Query-by-Example for Motion Capture Data

by

Bennett Lee Rogers

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2007

Author . . . .
Department of Electrical Engineering and Computer Science
t 31, 2007

Certified by. . . . .
Jovan Popović
Associate Professor
Thesis Supervisor

Accepted by . . . .
). Smith
Chairman, Department Committee on Graduate Students

Query-by-Example for Motion Capture Data

by

Bennett Lee Rogers

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Motion capture datasets are employed widely in animation research and industry, however there currently exists no efficient way to index and search this data for diversified use. Motion clips are generally searched by filename or keywords, neither of which incorporates knowledge of actions in the clip aside from those listed in the descriptions. We present a method for indexing and searching a large database of motion capture clips that allows for fast insertion and query-by-example. Over time, more motions can be added to the index, incrementally increasing its value. The result is a tool that reduces the amount of time spent gathering new data for motion applications, and increases the utility of existing motion clips.

Thesis Supervisor: Jovan Popović
Title: Associate Professor

Acknowledgments

I would like to thank my advisor Jovan Popović for his guidance and support throughout my time at MIT, for helping me find a project of interest, and for his patience while I chose my path. I would also like to thank the other members of the MIT Graphics Group, who allowed me to look forward to coming into the office each day, who gave me peers I could look up to, and more importantly, who became good friends. Thanks also to the American Society for Engineering Education who funded my research, which gave me the freedom to explore. Finally, I would like to thank my family for encouraging me and being proud of me. It is the foundation and ideals that they instilled in me that made all of this possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Motion capture data has become a critical tool in many areas of industry and academia, including movies, computer graphics research, video games, and medicine. Motion capture is a process by which the movements of a person may be recorded in 3-dimensional space for analysis and use in a variety of applications. The data generally consists of 3D locations of joints from the captured subject, tracked over time. Typically, the joints are joined together in a structure representing a human skeleton.

Many of the applications that use motion capture data require motion datasets tailored to that specific task, so that data recorded in the past is not useful in the present. New motions must be recorded for each new application. This means that even as the global collection of data grows, no project benefits from the overall accumulation. Our work presents a system that enables the retrieval of motions that are similar to a given query motion, meaning that as the global collection of motion capture data grows, our system will become increasingly capable of returning exactly the type of actions that are sought.

Suppose, for example, that you are developing an animation sequence that requires a character to perform a specific action. If you do not already have a motion clip that matches what you need, you will have to search existing motions, or capture a new sequence. If you choose to search for a relevant clip, it may be difficult to find what is needed. For instance, the Carnegie Mellon Motion Capture Database [3] indexes motions by using human generated filenames describing the actions that occur in each clip. This works well if the exact action you need is mentioned in the description, but it fails if you want a short sequence that may occur within a larger clip. In addition, it requires that you scan through the list of descriptions looking for particular relevant terms.

Another search method is to manually inspect longer motion clips by viewing the motion replays. Long motion files depicting complex actions may contain the desired sequence, but it is tedious to look through an entire database of motions for one particular action. The only remaining alternative is to capture a new motion sequence to generate exactly the action required, but this is a time consuming and expensive process. When complete, this new motion can be added to a repository of motion capture data, yet it is unlikely to ever be used again unless its filename happens to describe exactly what someone else looks for in the future.

In this project, we propose to help solve this data recycling problem. We would like to make it possible for the animator in the above scenario to find all instances of the required action in any clip in the database simply by providing an example of the action. This requires the system to maintain knowledge about the contents of the stored clips. The search also needs to be fast, which requires an efficient indexing scheme on the motions. Given such a system, many applications would become possible and the usefulness of this repository would grow with the motions contained within.

When designing a system for content-based motion capture retrieval, two main issues must be addressed. The first issue is how best to parameterize high dimensional motion data so that a compact, efficient representation is attained while retaining as much information as possible about the actual motion. Given an acceptable parameterization, the second issue addresses how best to index the motions so that they can be searched more efficiently than by sequentially comparing the query motion to each database motion frame-by-frame. Given the high dimensionality of motion capture data and the fact that individual motions can consist of hundreds of frames, this approach would be computationally infeasible.

We approach the parameterization issue by implementing a version of the geometric motion features proposed by Müller et al. in [12]. This parameterization captures the semantic meaning behind the poses and actions in the motion capture data, while enabling a clean indexing scheme. The indexing scheme we employ is derived from the idea of vocabulary trees as presented by Nistér et al. in [13]. Vocabulary trees are hierarchically clustered indexing structures that allow for fast insertion and querying, and they are compact enough to allow for many motions to be indexed efficiently.

By combining these two techniques, we are able to create a motion querying system that quickly returns motions relevant to the query motion. We base our analysis of system performance on visual inspection of the returned results, and on a scoring metric meant to measure both the number of relevant results returned and their position in the ranked retrieval list. Our results indicate that given a query motion, our indexing scheme is able to return motions that are labeled as 'relevant' about half of the time. That is, in a ranked list of 10 search results, 5 of those results on average would be labeled by an expert as representing the same motion, and in general, all of the returned motions are visually similar to the query action.

Overview

The rest of this document is organized as follows: In Chapter 2 we discuss previous motion retrieval systems and how they compare to our approach. Chapter 3 deals with the details of motion capture data and describes some possible parameterizations, including reasons for our choice of geometric features. The actual indexing system we implement is discussed in Chapter 4, which includes a discussion of the clustering required for building vocabulary trees. We present the results of our work in Chapter 5, and finish with conclusions and possible directions for future work in Chapter 6.

Chapter 2

Related Work in Motion Retrieval

Due to the increasing need for motion capture data in various industrial and academic settings, much work on motion retrieval has already been performed. For all of these techniques, one result is the ability to find motions in a database that are somehow logically similar to a given input motion. Notions of 'similarity' differ among the techniques, however, and the various methods use different means of parameterizing the motion data. Many of the approaches use a very shallow parameterization, and so their indexing and retrieval are performed on data that is very similar to the raw numeric spatial joint locations. This leaves those techniques overly sensitive to slight pose deformations that may dismiss otherwise relevant motions from retrieval results. Several of the current methods also employ dynamic time warping (DTW), a technique discussed in Section 3.2, to match similar pose sequences. Dynamic time warping is an expensive operation, as well as not being spatially invariant to skeletal orientation. Here we will describe some of these recent approaches to the problem of content based motion retrieval, and we briefly mention how they relate to our technique.

