Retrieving Video Data via Motion Tracks of Content Symbols

Tim T. Y. Wai and Arbee L. P. Chen*

Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 300, R.O.C.
Email : alpchen@cs.nthu.edu.tw

Abstract

Motion is the major feature that differentiates a video from a still image. In content-based retrieval, the motion track of a symbol object can be used as an index of video databases. Query processing based on such index can be treated as a curve matching problem. Two important criteria for solving the problem are sub-matching and approximate-matching. In this paper, two query processing approaches are proposed. The first approach expands the motion track index to a region and allows a user to query against this region. In the second approach, both the motion track index and the query curve are modeled as a combination of peaks and each peak is coded according to its orientation, angle and temporal information. The curve matching problem is then converted into a string matching problem, which is solved efficiently by a new finite automata based method. The proposed string matching algorithm is also shown to be scalable to the growth of the size of the database.

1. Introduction

With the improvement of the computer hardware, multimedia data have been widely applied in many realms. As a result of this, multimedia databases have gradually become an important research issue. Owing to rich information of multimedia data, queries can be specified not only by keywords or annotations, but also by its content. Consequently, content-based retrieval is significant for supporting powerful queries in multimedia databases.

To achieve the goal of content-based retrieval, special indexes for a video database should be established. The symbol objects appearing in the video and the motion tracks of the symbol objects can be used to build the indexes. Users can retrieve a *video sequence* [8] by the symbol objects and/or the motion tracks.

Motion track is the trajectory of a symbol object within one shot of the video. Every point in the motion track represents the location of the object appearing in a frame. Many researchers focus on automatic video indexing based on motion track. However, there are very few papers discussing query processing upon motion track indexes. A user can query the video database by specifying a curve, called *query curve*. Then the task of the query processing is to match the query curve to the motion track index.

There are two requirements in matching the motion track index. The first one is sub-matching. It means the user's query curve can be a subpart of the index. It is important because in most situations, users cannot precisely specify the starting and ending of the motion. The other requirement is the approximate matching of the query, since it is not reasonable for the user-specified query curve to perfectly match the index. Moreover, it is desirable that the similarity of the user's query curve and the motion track index be quantified.

We propose two query processing approaches according to the motion track index. In the first approach, we treat the motion track index as a region. In this way, the goal of approximate matching can be reached. The similarity degree of the matching is defined according to the variety of the region.

The second approach involves modeling a motion track as a string of codes. The motion track is modeled as a sequence of peaks. By coding each peak according to its orientation, angle and temporal information, the motion track is transformed into a string of code triples. An efficient finite automata based matching method is then proposed for efficient query processing.

This paper is organized as follows. In section 2, some related work is presented and our approaches are motivated. In section 3, the motion track index is expanded to a region and an associated matching method is proposed. In section 4, the query processing problem is converted into the curve matching problem. A new coding scheme and a matching approach are proposed. Experiments are also developed to show the efficiency and scalability of the approach. The prototype system on this work, called Venus is described in sectionr 5. We conclude this paper and

---

* To whom all correspondence should be sent.

point out our future work in section 6.

2. Related Work

Since a motion track is contained within a shot, in order to extract the motion track from a video we propose a shot change detection method on MPEG compressed video data in [5]. And then in [7], by the property of uniform *motion vectors* existing in the same symbol object in MPEG video data, a method to detect moving symbol objects is developed. After examining all motion vectors of a symbol object between the frames, we can obtain its motion track. It is stored as an index of the video in the video database.

In [2], motion tracks are extracted by the information used in rendering the animation. The model-based approach computes the optical flow field and repeatedly applies the least square method. Since this approach takes advantage of some information gained from the process of animation rendering, it has good performance.

In [3] and [4], the indexes of video databases have been extended in the form of a relational tuple *[Object, Motion, Video sequence]*. "*Object*" represents the symbol object in the video, "*Motion*" represents the motion track of the symbol object, and "*Video sequence*" represents a sequence of continuous frames of a video. By this extension, a user can freely query one or two of the three fields by the other. In these papers, the motion analysis is divided into three levels: motion trajectory based on rigid objects, motion trajectory based on non-rigid objects, and motion trajectory based on semantic levels. The representation of the motion track is based on four traditional schemes: (1) point representation, (2) B-spline representation, (3) chain code, and (4) differential chain code. Five trajectory matching functions are proposed in these papers. However, only the description of how the trajectory matching should be satisfied. No matching method is proposed.

Query processing upon motion track indexes is similar to similarity search of the time-series databases. In [11], a new segmentation method is proposed. A time-series sequence is transformed into a regular expression by means of this segmentation. However, there is no description about how to match two such regular expressions in the paper.

In [1], a time-series sequence is divided into several *sub-sequences*. After matching the sub-sequences of two time-series sequences separately, the similarity of the two sequences is computed by stitching the matching results of the sub-sequences. Amplitude scaling, offset translation and

non-matching gaps are allowed in this approach. However, it is not adequate for motion track matching, since no sub-matching is addressed.

In [6], with no motion track extraction algorithm proposed, the motion track representation is classified into four classes. It contains traditional chain code representation, rotation by clockwise or counter-clockwise, and rotation and translation in depth in the consideration of the 3-D model. Each representation code is called a *motion symbol*. The motion symbols are classified as 16 symbols for coding. Two signatures corresponding to the motion track are introduced. One corresponds to the appearance of the motion symbols in the track, and the other corresponds to their appearing order. The goal of the constructed signature file is to filter out unneeded tracks when processing the query. However, when many identical motion symbols exist in the same track, the filtering effect will be greatly reduced. Furthermore, this approach allows only exact matching. In our approach, the similarity of the user's query and the motion track index can be quantified

In [12], a motion track matching method is proposed based on the index of chain code. An interesting new idea in this paper is that it refers to the conception of velocity (acceleration) and size evaluation of the object to the motion track. An exhaustive search is applied to map the index motion track to the query curve, even though the query string is shorter than the index string. Thus the requirement of sub-matching can not be achieved. Furthermore, the exhaustive mapping of the motion track matching is inefficient. In our peak model approach, the goal of sub-matching is accomplished with high efficiency.

3. The Index Region Approach

In this section, a motion track matching approach based on its geometric outline is proposed. The motion track matching problem can be treated as a curve matching problem. Unlike exact curve matching, we convert the motion track index into a region, called *index region*. Thus the goal of approximate matching can be reached.

3.1 Querying against Index Region

Previous work on curve matching focuses on exact matching of curves. In motion track matching, however, it is preposterous to expect that query curve completely coincides with the index. Consequently, approximate matching is an important consideration in motion track matching problem.

Consider the condition in Figure 1. Each point of the motion track index is represented by its

(x , y) pair of the coordinate. They are generated by finding the points of local maximum y-value or the local minimum y-value among each motion track in the database. The query curve is drawn by a user in the video query plane. To perform the motion track matching, we align the starting point of the query curve to the starting point of the index. Then we designate a value *d*, in order to quantify the similarity. After that, we expand the index to a region by separately adding *+d* and *-d* to its y-values as shown in Figure 2.

