Processing Pictorial Queries with Multiple Instances using Isomorphic Subgraphs*

Andre Folkers and Hanan Samet
Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Science
University of Maryland at College Park
College Park, Maryland 20742
E-mail: {folkers, hjs}@umiacs.umd.edu

Aya Soffer
IBM Research Laboratory
Matam, Haifa 31905
Israel
E-mail: ayas@il.ibm.com

Abstract

An algorithm is given for processing pictorial query specifications that consist of a query image and a similarity level that must hold between the query image and database images. The similarity level specifies the contextual similarity *(how well the content of one image matches that of another) as well as the* spatial similarity *(the relative locations of the matching symbols in the two images). The algorithm differs from previous approaches in its ability to handle multiple instances of each object in both the query and database images by searching for isomorphic subgraphs. The running time of the algorithm is* $O(m\, 2^m)$ *in the worst case where all symbols in both the query and database image are from the same class, but falls far below this bound in the presence of spatial constraints.*

1. Introduction

A basic requirement of an image database is the ability to query the database pictorially. One of the main issues is whether the similarity criteria used by the database system match those of the user. In [10] we presented a pictorial query specification tool for spatially referenced image databases. Using this tool, a user can specify which objects should or could appear in a target image as well as the number of occurrences of each object. Moreover, it is possible to impose spatial constraints on the distance and relative direction between objects. We also described an algorithm for finding database images that satisfy such pictorial queries. In [11] we expanded this algorithm to handle multiple instances of symbols. This algorithm finds all images that contain the desired symbols and then checks the spatial constraints for all possible matchings of query and database symbols.

Most existing image database research has dealt with global image matching based on color and texture features [5, 8]. There has also been some work on the specification of topological and directional relations among query objects [1–3, 7, 9]. The focus of this work has been on defining spatial relations between objects and efficiently computing them. The issue of multiple instances of objects and the complexity arising from it is at best mentioned as a problem, and usually ignored.

In this paper we describe a method for processing such pictorial queries using isomorphic subgraphs. Both the query image and the database images can be viewed as graphs. The vertices represent symbols and the edges represent the relation between these symbols. There can be several subgraphs in the database image that match a given query graph since each image may have several instances of each symbol. Examining and matching each such subgraph is a very costly operation. This algorithm uses a bottom-up strategy to find all possible subgraphs of a database image that are isomorphic to a query image. The idea is that many potential subgraphs can be eliminated early on by the bottom-up process and thus do not need to be fully matched.

In [6] another strategy for solving this matching problem has been presented using *Error-Tolerant Subgraph Isomorphism Detection.* In the rest of this paper we review pictorial query specification, and present our algorithm and an example of its usage.

2. Notation and Definitions

Let V be the set of all symbols in the database images and in the query image. Let $D = (V_D, E_D)$, $V_D = \{d_1, d_2, \ldots, d_m\} \subseteq V$ be the graph of the symbols in one database image D. Let $Q = (V_Q, E_Q)$, $V_Q = \{q_1, q_2, \ldots, q_n\} \subseteq V$ be the graph of the symbols in the query image Q. We define three functions to denote the

*The support of the National Science Foundation under Grants IRI-97-12715 and CDA-950-3994 is gratefully acknowledged.

basic properties of symbols in V:

$$\mathrm{cl} : V \to C, v \mapsto \mathrm{cl}(v), \tag{1}$$

$$\mathrm{loc} : V \to \mathbf{R}^2, v \mapsto \mathrm{loc}(v) = (x, y)^T, \tag{2}$$

$$\begin{aligned}\mathrm{cert} : V \to [0,1], v \mapsto \mathrm{cert}(v) = \\ \Pr[\mathrm{cl}(v) \text{ is the correct classification of } v].\end{aligned} \tag{3}$$

Function cl assigns a class name to every symbol in V. It can also be applied on sets of symbols. The result is the set of classes that occur in the respective set of symbols. Using the function cl we define the numbers

$$m_c = |\{ d \in V_D \mid \mathrm{cl}(d) = c \}| \quad \text{and} \tag{4}$$

$$n_c = |\{ q \in V_Q \mid \mathrm{cl}(q) = c \}| \tag{5}$$

for each $c \in \mathrm{cl}(V)$. Function loc returns the location of a symbol in the 2-dimensional plane, and function cert returns the probability that the classification of a symbol is correct.

