^m possible row sequences, each of length m, we need to select those that are feasible. Of course, with this approach, our hope and expectation is that the number of entries in each list will be much less than n, hence that the actual number of row sequences to be checked will be much less than the n^m maximum. Using a bit map $\texttt{MARK}[1..n]$ to indicate whether or not any row number i has already occurred in the row sequence currently being checked, it is straightforward to determine the feasibility of each row sequence (and reset MARK to its initialized state) in $O(m)$ time.

Stage II carries out this calculation for the row sequences one by one; when a feasible row sequence is found, it is immediately passed on to Stage III for further processing, thus avoiding unnecessary storage. Hence the additional space requirement for this approach is only $O(n+m)$, while the total processing time is

$$O(mn^m). \tag{4}$$

4.2.2. Select Feasible Column Sequences

Another strategy is to select all feasible column sequences. Instead of using all the m lists, we choose the shortest list — the one with the fewest elements — and use it to identify all the feasible column sequences in T. A ***possible*** column sequence is a sequence of column numbers $(col_0, col_1, ..., col_{m-1})$ obtained when we compare row i of P to row i' of T, where i is the number of the list with the fewest number of elements, and i' is an element in list i. Thus a possible column sequence satisfies $P[i,j] = T[i', col_j]$ for all $j \in 0..m-1$. As in Subsubsection 4.2.1, a possible column sequence is also ***feasible*** if no two column numbers are equal. So this strategy involves comparing selected rows of P with those of T until a sequence of distinct column numbers is found, then passing the sequence on to the next processing stage.

Example

Again using the three lists (2) of Stage I, we choose list $1 = \{1, 2\}$ with the fewest elements. Then we compare row 1 of P, which is: $_1\boxed{c - d}$, to rows 1 and 2 of T, which are: $_1\boxed{c - d\ d\ e}$ and $_2\boxed{e\ a - c\ d}$. For every entry in row 1 of P, we record its occurrences (the column numbers) in rows 1 and 2 of T. The combinations of these occurrences form a set of all possible column sequences $\{(0, 1, 2), (0, 1, 3), (3, 2, 4)\}$, from which we filter out the infeasible ones (in this case, there are none) to yield the set of feasible column sequences.

Outline of Algorithm

Suppose list i is the one with the fewest entries. Then for every entry $i' \in$ list i, we first need to find, for every $k \in 0..m - 1$, a match $P[i, k] = T[i', k']$. Observe that the total number of such matches is

$$Q(i, i', k) = \prod_{\text{all distinct } P[i,k]} \binom{SigT\big[i', P[i, k]\big]}{SigP\big[i, P[i, k]\big]},$$

satisfying $Q(i, i', k) \leq \binom{n/m}{m} \in O\big((n/m)^m\big)$.

The processing determines a set of distinct occurrences, $SigP\big[i, P[i, k]\big]$ in number, of each distinct $P[i, k]$ in row $T[i', \bullet]$; over all choices of $P[i, k]$, all of these sets can be determined in $O(mn)$ time. then each possible column sequence can be generated in $O(m)$ time. A `MARK` bit vector can again be used to determine in $O(m)$ time whether each possible column sequence is also feasible. Thus the overall time requirement for this approach is

$$O\big((n/m)^m(mn + m)\big) = O(n^{m+1}/m^{m-1}), \tag{5}$$

while as in Subsubsection 4.2.1 the feasible column sequences are generated one at a time, so that only $O(n + m)$ additional space is required.

4.3. Stage III: Checking

In this stage, we need to test a feasible row or column sequence $S = \{s_0, s_1, \ldots, s_{m-1}\}$ received from Stage 2 in order to determine whether or not S defines a substructure of T that matches with P. Again we can employ one of two different strategies: either direct checking of the sequences or else direct checking preceded by another filter designed to reduce time requirements in some cases.

4.3.1. Direct Checking

Here for every $i, j \in 0..m - 1$, we compare $P[i, j]$ with $T[s_i, s_j]$ in a straightforward manner. If equality holds for every comparison, then we accept the sequence S as defining a substructure of T that matches P; but if in any case inequality holds, S is rejected.