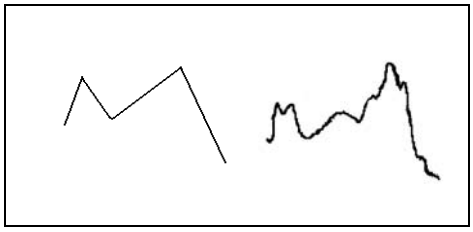

Figure 1 Motion track index and query curve

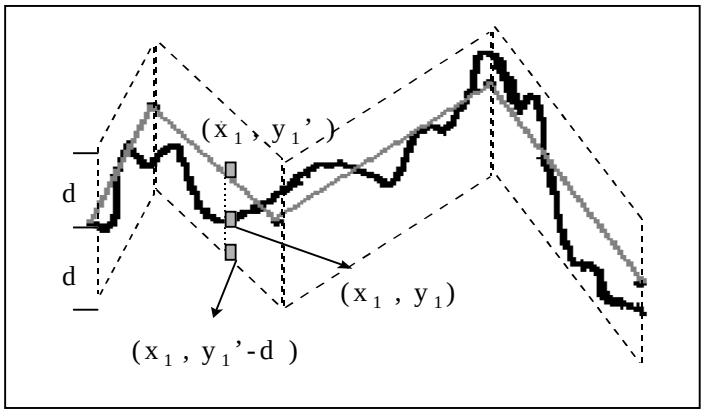

Figure 2 Querying by index region

Next, we check if the query curve is contained in the region. Suppose the *local maximum* and *local minimum points* of the query curve are $P_1(x_1 , y_1)$, $P_2(x_2 , y_2)$,..., $P_n(x_n , y_n)$. After aligning the starting point of the query curve to that of the index, let the mapping points in the index be (x_1 , y_1'), (x_2 , y_2'),...,(x_n , y_n'). For each y_i, if it satisfies $(y_i' - d) \le y_i \le (y_i' + d)$, the query curve is contained in the index region. The relation of these points is shown in Figure 2, and the query curve matches the index at the range of degree *d*.

3.2 Quantification of Approximation

To quantify the similarity of the motion track matching, we just use the procedure described

above at the degree of *d*, 2*d*, 3*d*,... until the query curve is contained in the index region at degree *kd*. Then the similarity of this matching is measured as *kd*. The larger the *k* is, the lower the similarity will be.

Nevertheless, there exists a problem. Consider the condition in Figure 3. The query curve is almost contained in the region of degree *d*, except one point P_2. If we use the measurement described above, it will be classified to the similarity of degree 2*d*. However, it resembles the region of degree *d* more. We refine the quantification in two ways to solve the problem.

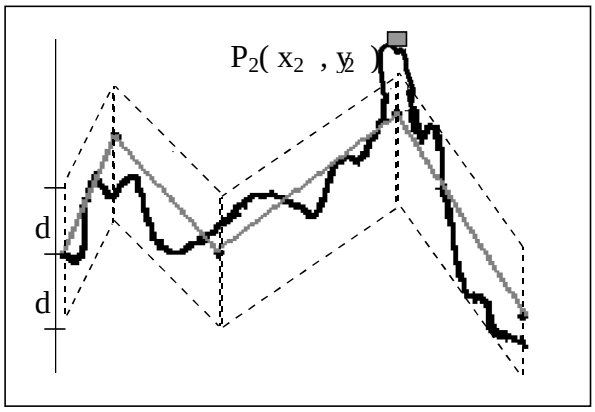

Figure 3 A special condition of query curve

In the first method we consider the proportion of the local maximum and minimum points that fall out of the region. If we have verified that all *n* local maximum and minimum points are contained in the region of degree *(k+1)d*, while *m* of which do not satisfy $(y_i' - kd) \leq y_i \leq (y_i' + kd)$, the similarity of this query is defined as $kd + (m/n)d$, which ranges from *kd* to *(k+1)d*.

In the second method, let E_i be the variance out of the region of degree *d* of the point P_i in the query curve. Then E_i is defined as :

If $y_i > y_i' + d$ then $E_i = y_i - (y_i' + d) = y_i - y_i' - d$

If $y_i < y_i' - d$ then $E_i = (y_i' - d) - y_i = y_i' - y_i - d$

else $E_i = 0$

Now, we define the average variance of the query curve out of the region of degree *d* as:

$$E_{av} = \left(\sum_{i=1}^{n} E_i \right) / n$$

Add up E_{av} to d, and then check if the sum is larger than 2d. If it is, change to test 2d and repeat the procedure described above. Otherwise the sum of $(E_{av} + d)$ is the similarity of this

query.

The index region approach possesses the property of *multi-resolution*. In the implementation, we allow users to specify the least degree of similarity in terms of *d*'s they require such that the matching time can be reduced.

4. The Peak Model for Motion Track Matching

The approach proposed in this section is based on the coding of the motion track. For a given motion track, a number of mechanisms exist for its coding. Below we list the most representative two (shown in Figure 4) :

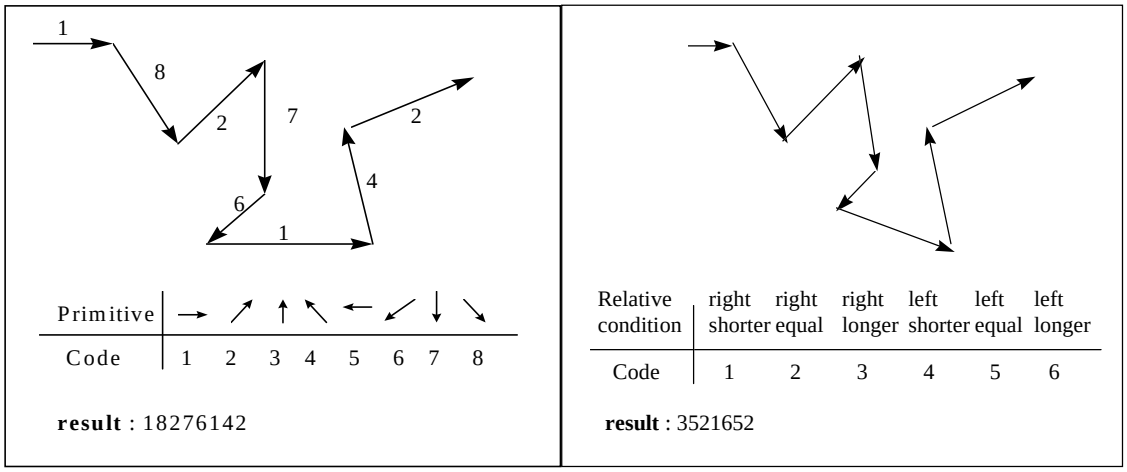

Figure 4 : (a) chain code (b) differential chain code

(1) Chain code : By using a set of orientation primitives in Figure 4(a), we can use a zig-zag line representation of the motion track can be used to generate the code.

(2) Differential chain code : Each line segment is coded relative to the last segment by the direction (left or right) and length [10] . An example is shown in Fig 4(b).

In this section, we propose a new coding approach and introduce the query processing mechanism based on this coding.

4.1 Modeling the Motion Track by Peaks

4.1.1 The Peak Model Coding

Owing to rich information of the motion track, neither the chain code nor the differential

chain code is suitable to represent the motion track. To preserve more information in the motion track, we model it by a sequence of peaks. A *peak* is defined by two connected edges of the motion track index. An example is shown in Figure 5. A motion track is coded by many peaks. The orientation of the peak is defined as the orientation of the angle bisector, and the angle of the peak is defined as the angle expanded by the two edges of the peak.

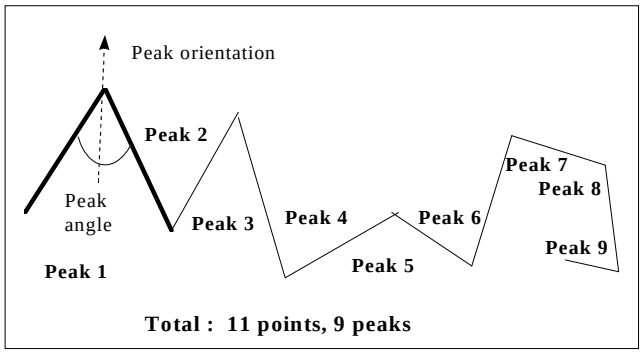

Figure 5 The peak model coding

To extract the peaks, each point of the motion track is examined and the points of the local maximum y-value or the local minimum y-value is detected. By connecting these points, the motion track is modeled by the combination of many peaks.

Each peak is represented as **(** *orientation* , *angle*, *time* **)**, by coding the orientation, angle, and temporal information separately, and the motion track is represented by an ordered list of peaks.

The orientation of the peak is coded according to the primitives shown in Figure 6(a).

The angle of the peak is coded by dividing 0° ~ 180° into 8 partitions, which is illustrated in Figure 6(b).