Cardle et al. [1] parameterize motions by recording the angular velocity of each joint for all poses in the motion. These angular velocities are then stored in multi-dimensional Minimum Bounding Rectangles and arranged into R-Trees, much like in the GEMINI framework described in [4]. Initial candidate matches are found using this structure and, with the aid of dynamic time warping, the optimal match in the database is found for a query motion. Additionally, the user is given the ability to specify which joints in the query are important, and only those joints are considered when the best match is calculated. Details of the indexing scheme used in their method are not described, however the use of dynamic time warping on the candidate matches indicates that retrievals will be slower as more possible matches are found. As the database size grows, it will be natural for more candidate segments to be discovered, and retrieval speed may suffer.

In [8], Keogh et al. allow for uniform scaling across the query motion to find better matches than are possible with DTW alone. This is because DTW attempts to find a one-to-one correspondence between the frames of two motions, even if there is no logical way to do so. Uniform scaling allows one motion to be stretched or compressed by a constant time factor in order to find more natural correspondences between motions. In addition, rather than requiring their technique to search through all possible scalings of a given motion in order to find a match, Keogh et al. introduce a method for calculating a lower bound on the distance between two motions. They accomplish this by precomputing a bounding envelope on the motions in their database and using these bounding envelopes to compute a lower bound on the distance between two motions over all uniform scaling factors. Once candidate motions are identified, all possible uniform scalings are searched for the best alignment between the motions, and the best match is returned. Again, the use of a time warping technique requires that their method scan all possible alignments of motions within certain frame windows, which can be a slow process. Also, the computation is performed on essentially raw motion data, which has the sensitivity issues mentioned above.

Kovar and Gleicher [9] recognize that motions that are similar logically, that is, motions that represent semantically related actions, may not be similar numerically based on a particular distance metric. Therefore they perform recursive queries that find all numerically similar segments to the input motion, and then perform the search again on all resulting motions. Their hope is that this will return related segments from the database that could not have been located using a single numerical search. In order to make the queries fast, they precompute a *match web* over the entire database. The match web essentially represents the links between all frames and motions in the database that would be considered numerically similar for a large enough threshold of similarity. Queries must then be segments of motions already present in the database and thus in the match web. Matching segments are returned and then those segments are used as the inputs for new queries, until no more unique matches are returned. One drawback to this approach is that adding a new motion to the database will require recomputing the entire match web to add the new frames. Therefore, increasing the size of the database will become very inefficient over time, whereas our technique allows motions to be added to the index essentially in real-time.

Rather than using the raw motion data, Forbes et al. [5] use principal component analysis (PCA) to reduce the dimensionality of the full space of possible motions. Because skeletons (and thus motions) are hierarchical, certain joints matter more in determining the characteristics of a motion. For example, a change in rotation of the shoulder joint will do much more to alter the appearance of a motion than a similar rotation of the wrist. Therefore, Forbes et al. weight the joints differently before the motions are projected into the PCA space. The weightings can be changed depending on the type of motions used, and the different weighted PCA spaces can be stored for future queries. Points in this PCA space are individual poses, and differences between poses are calculated using a simple Euclidean distance metric. To search the space with a given query motion, the 'characteristic pose' of the motion is calculated by finding the pose that is most dissimilar to the average pose of the motion. All poses in the database that are sufficiently close to this characteristic pose are found, and DTW is applied to the regions around these points. The result is a ranked list of pose sequences from the PCA space that are similar to the query motion. It is unclear how well the characteristic pose actually represents a given motion clip, but it seems likely that much information is left unaccounted for when the index is searched using one pose from the query motion. In addition, this approach requires that the user select the appropriate weights on the joints that will give the best retrieval results for the motion type being queried. This does not seem like an intuitive interface.

Liu et al. also use PCA in [10]. They begin by normalizing all poses in the database to the same root location and general orientation, and then they perform hierarchical PCA. At each step of the process, if all poses can be described by a linear model of user-specified dimension d, then a leaf node is formed. Otherwise, the set of poses in the node is split into two clusters, and PCA is performed on the resulting smaller pose sets. They then parameterize all motions in the database by storing a vector listing which leaf cluster each pose in a motion falls into. These vectors are compressed by replacing runs of identical cluster indexes with just one instance of that index. In order to find which database motions are similar to a query segment, the query is parameterized using the same process, and then the cluster transition sequences are compared using standard string distance techniques. Unfortunately, this technique requires the pose space to be clustered into exactly 2 groups at every stage of the hierarchy. This can lead to unnatural groupings, as opposed to our method where clusters are found based on the natural structure of the data. More importantly, once their index is generated from the hierarchical PCA, there is no way to add more information to it by adding new motions. Therefore the database must contain the desired final set of actions before the index is ever created.

In [12], Müller et al. characterize motions not by numerical values such as joint locations or angles, but by relational, geometric features. These features take binary values indicating whether specific geometric properties are true for a given pose. For example, one feature might be whether the right ankle joint is in front of the plane defined by the left ankle joint, the right hip joint, and the left hip joint. A set of 31 of these boolean features are used, and so each frame is described by a 31 bit vector. The use of relational features as opposed to absolute numerical features introduces spatial invariance to this technique. No matter the orientation or physical size of two skeletons, if they are in the same pose they will have the same feature vector. Temporal invariance is added by compressing runs of identical vectors similar to the way Liu et. al compress runs of identical cluster indexes in [10]. This temporal invariance means that two motions that are similar but have different timings, for example two walk cycles with different periods, will still have similar sequences of feature vectors. The database of motions is indexed by storing an inverted list relating the 2^{31} possible feature vectors to the frames in the database that match each configuration. Queries are performed by calculating intersections between the inverted list representing the query motion and the list representing all motions in the database. The result is the set of database motions that match the sequence of feature vector segments found in the query. Because of this intersection step, only motion segments with geometric signatures that exactly match the query motion will be returned as relevant. The concept of a 'fuzzy query' is utilized, but that merely means that only particular joint combinations are considered in the intersection. The retrieval process will still dismiss motions that are even slightly dissimilar to the query motion.