In our example, choosing $S = \{3, 2, 4\}$, we find (see Figure 3) that

$$P[0, 0] = T[3, 3] = -,\ \ P[0, 2] = T[3, 2] = a,\ \ P[0, 2] = T[3, 4] = b,$$

and so on, so that S does indeed define a matching substructure of T.

The processing time required for direct checking of each sequence is

$$O(m^2), \tag{6}$$

while only constant additional space is required.

4.3.2. Filtered Checking

Recall that in Stage I, $SigP$ and $SigT$ are computed for rows of P and T, respectively, leading to lists $i \in 0..m-1$ that specify row numbers of T possibly matching with row i of P. The information in these lists can be represented by bit vectors $R_i[0..n-1]$, where

$$R_i[j] = \begin{cases} 1 & \text{if } j \in \text{list } i; \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, if column signatures are computed in Stage I, we can define an analogous collection of column lists also representable by bit vectors $C_i[0..n-1]$, where

$$C_i[j] = \begin{cases} 1 & \text{if } j \in \text{list } i; \\ 0 & \text{otherwise.} \end{cases}$$

Given a feasible row sequence $(row_0, row_1, ..., row_{m-1})$, by Definition 3.1, if the substructure corresponding to this sequence is a match, then $(row_0, row_1, ..., row_{m-1})$ must also be a feasible column sequence. So we can check whether the corresponding column bits $C_0[row_0], C_1[row_1], ..., C_{m-1}[row_{m-1}]$ are all set to 1. If not, then we need not consider this row sequence further, because the substructure determined by it is not a match to P.

For example, using our pattern P and text T as shown in Figure 8, the row bit vectors and column bit vectors for P are:

$$R_0[0..4] = \boxed{1\ 0\ 0\ 1\ 1} \qquad C_0[0..4] = \boxed{1\ 0\ 0\ 1\ 0}$$
$$R_1[0..4] = \boxed{0\ 1\ 1\ 0\ 0} \qquad C_1[0..4] = \boxed{0\ 1\ 1\ 0\ 0}$$
$$R_2[0..4] = \boxed{0\ 0\ 1\ 1\ 1} \qquad C_2[0..4] = \boxed{0\ 0\ 1\ 1\ 1}$$

Then given a feasible row sequence (3,2,4) obtained in Stage II, the corresponding column bits $C_0[3] = 1$, $C_1[2] = 1$, and $C_2[4] = 1$, so the sequence (3,2,4) needs further checking. If however the row sequence were (4,1,2), the corresponding column bit $C_0[4]$ would be zero, and so (4,1,2) could be eliminated.

Thus this bit vector strategy functions as another filter: only the feasible row sequences or column sequences passing it need further consideration by direct checking. The bit vector check requires an additional $\Theta(mn)$ space for the bit vectors, but executes in $O(m)$ time. In using this strategy, the hope is that enough row lists can be eliminated from consideration in $O(m)$ time so that only a few of them need to be processed by the $O(m^2)$-time direct check, thus saving processing time overall.

4.4. Summaries of the Algorithms

Comparing the asymptotic time estimates for Stages I–III, we find that Stage II may generate up to n^m row sequences or up to $\binom{n/m}{m}$ column sequences, each potentially requiring $O(n)$ processing time plus $O(m^2)$ time in Stage III. Thus in the worst case, the processing required by combinations of these stages can certainly be exponential in m. On the other hand, only pathological cases that should not occur in practice (for example, all off-diagonal elements of P and T identical) lead to high processing time. In practice, especially for larger alphabets (thus when spatial identification is more precise), the algorithms derived from our Stage I–III strategies appear to be very efficient.

We consider the following four algorithms, based on combinations of strategies given in Stages I–III:

Algorithm 1

- Stage I:
  - Compute row signatures for the pattern and the text.
  - Compare two row signatures to get the lists corresponding to every row of the pattern.
- Stage II: Select feasible row sequences.
- Stage III: Direct Checking.

Algorithm 2

- Stage I: As for Algorithm 1.
- Stage II: Pick the shortest list, and select feasible column sequences.
- Stage III: Direct Checking.