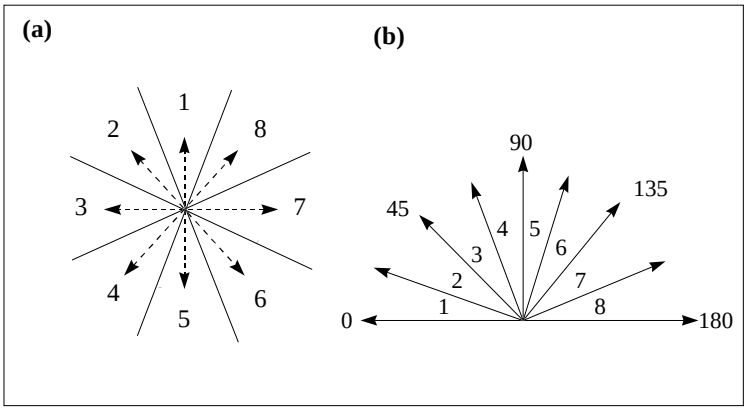

Figure 6 Primitives for coding peak orientations and angles

Peak	1	2	3	4	5	6	7	8	9
Orientation	1	5	1	4	1	6	2	8	6
Angle	3	3	2	4	6	3	4	5	4

Figure 7 The peak coding result of the motion track in Figure 5

Thus the orientation and angle of the motion track in Figure 5 can be coded as shown in Figure 7.

The symbol object of the video may stay in some point of a peak for some frames. It also takes some frames to move the symbol object from one point to the next point along an edge. Thus the temporal information "*time*" records the total number of frames in which the symbol object moves through the peak. Refer to the peak in Figure 8. If the symbol object stays in **Point 1** for 1 frame, **Point 2** for 3 frames and **Point 3** for 6 frames, and it takes 4 frames for the symbol object to move from **Point 1** to **Point 2**, and 2 frames from **Point 2** to **Point 3**, the temporal information "*time*" of the motion track index will record the number 1+4+3+2+6 = 16.

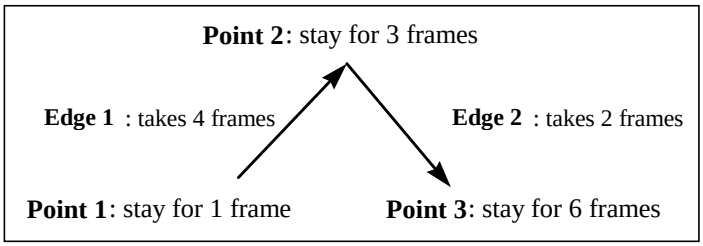

Figure 8 The temporal information for a peak

4.1.2 A Revised Peak Model Coding

It is easy to model the curve by peaks and to efficiently transform the query processing problem into the string matching problem (to be discussed in section 4.2.) However, there is a problem for the coding described in the above sub-section. Consider Figure 9, there are three peaks. At the first look of it, they are just slightly different. However, codes generated for them based on the scheme in 5.1.1 are (5 , 8) , (5 , 8) and (1 , 8) for the peak orientation and peak angle. By comparing the orientation of the peak first, peak 1 and peak 3 will be regarded as different, which is unreasonable.

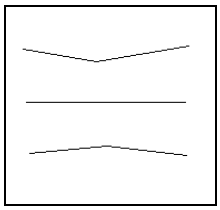

Figure 9 Three example peaks

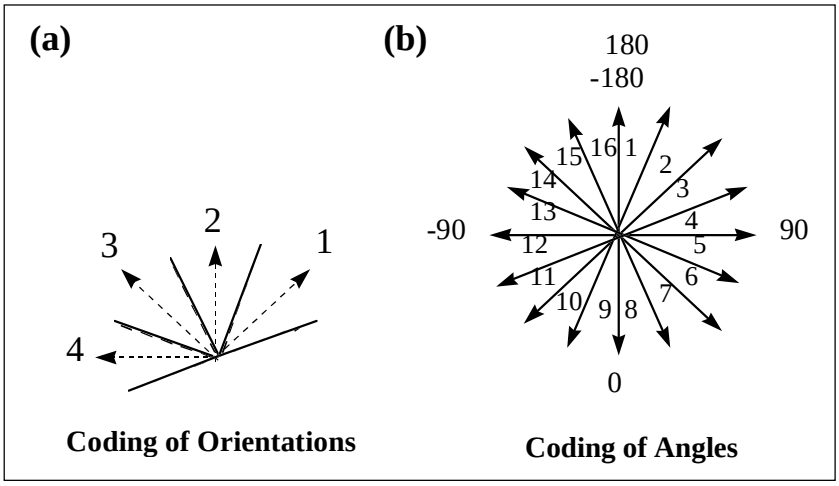

Figure 10 Modification of the peak orientation and angle primitives

We modify the codes of the orientation to the primitives shown in Figure 10(a). The other half orientations are represented by their associated opposite orientations, with the negative sign added to the angle. Thus the codes of the angle are modified as in Figure 10(b). It is coded by dividing -180 ° ~ 180° into 16 partitions, with 180° and -180° being aligned. By this coding scheme, the three peaks in Figure 9 will be coded as (2 , 1) , (2 , 1)(or (2,16)), and (2 , 16).

We can see that the orientation code of the three peaks are the same, but the angle codes are not. To quantify the difference between two peak angle codes a_1 , a_2 , we define it as :

if $a_1 < a_2$ *then* $(a_1 - a_2) \bmod 16$

if $a_1 > a_2$ *then* $16 - (a_1 - a_2) \bmod 16$

Thus the angle difference between the three peaks is

$(1 - 16) \bmod 16 = (-15) \bmod 16 = 1$

or $16 - (16 - 1) \bmod 16 = 16 - 15 = 1$

The angle difference between the peaks is at most 1, and they have the same orientation, which reflects the close similarity among them.

4.2 Query Processing

A user can query the motion track index by clicking the points as the query curve, which will be introduced in the system prototype, called Venus in section 5. The zig-zag line representation of the query curve is generated by connecting these points in order. Performing the coding in the same way as it is performed on the index, we will get a code of the orientation and the angle for each peaks in the query curve. Therefore, the motion track query processing problem is converted into a string matching problem.

A user can choose whether he or she wants to query the temporal information or not. If the temporal information is supposed to be queried, a user should move the mouse pointer to each point and each edge to specify how many frames it takes, by using Venus system prototype. Refer to Figure 11. A user can specify that the symbol object stays in the three points for 1, 4 and 5 frames and moves along the two edges for 3 and 3 frames. Thus the query temporal information of this peak is computed as 1+3+4+3+5 = 16 frames, which is matched with the temporal information of the peak shown in the example of Figure 8. Like peak orientations and peak angles, comparing the query temporal information with the index is also a string matching problem.

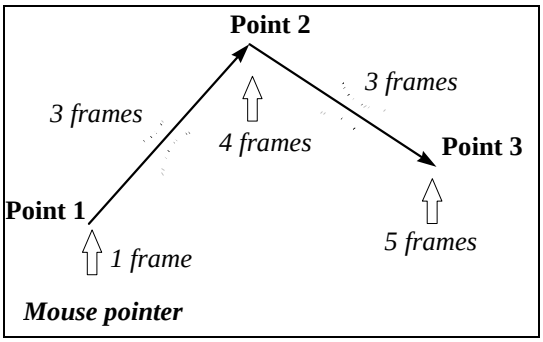

Figure 11 A query curve with temporal information

Section 4.2.1 introduces a finite automata based approach to efficiently achieve string matching. It is utilized when matching a single motion track index. In section 4.2.2, all motion track indexes are interposed in the finite automata. Two additional data structures are introduced to speed up the matching when using multiple motion track indexes in the same finite automata.

4.2.1 Single Motion Track

After both the motion track index and the query curve are transformed into codes, the curve matching problem is converted into string matching problem. To satisfy the sub-matching

requirement when querying the video database, it is necessary to accomplish sub-string matching. Taking advantage of the property of the video, we propose a query processing algorithm based on finite automata.