3. Pictorial Query Specification

3.1. Matching Similarity

The *matching similarity level* is a value between 0 and 1. It specifies how certain the classification of a symbol in the database image must be, so that we take it into account. If $\mathrm{cert}(v) \geq msl$, then v is considered similar with respect to the matching similarity level.

3.2. Contextual Similarity

We distinguish between four different *contextual similarity levels* that a database image D can satisfy. Their formal definition is

1. $\forall c \in \mathrm{cl}(V_Q) : m_c \geq n_c$, and $\mathrm{cl}(V_D) = \mathrm{cl}(V_Q)$
2. $\forall c \in \mathrm{cl}(V_Q) : m_c \geq n_c$
3. $\mathrm{cl}(V_D) \subseteq \mathrm{cl}(V_Q)$
4. $\exists c \in \mathrm{cl}(V_D) : c \in \mathrm{cl}(V_Q)$.

Let $R_l = \{D_1, D_2, \dots\}$ denote the set of images that satisfy *csl*=*l* for a certain query Q. For *csl*=1 and *csl*=2, multiple symbols from the same class act with an AND semantic, while for *csl*=3 and *csl*=4 they act with an OR semantic. This means that different numbers of instances of a certain class do not alter the sets R_3 and R_4, but they do alter R_1 and R_2. The query graph is always a subgraph of the graphs in database images in R_1 and R_2. Images in R_3 and R_4 can also be subgraphs of the query image.

3.3. Spatial Similarity

The *spatial similarity level* specifies how close the database image D and the query image Q are with respect to distance and directional relation between the symbols in the query. We distinguish between five different levels which are defined as

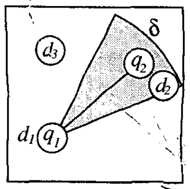

Figure 1. Similar directional relation

1. exact same location
2. same relation, bounded distance
3. same relation, any distance
4. any relation, bounded distance
5. any relation, any distance.

The case *ssl*=1 is ignored in the rest of this paper because there is a more effective algorithm to check this spatial constraint than the one we propose in Section 4.2.

We use the Euclidean distance between the symbols which is denoted by the function dist:

$$\mathrm{dist} : V \times V \to \mathbf{R}, (v_1, v_2) \mapsto \mathrm{dist}(v_1, v_2) \tag{6}$$

To compare two directed egdes $(q_1, q_2) \in E_Q$ and $(d_1, d_2) \in E_D$ with respect to their directional relation we relocate the vertices q_1 and q_2 so that $\mathrm{loc}(q_1) = \mathrm{loc}(d_1)$. The directional relation between (q_1, q_2) and (d_1, d_2) is similar with respect to a threshold angle $\delta \in [0, 2\pi]$, denoted by $(q_1, q_2) \sim_\delta (d_1, d_2)$, if the vertex d_2 is located within a sector with arc length δ. The center of this sector is at $\mathrm{loc}(q_1)$ and it is symmetric to the axis, which is defined by the edge (q_1, q_2). In Figure 1 we see an example where $(d_1, d_2) \sim_\delta (q_1, q_2)$ and $(d_1, d_3) \not\sim_\delta (q_1, q_2)$ holds.