In [11], Müller et al. extend on their work from [12] while using the same class of geometric features. They introduce the concept of a motion template, which is a matrix where the rows represent different binary features and the columns are sequential frames. Given several instances of a particular motion type, a template is built by averaging the boolean value of each feature across the correlated frames of the different motions. The correlation between frames is found by performing DTW on the related motions before the averaging step. Given a template for a particular motion type, a large database of motion capture data can be queried for matches to this template. First, keyframes in the template are chosen, and only candidate segments in the database where these keyframes occur in the proper order are considered. Using DTW, the optimal distances between the motion template and the candidate segments are computed. All segments that are within a pre-specified distance to the template are returned as matches to the query. Unfortunately, to be able to perform a query using a motion template the template first has to be constructed, which requires several hand-selected instances of a particular motion type. This requirement severely limits the usefulness of this approach as a query-by-example technique.

Chapter 3

Data

The purpose of our work is to develop a means for efficiently and effectively indexing a large body of motion capture data, but before our approach can be discussed we must describe the structure and meaning of the data we wish to index. In this chapter we define the details of motion capture data and the various ways it can be parameterized. We also provide justification for our decision to use geometric features rather than numeric features to represent this data.

3.1 Motion Capture Data

Motion capture is a method for recording the movements of a live actor in 3D space. These recordings can then be used in many applications, for example to analyze physical injuries in sports medicine, or to animate characters in movies or video games. The motion data is typically acquired using a complicated camera setup, with reflective markers on the subject to be recorded. These markers are placed at core locations, joints, and extremities, so that the skeletal pose of the subject can be estimated given the locations of the markers. Cameras record the positions of the markers in 3D space over time, usually at 30 frames per second or more, and by connecting the locations of the markers by inferred 'bones', a 3D skeleton is obtained.

There are two general ways to represent the positions of the joints in a motion capture skeleton. The first is to specify the absolute 3D location of each joint relative

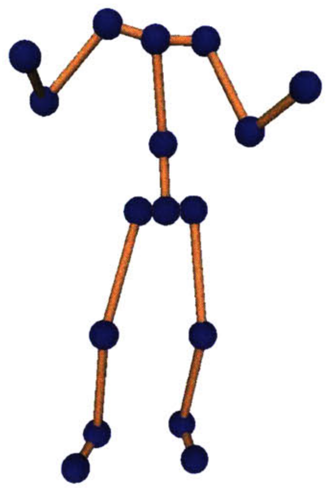
Figure 3-1: The motion capture skeleton used for the motions in our system. The skeleton consists of 18 joints connected in a hierarchy by inferred bones. Motion data specifies the 3D locations of each joint in each frame.

to some fixed point in space. The second method utilizes the fact that a human skeleton is essentially a tree structure with the root in the core of the body, and the branches fanning out into the limbs. Each joint except for the root has a parent joint above it in the hierarchy, and the child joint's location can be described by an offset and rotation relative to the location of the parent joint. The location of the root joint is specified relative to a fixed point in space. In this thesis, all motions are based on a skeleton with the topology shown in Figure 3-1.

The joint hierarchy and relative offsets are fixed and describe the overall topology of a motion capture skeleton. The actual motion can be specified in a number of ways, but in this thesis we use two motion formats, the Biovision Hierarchical (BVH) format and a custom format. In both file types, the motion is described by specifying the pose of the skeleton in each frame over time. In a BVH file, a pose consists of the absolute location of the root joint, followed by the rotation of each joint relative to its parent. The rotation is represented by 3 values corresponding to the rotation of the joint around the Z-, X-, and Y-axes. The custom format specifies the skeletal hierarchy and then the absolute 3D locations of each joint in each pose. By replaying the sequence of poses in each frame in order, the original motion is generated.

3.2 Motion Features

When searching a database of motion capture files for a particular action or pose sequence, it is wise to parameterize the motion in a manner besides using the raw translation/rotation data. A motion capture skeleton may consist of anywhere from a dozen to a few hundred individual joints, each of which can have up to three degrees of freedom. This can lead to a tremendously high dimensional source of data, which would be nearly impossible to index and search by conventional means. By detailing the locations of the joints on a per-frame basis, the raw data also specifies the precise scale and orientation of the skeleton, as well as the speed of the motion. In general, these factors are unimportant when trying to recognize particular semantic actions. For example, when trying to recognize a punching motion in the database, it does not matter in which direction the skeleton is facing, nor does it matter how tall the skeleton is. In most cases, the absolute speed of the punch does not matter either. What matters is the particular sequence of joint rotations that generate what can logically be described as a punch. In this instance, the joint rotations can be viewed as a *feature* extracted from the full set of original data. These features can be either numeric or geometric, and they abstract away some of the lower level details about a motion and allow the general characteristics of the motion to be represented. This makes it easier to compare two frame sequences to determine if they are related. This ability to compare is critical when attempting to retrieve logically similar motions from a database. The abstraction can also significantly lower the dimensionality of the data, meaning that operations that would be infeasible to execute on the raw motions become practical when applied to the extracted features.

In this work, we choose to use geometric features rather than numeric features, because they provide a certain level of tolerance when comparing semantically related poses and actions. Specifically, we use the set of 39 binary geometric features presented by Müller and Röder in [11]. The full description of this feature set can be found in Appendix A.1. In general, each of these features takes as input the 3D locations of the joints of a motion capture skeleton in a particular pose and generates as output a boolean value indicating whether the quality described by the feature is true or false. By concatenating all 39 binary values together, we generate a 39 dimensional feature vector for each pose. These feature vectors are used to index the motion capture database as described in Chapter 4. In the rest of this section, we will discuss the differences between numeric and geometric features, and the qualities of each that lead us to choose geometric features.

Numeric Features

Numeric features quantitatively describe the state of the joints in the skeleton as they move over time. Examples of numeric features include raw rotation values as mentioned above, angular velocities of joints as in [1], the local spherical coordinate of each joint relative to the root as in [2], or more complex features such as derivatives of the joint trajectories over time. Numeric features are precise in defining the skeletal pose, but one disadvantage with them in motion retrieval applications is that they contain no notion of the semantics of the motions involved. Two motions may be similar logically, but quantitatively they may be very different. For example, a punch with the right hand will be numerically very different from a punch with the left hand, and a walking motion with the hands down by the hips will be different from a walk with the arms extended. In addition, numeric features depend on the speed of the actions performed. When comparing two walking motions frame-by-frame, if one walk has a faster cycle than another, the motions will be numerically different. Dynamic time warping is a way to counter this time dependence, but it is computationally very expensive.