The algorithm is illustrated by an example shown in Figure 12. Figure 12(a) is the index and Figure 12(b) is the query. We want to find the video sequence with codes like 2, 1, 3, 7. First, we store the index in Fig 12(a) in a finite automata, shown in Figure 12(c).

Suppose the value of the codes ranges from 1 to 8. There are 8 states in this finite automata. Each state represents the code value 1,2, ... ,8. The transition from one state to another records the current peak number of the video. In other words, if there is a transition with peak number *f* from state *x* to state *y* (x , $y \in \{1,2,...,8\}$), it means that the *f-th* peak is coded as *x* and the *(f+1)-th* peak is coded as *y*. For example, there is a transition from state 1 to state 5 with peak number 9 in Figure 12(c), it means that the 9^{th} is coded as 1 and the 10^{th} peak is coded as 5, as Figure 12(a) shows.

(a) index

peak number	1	2	3	4	5	6	7	8	9	10	11
Code	4	2	1	3	7	6	8	2	1	5	6

(b) query

2 1 3 7

(c)

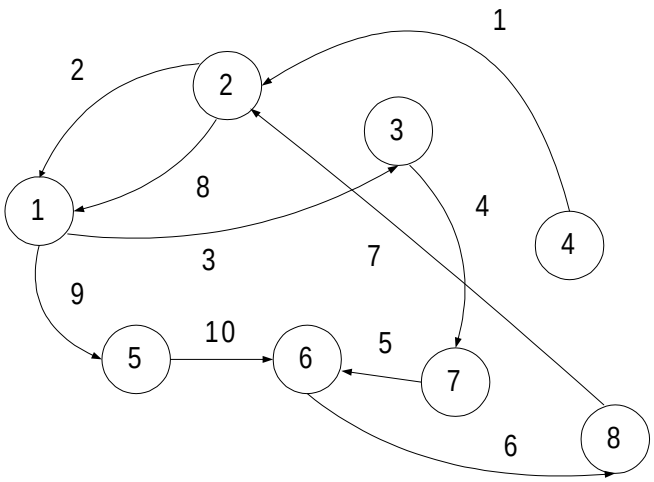

Figure 12 Sub-string matching example

After building the finite automata, the matching procedure begins to process the query in Figure 12(b). Since there are four codes in the query, it needs four steps. For the first step, one

filter is applied to filter out some unneeded transitions and to produce *candidate transitions*. For the other steps, two filters are applied. To measure the performance of the matching, the operation counts are computed in each step. The four steps are described as follows:

Step 1 : The first code of the query is 2. Then we inspect state 2 in the finite automata. There are two transitions from state 2, one to state 1 with peak number 2, and the other also to state 1 with peak number 8. A filter named *destination filter* is applied in this step. The destination filter checks whether the destination state of each transition equals the next code in the query. If so, it becomes the candidate transition and we can go to step 2. Otherwise, discard the transition. In this example, the destination state of both transitions is 1, which equals to the next query code. Thus both transitions are able to go to step 2.

For each of the 2 transitions, such a check is needed. Thus the operation count of this step is 2.

Step 2 : Now the current code of the query is 1. There are two transitions from state 1 , one to state 3 with peak number 3, and the other to state 5 with peak number 9. At this moment, we apply two filters. The first one is the destination filter, which is the same as we applied in step 1. The next query code is 3. We need to test if 3 is the destination state of the transition. The transition with peak number 3 passes the test and the other is discarded. The second filter named *continuity filter* is designed to check the continuity of the two sequential peaks. When applying the continuity filter, we have to check the peak number of the passed transitions from the destination filter to see if it is the subsequent number of some candidate transition derived in the last step, i.e., step 1. If not, discard this transition. In this example, the transition goes through the continuity filter because 3 is the subsequent number of 2 which is the peak number of the candidate transition in step 1.

For the first filter, 2 transitions need be checked. Thus 2 operation counts are needed. When it comes to apply the continuity filter, only one transition left. For this transition, continuity with the 2 candidate transitions of the last step, step 1, are checked. Two operation counts are needed in this filter. Therefore, this step takes 2 + 2 = 4 operation counts.

Step 3 : Now the current code of the query is 3. There is only one transition from state 3. After applying both filters, the transition passes and goes to state 7. In the first filter, this transition is checked for its destination. Thus 1 operation count is needed. In the second

filter, the continuity with the only candidate transition of the last step, step 2, is checked. One operation count is needed in this filter. Therefore, this step takes 1 + 1 = 2 operation counts.

Step 4 : The current code of the query is 7, which is the last one. We find that there is only one qualified path of transitions, with peak numbers 2, 3, 4. Therefore, the result is the sequence of the < 2, 3, 4, 5 > peaks. No operation count is needed.

The whole sub-string matching problem needs 2 + 4 + 2 = 8 operation counts. If we apply a *brute force* method to match the query 2137 and the index 42137682156, as shown in Figure 13, the worst case takes $4 \times (11-4+1) = 32$ operation counts (if the answer is the last four codes of the index).

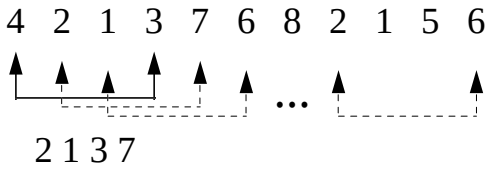

Figure 13 brute force sub-matching

A formal description of the query processing algorithm is shown in the Appendix.

4.2.2 Multiple Motion Tracks in Databases

There is only one track in the finite automata described in section 4.2.1. In a video database, many index tracks exist, which are labeled by *Track ID*. Each Track ID represents a motion track of a symbol object. To make query processing efficient, we store all of the motion track indexes in the same finite automata. Each transition is given additionally an extra attribute, Track ID. The two filters described in section 4.2.1 should be modified accordingly.

For the first one, destination filter, it is still necessary to check if the destination state of the transition equals to the next query code. For the above example query 2137, we should check all transitions from state 2 to see if its destination state is 1. If the number of transitions from state 2 equals n, it costs O(n) to execute the destination filter, which is inefficient. To avoid this exhaustive check, we build a *search index tree* beforehand for every state in the finite automata. The search index tree for state 2 is shown in Figure 14.

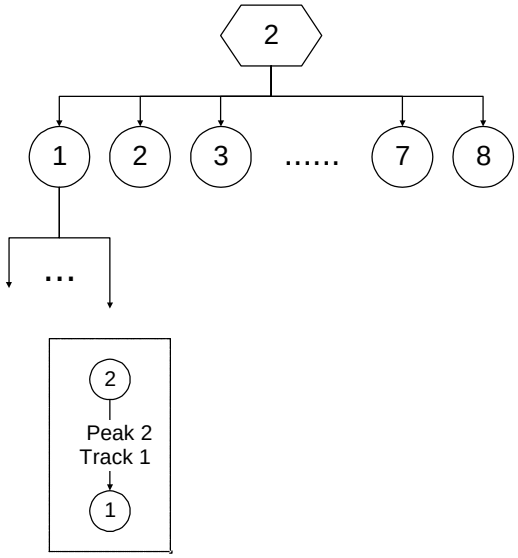

Figure 14 The search index tree for state 2

The root of the search index tree is 2. There are 8 children nodes of the root, which represents the 8 states in the finite automata. All transitions from state 2 to state j (j = 1,…,8) are recorded as the children nodes of j. We call these transitions the *transition cluster* from 2 to j. For example, the transition from state 2 to state 1 with fame number 2 and Track ID 1 is recorded as a child of node 1 in the search index tree of 2. Consequently, we no longer need to check all other transitions from state 2 to see if their destination state is 1. By looking up node 1 of the search index tree for state 2, all of the transitions from state 2 to state 1 can be obtained. The goal of the destination filter is then achieved in a constant number of operation counts, not O(n).

For the continuity filter, in addition to checking the continuity of the peak number of two transitions, we need to check if the two transitions belong to the same track, i.e. with the same Track ID. However, if the number of transitions increases, it is inefficient to apply this filter. We illustrate it by an example. The query is the same as before, 2137, and there are many tracks in the finite automata. Suppose we have already applied the first filter to check the destination state of each transition in the four steps. The candidate transitions left for each step is shown in Figure 15.