4. Pictorial Query Processing

4.1. Generate the Set of Candidates Images

The function *genCandidates* computes the set R_Q of database images that satisfy the matching similarity level and the contextual similarity level indicated by *msl* and *csl*. The set R denotes all images in the database. At the beginning, for each class c, the database images that contain symbols of class c are stored in sets R_c. If *csl*=1 or *csl*=2, then we have to intersect these sets to get all images that contain at least one symbol of each class in Q. Otherwise, we compute the union of the sets R_c and get a set R_Q of images, where each image D contains at least one of the symbols in Q. Note, that we have to eliminate duplicates in R_c during union or intersection of these sets.

```
funct genCandidates($Q, msl, csl$) : $R_Q$ ≡
  foreach $c \in \mathrm{cl}(V_Q)$ do
    $R_c = \{D \in R \mid \exists v \in V_D : \mathrm{cl}(v) = c \wedge$
        $\mathrm{cert}(v) \geq msl\}$
  end
```

```
if $csl = 1 \lor csl = 2$ then $R_Q = \bigcap_c R_c$ fi
if $csl = 3 \lor csl = 4$ then $R_Q = \bigcup_c R_c$ fi
$R_E = \emptyset$                    /* set of invalid candidates */
foreach $D \in R_Q$ do
  if $csl = 1 \lor csl = 3$
    then if $\mathrm{cl}(D) \not\subseteq \mathrm{cl}(Q)$ then $R_E = R_E \cup D$ fi fi
  if $csl = 1 \lor csl = 2$
    then if $\exists c \in \mathrm{cl}(V_Q) : m_c < n_c$
           then $R_E = R_E \cup D$ fi fi
end
$R_Q = R_Q \setminus R_E$.
```

4.2. Check Spatial Constraints

The algorithm given by function *getIsoSubgraphs* uses a bottom-up strategy to find all possible subgraphs in D that are isomorphic to Q with respect to the parameters of edges and vertices. It is an adaptation of a more general algorithm described in [4].

```
1  funct getIsoSubgraphs($Q, D, ssl$) : $S$ ≡
2    choose $q_1 \in V_Q$
3    $S^{(1)} = \emptyset$
4    foreach $d \in \{ d \in V_D \mid \mathrm{cl}(d) = \mathrm{cl}(q_1) \}$
5      $S^{(1)} = S^{(1)} \dot\cup (\langle q_1 \rangle, \langle d \rangle)$
6    end
7    $i = 1$
8    foreach $q_e \in V_Q \setminus \{q_1\}$
9      $S^{(i+1)} = \emptyset$
10     foreach $G = (\langle q_1, \dots, q_i \rangle, \langle d_1, \dots, d_i \rangle) \in S^{(i)}$
11       foreach $(q_e, d_e) \in$ validExt($G, Q, D, q_e, ssl$)
12         $S^{(i+1)} = S^{(i+1)} \dot\cup$
13           $(\langle q_1, \dots, q_i, q_e \rangle, \langle d_1, \dots, d_i, d_e \rangle)$
14       end
15     end
16     $i = i + 1$
17   end
18   $S = S^{(i)}$.
19 funct validExt($G, Q, D, q_e, ssl$) : $P$ ≡
20   /* $G$ is a set of pairs of isomorphic graphs $G_1, G_2$ */
21   $V_{G_2} = \{ d \in V_D \mid d$ is element of $G_2 \}$
22   $V_{E_D} = \{ d \in V_D \mid \mathrm{cl}(d) = \mathrm{cl}(q_e) \} \setminus V_{G_2}$
23   $P = \emptyset$
24   foreach $d_e \in V_{E_D}$
25     if isValid($G, Q, D, q_e, d_e, ssl$)
26       then $P = P \dot\cup (q_e, d_e)$ fi
27   end.
```

getIsoSubgraphs returns a set S of pairs of isomorphic subgraphs for the parameter graph Q in D that satisfy the spatial constraints indicated by *ssl*. It computes only solutions where the whole graph Q is mapped onto a subgraph of D. As a consequence, *getIsoSubgraphs* works only for queries where *csl*=1 and *csl*=2. Two isomorphic subgraphs are denoted by a pair of sequences of vertices, e.g., the pair $(\langle q_1, \dots, q_i \rangle, \langle d_1, \dots, d_i \rangle)$ denotes that vertex q_k is mapped onto vertex d_k $(k = 1, \dots, i)$.

First, an initial set $S^{(1)}$ of all isomorphic subgraphs with length 1 is created. It consists of pairs $(\langle q_1 \rangle, \langle d \rangle)$ where each $d \in V_D$ belongs to the same class as q_1. Starting with this set, the main loop of the algorithm searches for isomorphic subgraphs of size 2 and based on this, it looks for some of size 3 and so on.