Dynamic time warping

Dynamic time warping is a technique for computing correspondences in time-series data. In situations such as the walking cycle problem just mentioned, the goal of dynamic time warping is to find a one-to-one mapping between the frames of the two motions such that the difference between corresponding frames is minimized. In order to accomplish this, frames in either motion may be 'stretched out' so that they line up with more than one frame of the other motion, but the time progression through both motions must be monotonically nondecreasing. The result is that motions that differ only in timing of execution can be recognized as being similar, but the polynomial time complexity of dynamic time warping makes it undesirable in situations where speed is important.

Geometric Features

Geometric features qualitatively describe the configuration of the motion capture skeleton over time. At each frame, a geometric feature indicates relations in 3D space between particular skeletal joints. These features are used extensively in [12], and it is from there that we draw the basis for the features that we use.

As an example, Figure 3-2 displays one such geometric feature. In order to determine the status of this feature, we calculate the anchor position and normal vector of a plane that is centered at the root of the skeleton and that spans the root, left hip, and left foot joints. At each pose of a motion, a boolean value is computed that represents whether the right ankle joint is in front of or behind this plane. As opposed to numeric features, a geometric feature such as this has a semantic meaning, in this case whether the right foot is in front of the left foot. Figure 3-3 shows a different type of geometric feature. Here we calculate the angle at which the right elbow is bent. To do so, we simply find the vectors that represent the humerus and the forearm of the skeleton, and compute the angle between them. By specifying a threshold value, in our case $110°$, we arrive at a feature that logically represents the cases where the right arm is extended or bent. Other possible features may test whether absolute speeds of joints through space are above particular thresholds or whether the proximity between two joints, such as the hands, is below a certain value. By using a set of such features defined for various joint relationships across the whole skeleton, we can approximately describe the pose of the skeleton at every frame of motion. The set of features we use to parameterize our motions is described in Appendix A.

Our motion feature vectors consist of a set of these boolean values, one value for each feature and one feature for each frame. While numeric features may be very

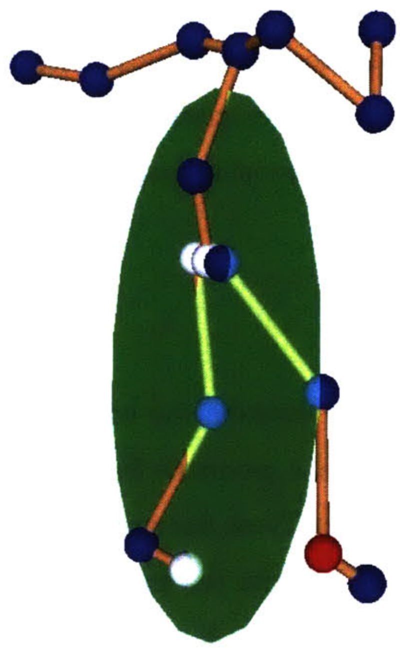
Figure 3-2: This geometric feature tests whether the right ankle, marked as a red joint, is in front of the green plane that spans the root, left hip, and left foot joints, marked in white.

different even for fairly similar poses, due to the fact that they take boolean values geometric features are much more tolerant of variations between logically related poses, and so are useful when searching for related motions in applications such as ours.

Adaptive segmentation

The time dependence problem seen with numeric features is still present when using geometric features, however, there are solutions to this problem other than dynamic time warping. Since the geometric features are less exact than numerical features, poses that may be slightly different numerically will correspond to the same geometric feature vector. This results in 'runs' of identical feature vectors. By compressing these runs into single instances of each feature, we are left with a sequence of features consisting only of the distinct pose states through which the skeleton passes. This process, used in [12], is known as *adaptive segmentation*. Geometric features

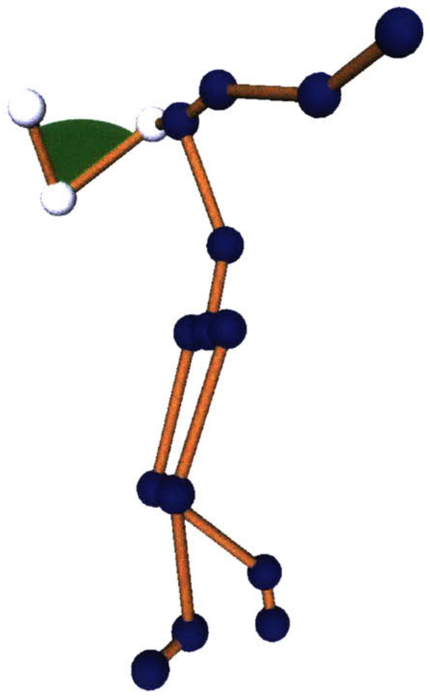

Figure 3-3: A different geometric feature that examines the angle of the right elbow joint. If the angle, marked in green, between the humerus and forearm is less than 120°, the feature returns a true boolean value.

compressed by adaptive segmentation can be viewed as a pattern of sorts for the given motion, and regardless of timing, similar motions will have the same pattern. The idea is that by comparing these compressed motion patterns, simple differences in timing between two motions may be disregarded, and retrieval performance will improve. We implement this idea in our solution.

Chapter 4

Indexing

The task of indexing is essentially that of building a map from a key or feature to a subset of objects that contain that key from within a larger set. The simplest example is that of an index in the back of a book, which is a mapping from particular words or concepts to the chapters or pages in the book that deal with those topics. These indexes are useful tools for the problem of content-based retrieval of information because they can be used to speed up the task of finding similar objects. In motion retrieval the benefits are large because of the high cost of comparing motions sequentially. Rather than comparing the query motion with each database motion frame-by-frame, a vector can be generated representing the index of each motion, and these index vectors can be compared efficiently.

The first step in generating an index for motion capture data is to decide on the feature to use to represent the data. As discussed in Chapter 3, we have decided to use geometric features based on the work of Müller and Röder in [11]. Once these features are extracted from the motions, they must be stored in a global index structure that represents all motions in the database. The indexing scheme we have designed is largely based on the work presented by Nistér and Stewénius in [13], where they use vocabulary trees to index a database of images.