For example, when checking the transition from state 1 to state 3 with Peak 3 and Track 1, we must scan each transition from state 2 to state 1 (in this example, there are 6 such transitions) to check which one is with Peak 2 (to achieve the continuity of the two sequential peaks) and Track 1 (within the same track). There are 5 transitions from state 1 to state 3 and 6 transitions from state 2 to state 1. Therefore, in step 2 (check from state 1 to state 3), it takes $5 \times 6 = 30$

times to apply the continuity filter in worst case.

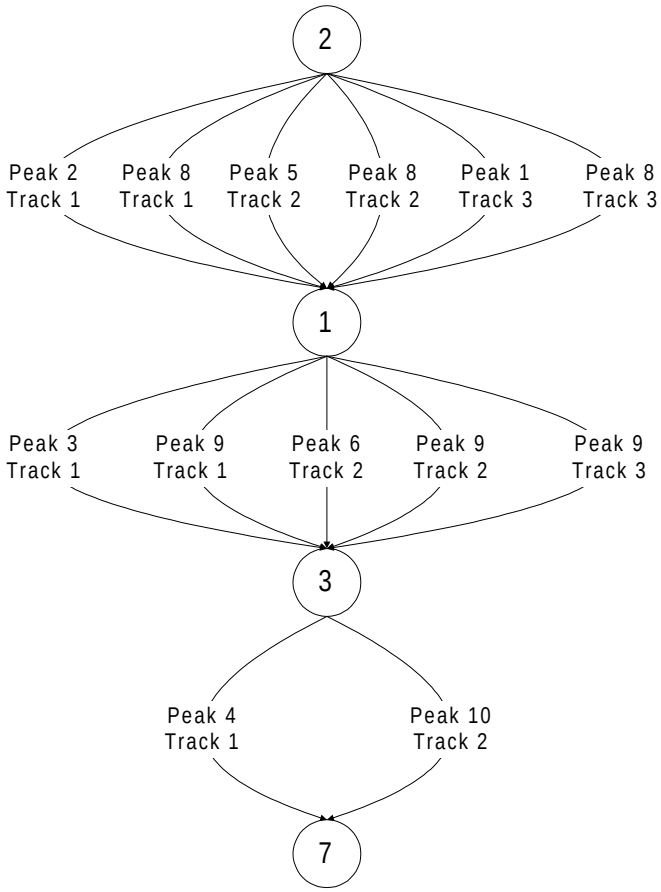

Figure 15 The candidate transitions after the destination filter

We can sort the transitions on (Track ID, Peak number) in the search index tree beforehand to improve efficiency. Using the same example, Figure 16 shows the search index tree after sorting the transitions. We proceed in the second step. All the transitions produced from the destination filter in this step (transitions from state 1 to state 3) are treated as the first list, while all the candidate transitions after the last step (transitions from state 2 to state 1) are treated as the second list. Since the two lists are sorted, the conception of merge sort can be applied to match the subsequent transitions among them.

Two pointers are utilized. Each points to a list of transitions, as shown in Figure 16. The two pointers move toward right in conformity with the merge sort scheme. With the movement of the pointers the continuity of the two transitions pointed to by the two pointers are checked and the unqualified transition is discarded.

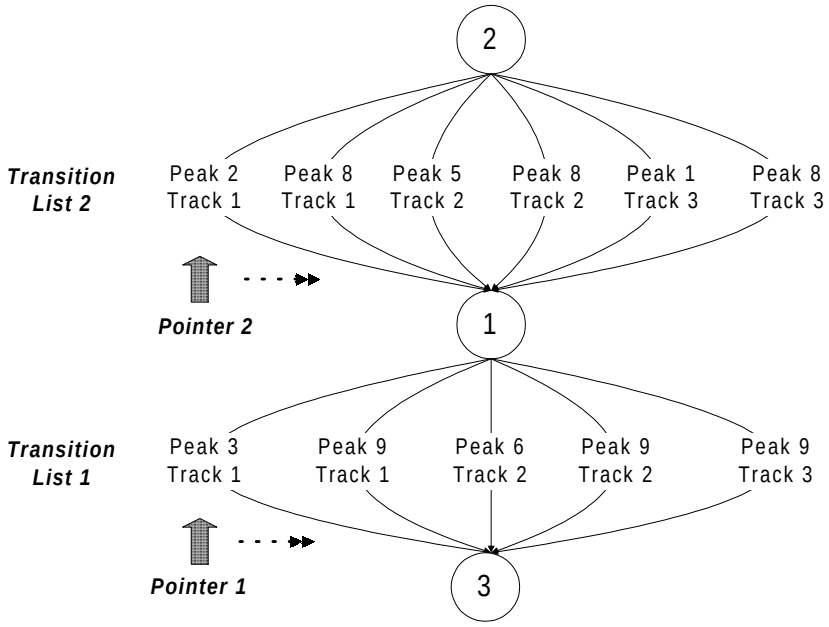

Figure 16 Applying merge sort

If there are n candidate transitions the destination filter of this step produces and m candidate transitions the last step produces, the time complexity happens when both pointer move to the last transition of the list, which is O(m+n). Note that the complexity of the original approach without sorting is O(m×n).

Let the length of the query curve be *p*, and *n(i)* represents the number of candidate transitions the destination filter of the i-th step produces. The time complexity of the query processing is O($\sum_{i=1}^{p} n(i)$). By using the search index tree, *n(i)* represents the number of a transition cluster. If all the transitions are normally distributed in the search index tree, *n(i)* is about $\frac{p}{8\times 8}$ the total transition number. Since *p* may not be so large, it means that we just need to search a small portion of all the transitions.

4.3 Experiments and Result Analysis

When modeling the motion track by peaks, the curve matching problem is converted to the string matching problem. The time complexity of the proposed string matching algorithm is analyzed in section 4.2.2. In this section, We develop four experiments to examine the execution performance under the variation of the track number in the database, the query processing

strategies, the query string length and the index string length. The experiments are all performed on SUN SPARC station ELC, with SUN OS 4.1.1.

4.3.1 Variation of the Track Number

At first, we design an experiment is performed to examine the relationship between the execution time and the track number in the database. One motion track is stored as a string in a file. The size of the search space is 8, and the string length is 20. To test the sub-string matching, the length of the query string is set to be 4. We gradually increase the number of the tracks in the database and inspect the execution time. The proposed string matching algorithm is compared with an UNIX built-in utility *grep*. The experiment result is shown in figure 17.