The function *validExt* returns extensions of form (q_e, d_e) where q_e is fixed as it is given as a parameter. The subgraphs that we want to extend are given in $G = (G_1, G_2)$. G_1 is the sequence of vertices of the current subgraph of Q, while G_2 is the sequence of vertices of the current subgraph of D. By construction, q_e is always a new vertex that does not occur in G_1. We compute the set V_{E_D} of remaining vertices which could be used as extensions to the sequence in G_2 (line 22) because we do not want a vertex of V_D to occur twice in sequence G_2.

Every possible extension pair is tested by function *isValid* to determine if it satisfies the spatial constraints. If yes, then it is added to the set *P*. Depending on the *ssl* value, the function *isValid* checks for $k = 1, \dots, i$, if $\mathrm{dist}(d_k, d_e) \le \mathrm{dist}(q_k, q_e)$, or if $(d_k, d_e) \sim_\delta (q_k, q_e)$, or if both hold, and returns the result. It just returns true if *ssl*=5.

The function *getIsoSubgraphs* returns the complete result only for *csl*=1 and *csl*=2. If *csl*=3 or *csl*=4, then we also have to return the mappings of all subgraphs of Q onto subgraphs of D. The brute force approach to compute these mappings is to call *getIsoSubgraphs* for each subgraph of Q and to take a union of all of the results. Since Q is usually a small graph we can use this brute force approach

4.3. Execution Time

First, we derive the worst case complexity of function *getIsoSubgraphs*, assuming that Q has n and D has m vertices, which are all from the same class. We count how often the union operations in lines 5 and 12 are executed. During initialization, we have m union operations. The main loop has $n-1$ iterations and in the i-th iteration, $i = 1, \dots, n-1$, we get $\binom{m}{i}(m-i)$ union operations. This is because the maximal number of elements in $S^{(i)}$ is $\binom{m}{i}$ and *validExt* returns at most $m - i$ extension pairs, which actually occurs if *ssl*=5. Therefore, after some transformations and exploiting that $n \le m$, we get an upper bound on the number of union operations:

$$m + \sum_{i=1}^{n-1} \binom{m}{i}(m-i) \le m(2^m - 1). \qquad (7)$$

This implies a worst case complexity in $\mathrm{O}(m\,2^m)$ for *csl*=1 or *csl*=2. For *csl*=3 and *csl*=4, we get an additional factor of 2^n since we invoke *getIsoSubgraphs* this many times. The worst case complexity is $\mathrm{O}(m\,2^{n+m})$.

Once we add spatial constraints, the order in which the vertices of Q are processed can affect the running time significantly. Choosing a vertex q_i so that its edges with the vertices $\{q_1, \ldots, q_{i-1}\}$ are the most restrictive ones with respect to the spatial constraints leads to a shorter running time for *getIsoSubgraphs* (see the example in Section 4.4).

4.4. Finding the Isomorphic Subgraphs

We want to show how function *getIsoSubgraphs* finds the possible mappings. In Figure 2 we see a query graph $Q = (\{a, b, c\}, E_Q)$ and a result graph $D = (\{\alpha, \beta, \gamma, \sigma\}, E_D)$. The distance between the vertices is indicated by the number next to the respective edge. In this example, all vertices are members of the same class. Therefore, each vertex in Q can be mapped onto each vertex in D.

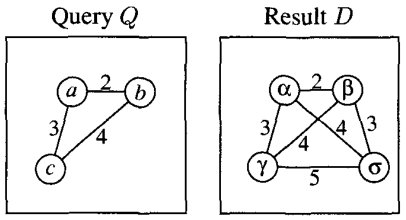

Figure 2. Sample Query and Result

We want to find all subgraphs in D that satisfy query Q, when *ssl*=4. First, we describe a short run of function *getIsoSubgraphs*. We choose $q_1 = a$ as the starting vertex and build the initial set $S^{(1)} = \{ (\langle a \rangle, \langle d \rangle) \mid d = \alpha, \beta, \gamma, \sigma \}$. Then we choose $q_e = b$ in the main loop. For both isomorphic subgraphs $G = (\langle a \rangle, \langle \alpha \rangle)$ and $G = (\langle a \rangle, \langle \beta \rangle)$, *validExt* returns only one extension pair. The remaining two values for G, i.e., $(\langle a \rangle, \langle \gamma \rangle)$ and $(\langle a \rangle, \langle \sigma \rangle)$, do not lead to any more extension pairs and are excluded from further processing. Therefore, $S^{(2)}$ contains only two elements, when we reach the third iteration where q_e is set to c. Each pair in $S^{(2)}$ also has only one extension pair and so we get two elements in $S^{(3)}$, which is returned as the result.