Once the index is built, it can be used for retrieval. To search the index for motions similar to a query motion, first motion features must be extracted from the query motion. These features form a set of *terms* that represent the query motion.

Motions that contribute similar sets of terms to the index are similar in content, so database motions that exhibit similar terms to the query motion are candidates for retrieval. All that remains is to decide how to weight the different terms in the index, and how to score one set of terms against another. The specific methods we employ to build and use an index with vocabulary trees will be discussed in the this chapter.

4.1 Vocabulary trees

A vocabulary tree as defined in [13] is a structure that organizes feature vectors into hierarchical clusters. For a visual example of a vocabulary tree, see Figure 4-1. The tree structure allows the index to be built and searched much more efficiently than a flat structure, and so allows for a larger vocabulary of features to be indexed. With a wider variety of motion features stored in the index, more accurate retrievals should be possible. Another benefit of vocabulary trees is that once the tree has been built, inserting new motions is very fast, and every new motion increases the retrieval power of the system. Vocabulary trees are also very compact. As we will discuss, all that needs to be stored is one feature vector for each node of the tree, and the number of features from each motion that are associated with each node.

Building the tree

The first step in building a vocabulary tree for motion is to select a representative set of motions. This step is very important, as the structure of the tree depends on the space of motions it is initially built with, and this structure will not change once it has been formed. Therefore the representative set should accurately reflect the space of motions that may be inserted into and queried from the index.

Given an initial set of motions, feature vectors are extracted for each frame that is to be inserted into the tree. As discussed in Chapter 3 we use 39 binary geometric features, so our features are 39 dimensional binary vectors. Given that most motion capture systems use frame rates of at least 30fps, there is a lot of redundant data, at least as pertains to obtaining a representative set of frames. Therefore, we sample

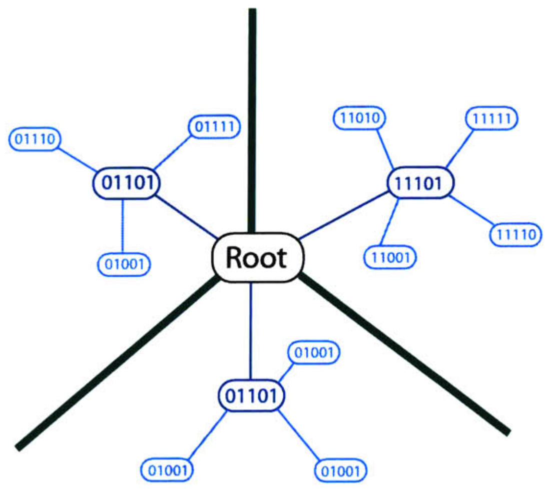

Figure 4-1: A simple representation of a vocabulary tree. Three top-level exemplars, represented as 5 dimensional binary vectors, branch off of the root. Each of these in turn branches into several child nodes. This tree only contains 2 levels of nodes, but in practice the hierarchy can be arbitrarily deep.

long input motions at a lower rate, only using 1 out of every 10 frames to build the initial tree. This allows us to insert many more motions into the tree to gain a representative set, while keeping the index generation time reasonable. Many motions we may wish to insert into the index are no more than a few dozen frames long, so in order to retain the value of variety these shorter motions add to the tree, we employ an adaptive sampling scheme. We sample shorter motions more densely, and we sample long motions sparsely, down to a threshold of 1 out of 10 frames. Again, the variety of poses that the tree is built with is what is important, so this uneven sampling does not distort the retrieval quality of the index, it simply allows a broader space of poses to be represented.

The next step is to cluster the feature vectors to form the root nodes of the tree. There are many clustering techniques available, but we choose to employ an algorithm called affinity propagation, developed by Frey et al. and described in [6]. In Section 4.2 we discuss the various clustering methods and our reasons for choosing affinity propagation. We would like the tree to be not too flat and not too narrow, so we tune the parameters of the affinity propagation to give us a number k of clusters where $5 \leq k \leq 20$. We allow this range so that the affinity propagation algorithm

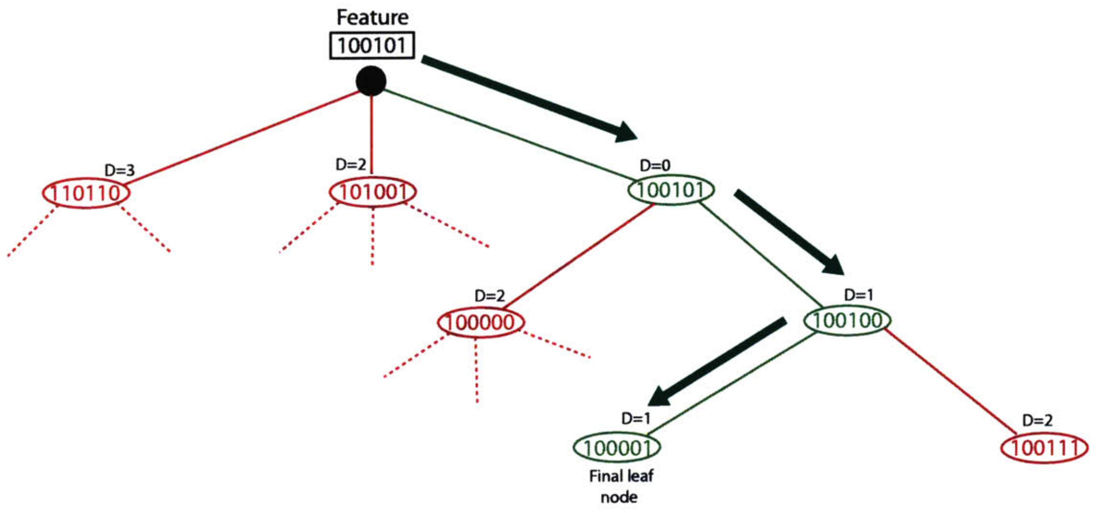

Figure 4-2: Here we show an example of how a feature vector might propagate down a few levels of the tree. At each level, the vector is compared to the exemplar for every cluster at that level. The node with the best match is found, the vector is pushed down to the children of that node, and the process repeats until the vector reaches a leaf of the tree.

can choose the number of clusters that is appropriate for the structure of the data. This concept will be explained in Section 4.2 as well.