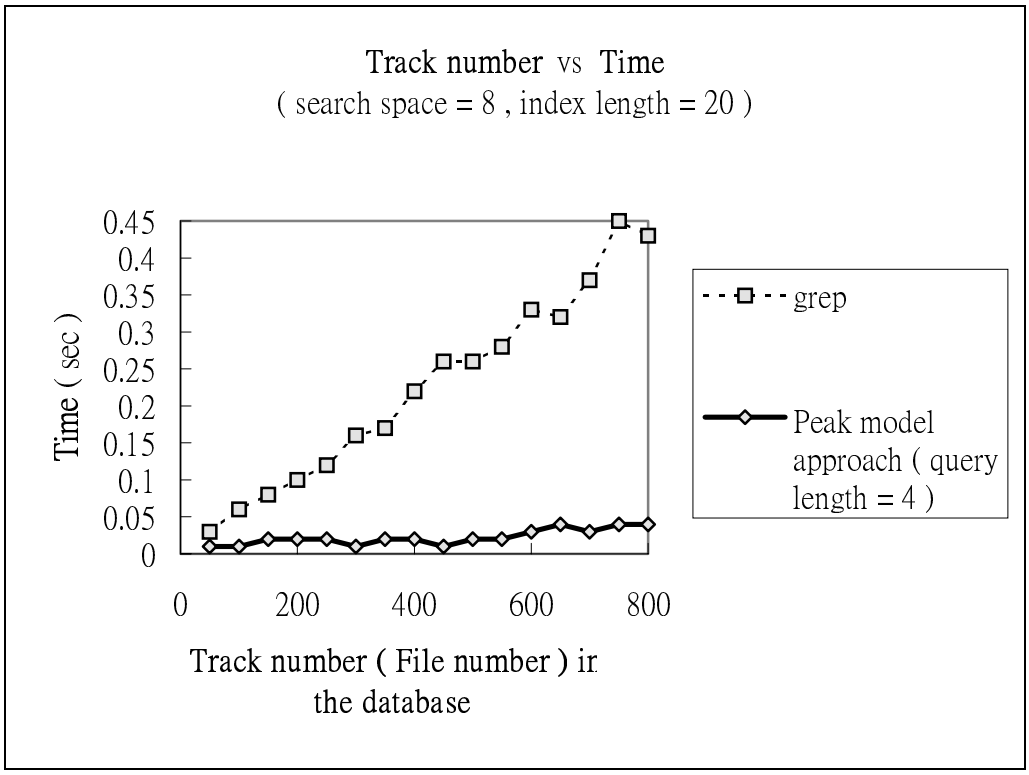

Figure 17 Compare with *grep* by execution time

It takes the proposed string matching algorithm some pre-processing time to construct the finite automata (or search index tree) described in section 4.2.1 and 4.2.2. By deducting the pre-processing time, we found that the matching time grows slowly with the growth of the track number in the database. This is because the increased transitions of the tracks are distributed to each transition cluster. From the point of view of time complexity, it means that the growth of the

n(i) is slow.

4.3.2 Variation of the Query Processing Strategies

The result of the query processing can be obtained in two ways. In the first way, the string matching algorithm is applied to the angle, orientation, and temporal information separately to produce three sub-results. The final result is generated by intersecting the three sub-results. However, the string matching algorithm needs to be performed 3 times. Furthermore, it requires an additional cost to process the intersection.

In the second way, we combine the three coding schemes together. For example, if we code each angle, orientation and temporal information using 3 bits (the search space : $2^3 = 8$), it takes $3 \times 3 = 9$ bits to represent each peak. The size of the search space is 2^9 (= 512). Thus the string matching algorithm needs to be performed just once, and no intersection time is needed. We will show in the second experiment that based on our string matching algorithm, the matching time is slightly decreased when the search space changes from 8 to 512. Therefore, by combining the three coding schemes together, the query processing time turns out to be less than one third of that using the first way.

The second experiment is designed to observe the effect of the variation of the search space. Assume the database contains 800 tracks and the length of each is 20. The length of the query string is 4. The experiment result is shown in figure 18.

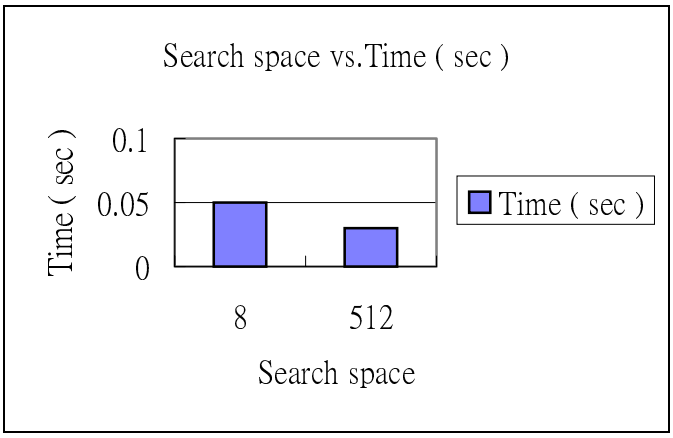

Figure 18 Variation of the search space

The experiment is performed on two sizes of the search space. The growth of the search space causes the increase of the number of the transition clusters. If the number of the transition

clusters is larger, the number of transitions of each transition cluster *n(i)* will be smaller, since the total number of the transitions is constant. The execution time decreases if *n(i)* is smaller. The effect is illustrated in the experiment result.

4.3.3 Variation of the Query String Length

We want to check the effect of the variation of the query string length. It is shown in figure 19. The database contains 800 tracks, the search space is 8 and the index string length is 20.

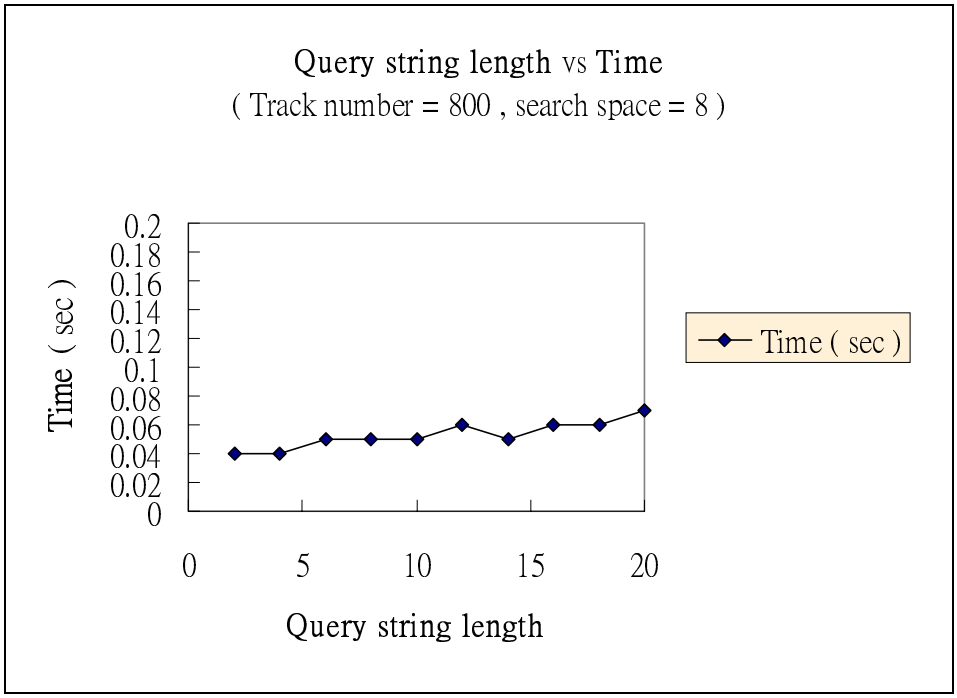

Figure 19 Variation of the query string length

The execution time increases with the increase of the query string length *p*. From this experiment we found that the variation of the query string length only slightly affects the execution time.

4.3.4 Variation of the Index String Length

We want to check the influence of the index string length. It is shown in figure 20. The database contains 800 tracks, the search space is 8 and the query string length is 4.

The longer the index string length, the more the transitions in the search index tree. Thus the execution time increases with the growth of the index string length. From this experiment we

found that the variation of the index string length only slightly affects the execution time.