This is a short run of *getIsoSubgraphs* because many subgraphs are excluded early in the process; e.g., all subgraphs where a is mapped onto γ or σ are excluded in the first iteration. The reason for these early exclusions is that we chose a, whose outgoing edges are the most restrictive ones, as the first vertex.

If we start with $q_1 = c$ and continue with $q_e = b$ in the main loop, the running time of *getIsoSubgraphs* is longer. For both pairs $(\langle c \rangle, \langle \alpha \rangle)$ and $(\langle c \rangle, \langle \beta \rangle)$ the function *validExt* returns three valid extensions, while for the pairs $(\langle c \rangle, \langle \gamma \rangle)$ and $(\langle c \rangle, \langle \sigma \rangle)$ it returns two valid extensions, respectively. All together we get ten elements in $S^{(2)}$ and only two possible subgraphs are excluded, because $\text{dist}(\gamma, \sigma) > \text{dist}(c, b)$. So, in the third iteration where q_e is set to a, we get ten calls to *validExt* compared with only two in the short run. We call this a long run, because most of the invalid subgraphs are excluded in the last iteration.

5. Concluding Remarks and Future Work

The computation of subgraph isomorphism is known to be NP-complete [4]. Therefore, we cannot expect to find an efficient algorithm that solves our problem fast in all cases. Future work consists of exploiting the spatial constraints and finding more strategies to reduce the search space in addition to the ones we described.

References

[1] S. K. Chang, Q. Y. Shi, and C. Y. Yan. Iconic indexing by 2-D strings. *IEEE PAMI*, 9(3):413–428, May 1987.

[2] A. Del Bimbo and P. Pala. Visual image retrieval by elastic matching of user sketches. *IEEE PAMI*, 19(2):121–132, Feb. 1997.

[3] M. J. Egenhofer. Query processing in spatial-query-by-sketch. *Journal of Visual Languages and Computing*, 8(4):403–424, Aug. 1997.

[4] R. Englert and J. Seelmann-Eggebert. P-subgraph isomorphism computation and upper bound complexity estimation. Technical Report IAI-TR-97-2, Institute of Computer Science III, University of Bonn, Jan. 1997.

[5] C. Faloutsos, R. Barber, W. Equitz, M. Flickner, W. Niblack, and D. Petkovic. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, pages 231–62, 1994.

[6] B. T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE PAMI*, 20(5):493–504, May 1998.

[7] D. Papadias and T. K. Sellis. A pictorial query-by-example language. *Journal of Visual Languages and Computing*, 6(1):53–72, Mar. 1995.

[8] A. Pentland, R. W. Picard, and S. Sclaroff. Photobook: Content-based manipulation of image databases. In *Proceedings of the SPIE, Storage and Retrieval of Image and Video Databases II*, volume 2185, pages 34–47, San Jose, CA, Feb. 1994.

[9] A. P. Sistla, C. Yu, and R. Haddad. Reasoning about spatial relationships in picture retrieval systems. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *20th Intern. Conf. on Very Large Data Bases*, pages 570–581, Santiago, Chile, Sept. 1994.

[10] A. Soffer and H. Samet. Pictorial query specification for browsing through spatially referenced images databases. *Journal of Visual Languages and Computing*, 9(6):567–596, Dec. 1998.

[11] A. Soffer and H. Samet. Query processing and optimization for pictorial query trees. In D. P. Huijsmans and A. Smeulders, editors, *3rd Intern. Conf. on Visual Information Systems*, pages 60–67, Amsterdam, The Netherlands, June 1999. (Also Springer-Verlag Lecture Notes in Computer Science 1614).