The result of this process is a set of clusters where each cluster is centered around the feature vector that best describes the characteristics of that cluster. This feature vector is known as the *exemplar* for that cluster. The feature vectors in a given cluster are those that are most similar to the corresponding exemplar, as defined by the distance metric used for clustering. A discussion of distance metrics is presented in section 4.2.3.

Once the k root cluster nodes are generated, each node is re-clustered by the same process. Clustering continues recursively down each branch of the tree until the affinity propagation algorithm only finds one cluster in a node on that branch, or until there are less than 20 feature vectors in a cluster. Once the recursion finishes on all branches, the tree structure is complete. As long as we wish to use the same tree structure, we never need to regenerate it. Storing the tree requires only recording its topology and the feature vector representing the exemplar at each node.

When the tree structure is complete, it can be populated with as many motions as desired. In order to insert a motion into the tree, first we extract a feature vector for each frame of the motion. We do not sample the motions in this step, since insertion is very fast and each frame contains valuable information to help match with a query motion. Features are inserted by first comparing the feature vector to all root exemplars to find the one that is most similar to the current feature. This process is then repeated recursively on the children of that node, continuing until the feature has propagated down to one of the leaves of the tree. See Figure 4-2 for a visual example. At the leaf that finally receives the current feature, an *inverted file* is maintained that keeps track of the motions that contribute features to that node, and how many features come from each motion. An inverted file is itself an index structure much like the index in the back of a book. In our case, a complete inverted file would contain an entry for each of the 2^{39} possible feature vectors, with each entry listing how many times that feature appears in each motion in the database. Of course, we do not store every entry at every leaf node, since the vast majority of the entries would be empty. We only store entries in the inverted file of a leaf node for feature vectors that actually get propagated to that leaf.

Querying the tree

Once populated, the vocabulary tree is ready for querying to find all motions in the database similar to an example motion. The intuition behind this process is to first calculate the paths down the vocabulary tree for every feature in the query motion, similar to the process shown in Figure 4-2. Then we find all database motions with feature vectors that generate a similar set of paths down the tree. The process for doing this starts by creating an empty tree with the same structure and exemplars as the original tree. Feature vectors are extracted for all frames of the query motion, and the new tree is populated with these features. All that then remains is to compare the inverted files across all nodes of the tree with those of the query tree, for all motions in the database. The inverted file for a node in the middle of the tree is generated by simply concatenating the inverted files of its children. A similarity score is generated between the query motion and each database motion, and the motions are ranked and returned based on their scores.

The score for a particular database motion is based on the similarity of a vector d, representing its occupation of the vocabulary tree, to a vector q, representing the occupation of the query motion. Both q and d contain an entry for each node in the vocabulary tree, and entries are weighted differently depending on a weighting value assigned to each node of the tree. The entries in these vectors for node i are defined as

$$q_i = n_i w_i \tag{4.1}$$

$$d_i = m_i w_i \tag{4.2}$$

where n_i and m_i are the number of features from the query and database motions that pass through this node. w_i is the weighting assigned to node i, and is defined as

$$w_i = \ln \frac{N}{N_i} \tag{4.3}$$

where N is the number of motions in the database, and N_i is the number of motions with at least one feature contained in node i. This weighting is based on the *inverse document frequency* system commonly used in text retrieval. It gives less weight to nodes that contain features from many different motions and therefore do not contribute much discriminatory power to the scoring.

Having defined the query and database scoring vectors q and d, we calculate the score for a particular motion in the database as the normalized difference between the two scoring vectors:

$$s(q,d) = \left\| \frac{q}{\|q\|} - \frac{d}{\|d\|} \right\| . \tag{4.4}$$

Any norm can be used, but based on the arguments presented in [13] we choose to use the standard L_1-norm. The scores must be calculated for every motion in the database for a given query, but much of the work can be accomplished in a pre-computation step. The d vector can be calculated and stored for every motion in the database, and only needs to be updated when new motions are added. Then when a query is performed, all that needs to be computed is the q vector and the distances to each d vector.

4.2 Clustering

In order to implement a vocabulary tree index, we must have a way of clustering the feature vectors extracted from the motion capture data. The features we use are 39 dimensional binary vectors, and so whichever method we choose must perform well on data of this type. After researching the available clustering techniques, we decided to utilize a fairly recent algorithm known as affinity propagation. Affinity propagation has the desirable characteristics of being deterministic, fast to execute, more accurate than competing methods, and it allows for user-defined distance metrics on the data points. The next section describes the details of affinity propagation, as well as discussing some other clustering options, and we then discuss the distance metric used.

4.2.1 Affinity Propagation

Affinity propagation[6] is a clustering method that relies on a message passing process. It takes as input a set of similarities or distances S between pairs of data points, and a set of preferences P, representing the degree to which each data point would make a good cluster center, or exemplar. In our case, every point is equally likely to be an exemplar, so $P(x)$ is the same for every data point x. As the algorithm iterates, the messages that are passed between data points converge on a decision about which points should be exemplars and which points are assigned to each exemplar.

There are two types of messages passed. *Responsibility* messages passed from point i to point k indicate how much point i thinks point k should be its exemplar, given all of the other points that i could choose as its exemplar. *Availability* messages passed from point k to point i indicate how much point k thinks it is a good exemplar for point i, given all of the other points that want k to be their exemplar. As the algorithm iterates and these messages are passed among all data points, eventually a consensus is reached and certain data points are chosen as the exemplars for the points most similar to them.

The number of clusters discovered using affinity propagation is not fixed before the algorithm is run. Instead, the number of clusters is a result of the preference $P(x)$ chosen for each data point. If $P(x)$ is chosen to be large for every point, the algorithm will result in a large number of clusters, and vice versa. But unlike in k-means clustering, the number of resulting clusters fits the natural layout of the data. With low preferences on each data point, only larger trends will be uncovered in the data, resulting in larger clusters. As the preferences are increased, smaller clusters within the overall layout will begin to emerge. The preferences can be tuned to generate the level of detail desired in the cluster assignments. In some applications it would be helpful to choose beforehand how many clusters are desired, but letting the algorithm decide does result in a more natural fit with the data, and by tuning the preferences the cluster count can be targeted to a general range. We tune the preferences automatically as the clustering is run in order to generate in the range of 5 to 20 clusters.