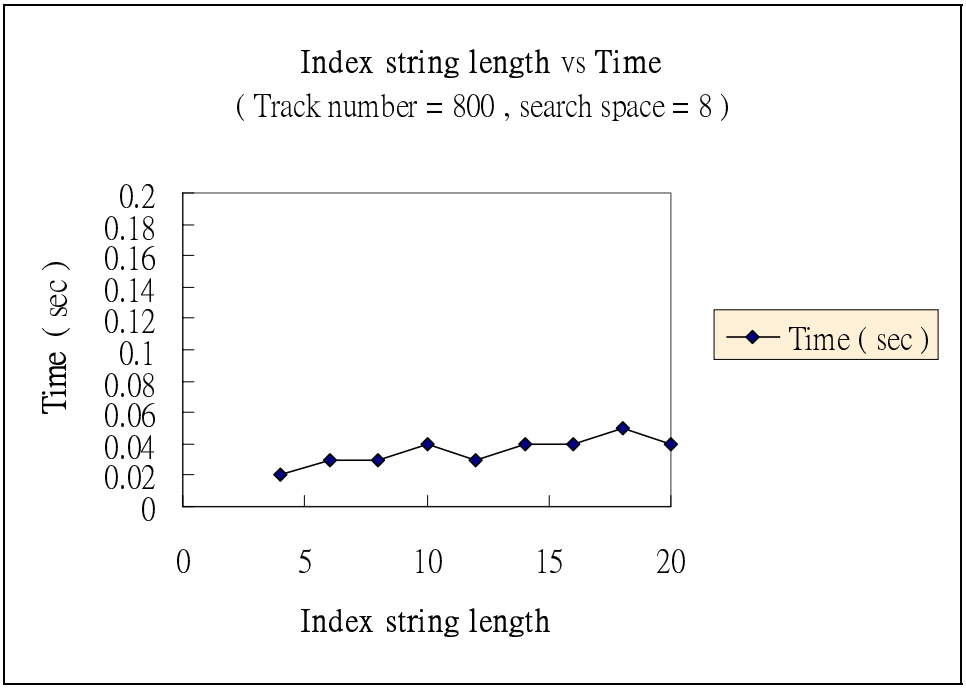

Figure 20 Variation of the index string length

4.4 A Similarity Model

It is more desirable if the similarity is quantified in the peak model approach. As the discussion in 4.3.2, combining the coding schemes together results in better performance. Because of the affinity between the angle and orientation of the peak, the proposed similarity model is built on the combination code of the angle and the orientation of the peak, which is called *code pair*.

At first, the similarity of the code pairs should be defined. Take Figure 6 as an example. Both code range from 1 to 8. The similarity of a code is defined according to the extent of the nearness to the other code. Suppose (1 , 4) represents the code pair of the angle coded by 1 and orientation coded by 4. The similarity of (1 , 4) to the other code pairs is defined by the table as follows :

Similarity	Code pairs
8 / 8	(1 , 4)
7 / 8	(1 , 3) (1 , 5) (2 , 4) (8 , 4)
6 / 8	(2 , 3) (2 , 5) (8 , 3) (8 , 5) (3 , 4) (7 , 4) (1 , 2) (1 , 6)

5 / 8	(2 , 2) (2 , 6) (8 , 2) (8 , 6) (3 , 3) (7 , 3) (3 , 5) (7 , 5) (4 , 4) (6 , 4) (1 , 1) (1 , 7)
4 / 8	(1 , 8) (5 , 4) (2 , 1) (2 , 7) (8 , 1) (8 , 7) (4 , 3) (6 , 3) (4 , 5) (6 , 5) (3 , 2) (3 , 6) (7 , 2) (7 , 6)
3 / 8	(2 , 8) (8 , 8) (5 , 3) (5 , 5) (3 , 1) (3 , 7) (7 , 1) (7 , 7) (4 , 2) (6 , 2) (4 , 6) (6 , 6)
2 / 8	(3 , 8) (7 , 8) (5 , 2) (5 , 6) (4 , 1) (4 , 7) (6 , 1) (6 , 7)
1 / 8	(5 , 1) (5 , 7) (4 , 8) (6 , 8)
0	(5 , 8)

When matching a list of code pairs, the similarity is measured based on the table. Suppose the query is (1 , 4) (2 , 3) (1 , 4) and the candidate transitions after applying the destination filter is shown in Figure 21. When applying the continuity filter, the original exact matching produces two candidate answers : (Track 1, Peak 2) → (Track 1, Peak 3) and (Track 1, Peak 8) → (Track 1, Peak 9). They both satisfy (1 , 4) → (2 ,3) and (2 , 3) → (1 ,4). The similarity of them is 1.00×1.00=1.00. Consider the transition from state (1 , 3) to state (2 , 3), it satisfies the continuity filter for (Track 2, Peak 7) → (Track 2, Peak 8). However, the first transitions is (1 , 3) → (2 , 3), but not (1 , 4) → (2 , 3). Therefore the similarity is (7 / 8) × 1.00 = 0.875. The matching similarity can be measured by multiplying the similarity in each step.

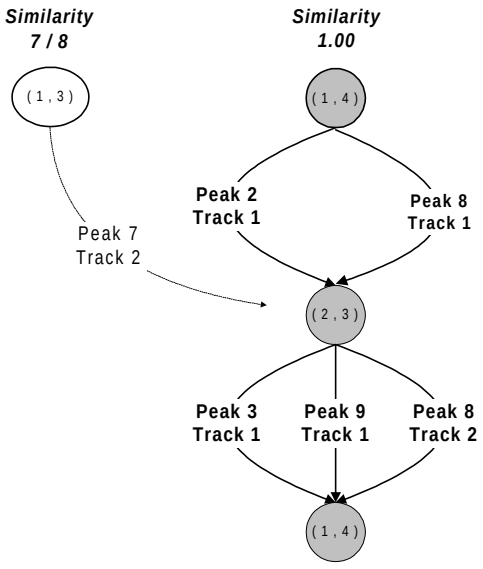

Figure 21 Similarity specification

5. System Implementation and User Interface

To support content-based video retrieval, we have started a project called Venus [6] on X-window systems by X11R5 and Motif library. The main functions that the Venus video database provides are video indexing, video query interface, query language (called CVQL) and video query processing. When a video is inserted into the video database, the motion tracks of a symbol object is extracted semi-automatically. For example, figure 22 shows three frames of a video to be indexed. By applying suitable symbol object detecting algorithm as in [7][10], we can find the positions of a tree in each of the these frames. Then the motion tracks of the two symbol objects can be constructed by connecting the corresponding positions of the symbol objects in each frame.

Figure 22 A video example

Figure 23 shows a video query interface we have implemented in Venus. It consists of four parts. The video browser provides structural video class hierarchy for a flexible search space. The symbol browser provides icons for users to choose for the specification of the symbol objects in video queries. The CVQL editor let users to specify the query. The motion track matching plane is the place in which users can specify their query curve.

Assume a user wants to retrieve the videos similar to the video shown in Figure 22. To specify this video query, a tree icon is selected from the symbol browser. As the result of the camera operation, the appearance of the tree will move from the lower-right corner to the lower-left corner. We can draw a motion track by clicking the points of the curve. When querying by clicking the points, the Venus system will simulate the query curve by the B-spline representation. The video query is shown in Figure 23.

Both the index region approach and peak model approach are implemented on the Venus system. After specifying the query curve, a user can choose either approach to achieve motion track matching.