Algorithm 3

- Stage I:
  - As for Algorithm 1.
  - Also compute and compare column signatures for the pattern and the text, and compute the column bit vectors for the pattern.
- Stage II: Select feasible row sequences.
- Stage III:
  - Use column bit vectors to filter out infeasible row sequences.
  - Direct Checking.

Algorithm 4

- Stage I:
  - As for Algorithm 1.
  - Also compute the row bit vectors of the pattern.

- Stage II: Pick the shortest list, and select feasible column sequences.
- Stage III:
  - Use row bit vectors to filter out infeasible column sequences.
  - Direct Checking.

5. Experimental Results

The four algorithms described in Section 4 have been implemented and tested against computer-generated images (both pattern and text images) in which objects are created in pseudorandom position within the image and with sizes pseudorandomly generated according to two different "systems":

System 1: objects tend to be large with respect to the space available for the entire image, and so tend to overlap frequently;

System 2: objects tend to be small with respect to the image, and so rarely overlap.

System 2 yields images in which the objects have pairwise spatial relationships that tend to be highly repetitive. For example, "disjoint" is itself a category of Type 0 matching (Section 2.2); thus almost all Type 0 entries in pattern and text matrices generated under System 2 will be identical. Furthermore, it turns out that even for Types 1 and 2 matching, System 2 matrices contain entries that with high probability are drawn from a small subset (four letters or so) of the alphabet. thus System 2 yields matching problems that are "difficult" in the sense of the subgraph isomorphism problem: since the alphabet effectively used is small, a very large number of false matches may occur.

On the other hand, System 1 yields images in which there is more variety in the spatial relationships that are likely to exist between pairs of objects. For Type 0 matrices, disjoint objects are still very common, but in the Type 1 and Type 2 matrices, there is a much more even spread among the probabilities of occurrence of the various spatial relationships.

Thus, not surprisingly, in our experiments on the implementations of Algorithms 1–4, we found that problems generated using System 1 with Type 1 or Type 2 spatial relationships could be solved very quickly — typically, for $n = 50, m \leq 20$, less than 5 milliseconds were required on a 100-megahertz Solaris computer. On the other hand, Type 0 runs on System 2 problems could require minutes.

We do not give these results in detail because, as discussed in the next section, we believe that detailed timings will be interesting only for real-life images or for those generated using more sophisticated techniques to simulate real-life images more convincingly. But more fundamentally, it seems clear that the three distance types traditionally used to provide the matrix elements need to be reconsidered: it appears that none of the Types 0–2 are sufficiently precise and flexible to provide good descriptions of the relative positions of objects in the plane.

That being said, we can provide provisional information about the relative efficiency of the four algorithms. Based on tests done using Systems 1–2 and Types 0–2, we have found that Algorithm 3 is never the fastest of the four, while Algorithm 4 appears to be the most consistent: generally either the fastest or within 10% of the fastest algorithm.

This reliability perhaps reflects the fact that Algorithm 4 uses a filter both in Stages II and III.

6. Conclusions & Future Work

In this paper we have described strategies for locating "pattern" images within "text" images, where the objects in each image are specified only in terms of their pairwise spatial relationships. Thus both pattern and text are represented by square matrices whose elements are drawn from an "alphabet" of spatial relationships. A match is found when the pattern image is identified as a "substructure" of a given text image. For an alphabet of size 2, this pattern-matching is equivalent to the NP-hard subgraph isomorphism problem, but we develop filtering strategies that in practice, for larger alphabets, allow us to process realistic text images containing 50 or so objects in milliseconds.

We have tested four algorithms based on different filtering strategies, but have been able to form only tentative conclusions about their relative efficiency. The reason for this is that the execution time of the algorithms depends critically on the way in which the spatial relationships are modelled. We have used three well-known models (Types 0–2) and two different systems for generating test data, but as explained in Section 5, we did not find any of them fully satisfactory as a basis for representing spatial relationships. Thus our first priority for further study is development of a model that encourages a wide diversity in the representation of spatial relationships between pairs of objects.

At the same time, we believe that more effective filtering strategies can be found, leading to even more efficient algorithms that will permit very rapid scanning of text images to search for a specific pattern. We believe that such algorithms would find wide application in geographic information systems.