Affinity propagation is shown in [6] to result in less error in cluster assignments than k-means clustering. This means that the sum of the dissimilarities between each point and its exemplar is smaller than for k-means. The algorithm is more expensive than one run of k-means, but in order to achieve a trustworthy cluster assignment from k-means it must be run multiple times. In one experiment in [6], in order to achieve comparable error results to affinity propagation, k-means clustering had to be run 10,000 times taking two orders of magnitude more time to complete. Even then, affinity propagation resulted in more accurate cluster assignments.

Parameters

As mentioned above, two of the input parameters to affinity propagation are the similarities between data points, and the preferences each point has for being an exemplar. The similarities are user-defined, and reflect how 'close' each point is to other points in the space in which the data lies. To generate similarities for our 39 dimensional binary data, we use the Hamming distance between points. Details of this and other possible distance metrics are discussed in Section 4.2.3.

The other parameter, the exemplar preferences, ultimately determines how many clusters will be discovered in the data. Since we have no prior knowledge of which data points will make good exemplars, all preferences are set to the same value, typically the median of the similarity values as suggested in [6]. Since we would like our vocabulary tree to have $5 \leq k \leq 20$ clusters at each level, and since there is no way to specify this using affinity propagation, we implement an automatic tuning step for the clustering. This step simply examines the number of clusters k generated by affinity propagation, and if k is not in the desired range, we increase or decrease the preferences accordingly and rerun the algorithm. We have found that by using this method we are able to generate the desired number of clusters at each node level by automatically tweaking the affinity propagation parameters just a couple of times in each case.

4.2.2 Other Clustering Options

K-Means Clustering

K-means clustering is a divisive clustering technique. Given a set of data points and a metric for measuring distances between pairs of points, k-means clustering divides the data into k clusters, with the intention that all points within one cluster are somehow similar, and pairs of points in different clusters are somehow dissimilar. Each cluster has a centroid at the center of the cluster, and every point is assigned to one centroid. The goal of k-means clustering is to minimize

$$V = \sum_{i=1}^{k} \sum_{x_j \in S_i} |x_j - \mu_i|^2$$

where k is the number of clusters, S_i are the clusters themselves, x_j are the data points, and μ_i is the centroid of cluster i.

The basic process for k-means clustering begins by assigning every data point to one of k clusters at random. The centroids of the clusters are then placed at the positions of the means of the data points assigned to each centroid. Every data point is then reassigned to the centroid that is the closest to it. The algorithm repeats the steps of adjusting the centroids then reassigning the data points until the process converges. This occurs when no data points are reassigned to a different cluster in an iteration.

K-means clustering is a simple clustering algorithm that is fast to converge, but one problem is that it does not necessarily find a globally optimal solution. In fact, the algorithm can reach different solutions depending on how the cluster centroids are initially assigned. For this reason, it is standard practice to perform multiple runs of the k-means process on a single dataset, and to accept the cluster assignment with the most optimal solution. The necessity to perform repeated iterations impacts the efficiency of the algorithm.

Another quality of k-means clustering that is undesirable in some applications is that the number k of clusters must be specified by the user. In many cases the user will not know how the data should be divided, and forcing the data into an arbitrary number of clusters ignores the natural layout of the points.

Agglomerative Clustering

Agglomerative clustering works in the reverse direction from k-means clustering. Initially, it treats every data point as its own cluster. The distance between every cluster is calculated, and the two points with the shortest distance between them are joined into a cluster. This process repeats until all of the data points are joined into one large cluster.

The method used for calculating the distance between clusters and then linking them varies. The three main ways of linking in agglomerative clustering are single linkage, complete linkage, and average linkage [14]. In single linkage the distance between two clusters is defined by the distance between the two closest points in those clusters. This method can lead to chains and long, unnatural groupings. Complete

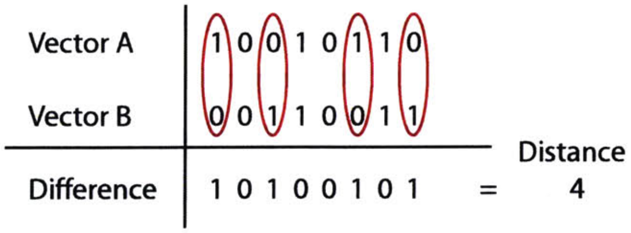

Figure 4-3: Here we show a sample distance calculation between two binary vectors using the Hamming distance. Vectors A and B are 8 dimensional binary vectors. Differences between the two vectors are circled in red. We can generate a vector representing the difference between these two vectors by computing the XOR function between them. Then the computed distance between the vectors is equal to the number of 1s in the difference vector.

linkage defines the distance between two clusters as the largest distance between two points in the clusters. Noise in the data can adversely affect the performance of this linkage scheme, as outliers can incorrectly prevent two clusters from joining. Finally, average linkage calculates the center of each cluster, and the distance between two clusters is defined by the distance between their centers. The disadvantage here is simply that recalculating the average of each cluster at every stage of the hierarchy can be prohibitively expensive.

4.2.3 Distance metric

In order to determine the similarity of two motions, we need to be able to compute the difference between individual poses. For our approach, this becomes an issue when building the similarity inputs for the affinity propagation, as well as when we populate the vocabulary tree by comparing feature vectors to each node exemplar. The appropriate distance metric to use depends on the form of the data, which in our case consists of binary vectors. The distance computation is simplified by the fact that our feature vectors will always have the same dimension, which depends on the number of geometric features used.

The distance metric we choose to use is the Hamming distance. The Hamming distance applies to two vectors of the same dimension, and is defined as the number of element substitutions required to convert from one vector to the other. It is essentially a simplification of the string edit distance calculation used in many text retrieval techniques. A simple example of how we compute the distance between two binary vectors is shown in Figure 4-3. The Hamming distance is very efficient to compute, and for binary data vectors it is the most intuitive approach. We choose to compute the distance function on our binary vectors as opposed to trying to calculate pose similarities using the raw motion data, as this raw data is very complex and does not represent the underlying motion in a general, logical way. Distance measures, such as Euclidean distance, between the raw data of motion poses would be very expensive to compute when considering the large quantities of motion data we would like to support. By instead working with the Hamming distance between binary vectors, we allow the distance calculation to be more flexible in capturing logically related motions, while gaining the added benefit of being very fast to compute.