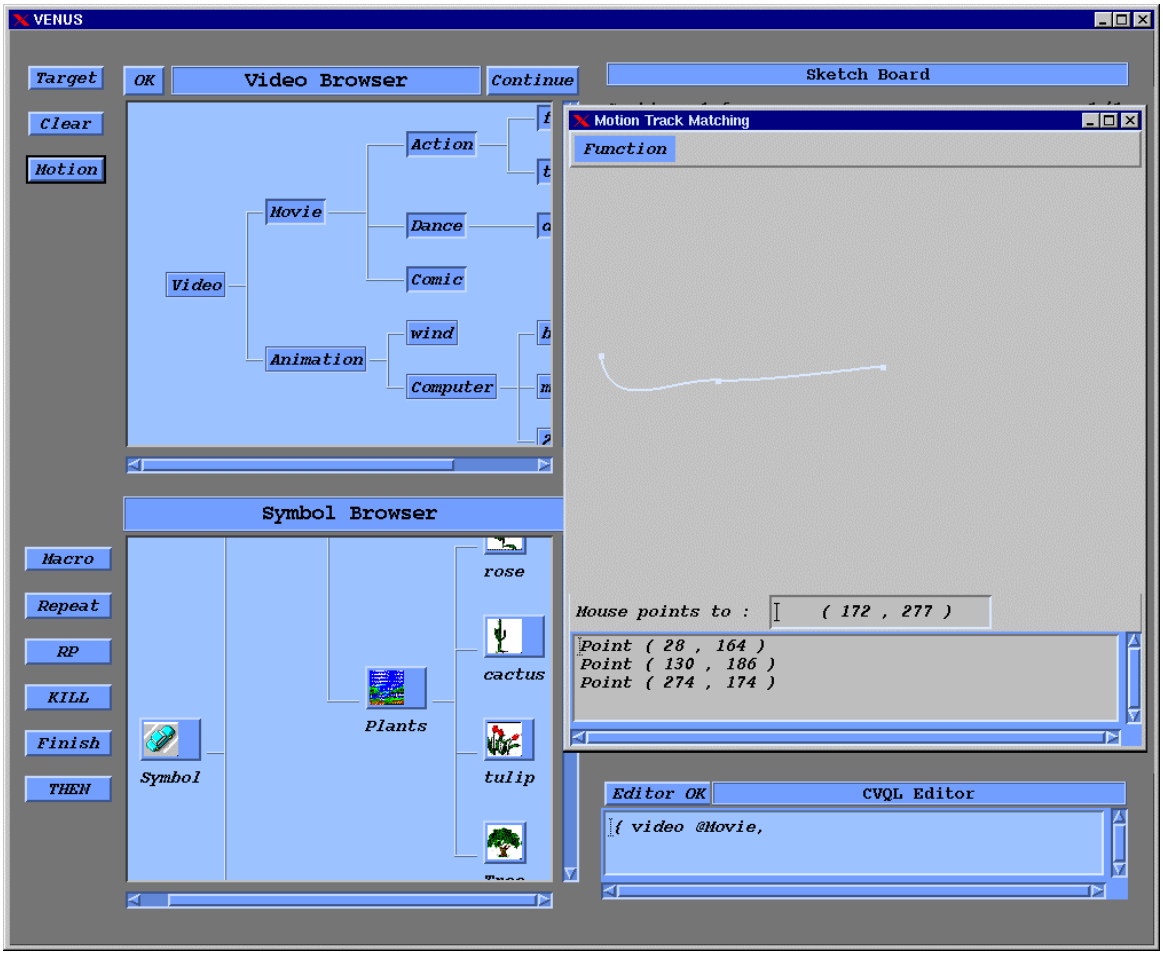

Figure 23 Venus video database system

After query processing, the video sequence like Figure 22 is played.

6. Conclusion and Future Work

Content-based retrieval is an important research issue in video databases. Motion track is a significant feature of video data. Previous work focus on the extraction of the motion track for automatic indexing. However, there are few efficient and powerful methods for the query processing based on such indexes. In this paper, two query processing methods are proposed.

There are two important issues on query processing based on the motion track index: (1) sub-matching and (2) approximate matching. In our index region approach, the motion track index is expanded to a region and a user specifies a query curve against the region. The similarity of the query curve can be quantified such that (2) can be achieved. To achieve (1), it requires an exhaustive check for all points. Our future work is to decrease the execution time of the sub-

matching.

Only one paper [18] about the motion track matching involves approximate matching, which achieves (2). However, the approach in [18] forces to map the query curve to the whole index motion track. The goal of (1) can not be achieved.

In our peak model approach, the motion track matching problem is converted into the string matching problem. An efficient sub-string matching method is proposed such that the goal of (1) is achieved. By the experiment results and complexity analysis, we conclude that the query processing time is only slightly affected by the variation of the track number in the database, the query length and the index length. It means that users will get the query results almost in constant time under the variation of the size of the database, the query length and the index length. The peak model approach is also extended to achieve the goal of (2).

Appendix The Peak Model Query Processing Algorithm

```
DEFINE TYPE TRANSITION =
 {   CODE  source ;        /*  source state of the transition */
     CODE  destination ;   /*  destination state of the transition */
     int   frame_no ;      /*  frame number of the transition */
  }

DEFINE TYPE TRANSITION_SET = SET OF TRANSITION

PROCEDURE  query_process ( F , query_string )
  INPUT  :  FINITE_AUTOMATA  F ;  /* The finite automata */
            STRING  query_string [ 1..QUERY_LENGTH ] ;  /* The query string */
  OUTPUT :  VIDEO_SEQUENCE_SET  V ;  /* Set of video sequences that match the query */
  {   TRANSITION_SET  candidate ;  /*  Candidate transitions after each step */
      TRANSITION_SET  candidate_after_des ;  /*  Candidate transitions after destination filter  */
      TRANSITION  t ;

      candidate = destination_filter ( F , query_string[1] , query_string[2] ) ;
      for ( i = 2 to QUERY_LENGTH -1 )
        {  candidate_after_des = destination_filter ( F , query_string[i] , query_string[i+1] ) ;
           candidate = continuity_filter ( F , candidate_after_des , candidate ) ;
        }
      for  ( ∀ t ∈ candidate )
        add to V a frame sequence of frame no. < t.frame_no-QUERY_LENGTH+2 ,…, t.frame_no+1 >;
      return V;
  }

PROCEDURE  destination_filter ( F , cur , next )
  INPUT  :  FINITE_AUTOMATA  F ;  /* The finite automata */
            CODE  cur ;  /* The current code */
```

```
                CODE    next ; /* The next code */
  OUTPUT :    TRANSITION_SET    T;    /* The candidate transitions after this filter */
{
    TRANSITION    t ;

    for    ( ∀ t ∈ F && t.source == cur )
        if    ( t.destination == next )
            Add t to T;
    return T ;
 }


PROCEDURE    continuity_filter ( F , candidate_after_des , candidate )
  INPUT    :    FINITE_AUTOMATA    F ;    /* The finite automata */
                TRANSITION_SET    candidate_after_des ;
                        /*    Candidate transitions after destination filter    */
                TRANSITION_SET    candidate ;      /*    Candidate transitions after the last step    */
  OUTPUT :    TRANSITION_SET    T;    /* The candidate transitions after this filter */
{    TRANSITION    s , t ;

    for    ( ∀ s ∈ candidate_after_des )
       for    ( ∀ t ∈ candidate )
          if    ( s.frame_no == ( t.frame_no + 1 ) )
              Add s to T;
    return T;
}
```

References

[1] R. Agrawal, K. Lin, H. S. Sawhney, K. Shim, “Fast Similarity Search in the Presence of Noise, Scaling, and Translation in Time-Series Databases,” Proceedings of the 21st VLDB Conference, 1995, 490-501.

[2] M. Agrawala, A. C. Beers, N. Chaddha, “Model-Based Motion Estimation for Synthetic Animations,” Proceedings of ACM Multimedia, 1995, 477-488.

[3] N. Dimitrova and F. Golshani, “Rx for Semantic Video Database Retrieval,” Proceedings of ACM Multimedia, 1994, 219-226.

[4] N. Dimitrova and F. Golshani, “Motion Recovery for Video Content Classification,” ACM Transactions on Information Systems, Vol. 13, No. 4, October 1995, Pages 408-439.

[5] T. C. T. Kuo, Y. B. Lin, A. L. P. Chen, S. C. Chen, and C. Y. Ni, “Efficient Shot Change Detection on Compressed Video Data,” Proceedings of International Workshop on Multimedia Database, 1996,101-107.

[6] S. Y. Lee and H. M. Kao, “Video Indexing - An Approach Based on Moving Object and

Track," Image and Video Processing Conference; Symposium on Electronic Imaging: Science and Technology, Vol. 1908. IS&T/SPIE, 25-36.

[7] Y. B. Lin, "An Efficient Method to Build Video Indexes from Compressed Data," Master Thesis, National Tsing Hua University, 1996.

[8] T. D. C. Little and A. Ghafoor, "Interval-Based Conceptual Models for Time-Dependent Multimedia Data," IEEE Transactions on Knowledge and Data Engineering, 5(4), 1993.

[9] C. C. Liu and A. L. P. Chen, "Modeling and Querying Processing of Distributed Multimedia Databases," Proceedings of Real-Time and Media Systems, 1996.

[10] R. J. Shalkoff, "Digital Image Processing and Computer Vision," John Wiley and Sons, New York, 1989.

[11] H. Shatkay, S. B. Zdonik, "Approximate Queries and Representations for Large Data Sequences," Proceedings of IEEE ICDE, 1996, 536-545.

[12] A. Yoshitaka, M. Yoshimitsu, M. Hirakawa and T. Ichikawa, "V-QBE: Video Database Retrieval by means of Example Motion of Objects," IEEE Proceedings of Multimedia 1996, 453-457.