Chapter 5

Results

In this chapter we present observations about the performance of our index when tested on a set of hand-annotated motions. The result of a query on our system is a list of motions found to be similar to the input action. In general, we find that we are able to recover about half of the relevant motions from the database for any given query, where relevant motions are defined to be those that are annotated with the same label as the query. For example, if we search for clapping motions in a database that contains 20 instances of clapping motions, on average the resulting list will contain 10 of those clips in the first 20 spots. In addition, for databases containing between 10,000 and 50,000 frames of motion, queries on our test indexes only take between 5 and 12 seconds to find the optimal match. This indicates that our technique does provide an efficient means of searching the motion data.

Inserting new motions into the index is very fast as well. Theoretically, since inserting a new feature vector means propagating the feature from the root to the leaves of the tree structure, inserting new frames should be roughly logarithmic in the number of tree nodes, depending on the (possibly irregular) distribution of nodes throughout the tree. In our tests we find that we can insert frames at around 60 frames per second when using a tree generated with 20,000 frames. The index is also space efficient, as we found that adding new motions to a pre-generated index only adds about 8KB for every 1000 new frames. This growth rate was calculated using the same index initially constructed from 20,000 frames, and constructing the index with a larger representative set would increase the growth rate somewhat, but any growth rate in this range will be easily handled by modern storage systems.

Our tests also indicate that adding new motions to an index does not seem to increase retrieval performance, but the performance ratio does stay roughly constant, and with more motions in the index a constant performance ratio means that a larger number of relevant motions are being retrieved. Possible explanations for these results are discussed below after we present the results themselves.

5.1 Retrieval performance evaluation

In order to quantitatively evaluate the performance of our retrieval system, we need an objective way to determine if returned motions are similar to a particular query motion. Unfortunately, this notion of similarity is exactly what we are attempting to capture in our scoring metric, and it would be circular to use the same metric for both retrieval and evaluation. Therefore, we choose to base our performance assessment on a motion capture dataset that has been labeled by hand. The dataset consists of Lindy Hop dance sequences generated for use by Hsu et al. in[7]. The dance sequences have been cut into 8 distinct 'steps', and each file is labeled according to the dance step with which it corresponds. Our evaluation involves populating our vocabulary tree with motions from this dataset, then calculating how often the motions returned from a query have labels that match that of the query motion. This technique injects some expert knowledge into the evaluation, as the dance steps are cut and labeled by someone who recognizes the motions. We perform multiple tests on this dataset in order to determine the performance of the system under several different conditions. Here we will describe the general test layout, as well as the scoring metric used to evaluate the results. Then the specifics for each test and the numerical results for each will be presented.

5.1.1 Evaluation structure

The dance dataset we use for these evaluations consists of 288 motion clips, each depicting one of 8 dance steps. All of the motions are of roughly similar length, and they all use the same motion capture skeleton and frame rate. For these tests, we build a vocabulary tree with a subset of these motions, then populate the tree with some or all of the 288 motions. We query the resulting tree with each of the 288 motions in turn, generating a ranked list of the top matches for each query. These ranked lists are scored as described in Section 5.1.2, and we can use these scores to understand both the performance of our indexing scheme in general, and to determine how to build the vocabulary tree to generate optimal retrieval.

5.1.2 Evaluation rank metric

Given our annotated dataset and a ranked list of returned matches from a particular motion query, we can evaluate how well the retrieval performed versus its possible optimal performance. For example, if we are querying using a motion tagged with label 1, and there are 90 motions of type 1 in the vocabulary tree, then a perfect retrieval would return all type 1 motions in the first 90 positions of the ranked list. Obviously this is unlikely, as no motion similarity metric is going to perfectly divide up the space of all motions, but it provides us with a standard against which we can judge retrieval performance.

To score an imperfect retrieval ranked list, two factors are considered: we would like as many of the matching motions to appear in the list as possible, and we would like those motions to appear near the top of the list. Therefore we adopt a weighted percentage scoring scheme that increases with the number of correct entries and assigns more value to entries near the top of the list. An example of a ranked list scored with this metric is shown in Figure 5-1.

Scoring begins by generating a sum depending on the locations of relevant results in the retrieval list. A correct retrieval at the bottom of the list will add 1 to the score, a correct retrieval one position higher will add 2, and so on up to the top of the

Query Type 1

Rank	1	2	3	4	5	6	7		
Retrieved Type	1	1	2	1	4	6	1		Sum
Score	7	6	0	4	0	2	0	=	19

Maximum Possible Score = 28

Final Retrieval Score = 19/28 = **67.9%**

Figure 5-1: A sample ranked list from a motion query, and its associated calculated score. Here we assume that we are querying the index with a motion of type 1, and that there are 7 instances of type 1 motions in the index. For each entry in position i of the returned list, we check to see if that entry is of type 1. If so, we increase the score by $7 - (i - 1)$, meaning that the maximum possible sum is $\frac{(7^2+7)}{2} = 28$. For the entries that are not of type 1, shown in red, we do not increase the sum. We then divide by the maximum possible sum to find our retrieval score, here 67.9%.

list, where a correct entry will add a value to the score equal to the total number of instances of the current motion type in the tree. Additionally, motions higher on the retrieval list should be more similar to the query motion, and so to encourage results that place more similar motions higher in the list those entries contribute more weight to the final score. To see why we would want to do this, consider the situation where retrieval lists are returned for two different queries of motion type 1, when there are 7 motions of type 1 in the index. The first list contains 1 motion of the correct type in position 1, and the second list contains 1 motion of the correct type in position 7. Intuitively, we would like to reward the first list more for placing the correct result higher in the ranked list, but we would still like to reward the second list somewhat for returning a relevant result in the top 7 results. Our inverse weighting scheme allows this.

Once the sum is generated, the score is then divided by the total number of possible points for the current motion type. This normalization allows for retrievals of dance types with differing instance counts to be compared.

5.1.3 Test cases

In this section we describe the various tests we use to analyze the performance of our indexing scheme. As mentioned above, for each of these tests we build a vocabulary tree using different motion capture files and different parameter settings. We then populate the tree with either a subset of the motions, or with all 288 dance clips, as described in Section 4.1, and we query each tree with every motion to get an average score for each dance step type.

Adaptive segmentation

Initially, we would like to determine whether the adaptive segmentation technique described in Section 3.2 benefits retrieval performance in terms of speed and accuracy. The first test consists of building and populating one tree with the first 50 motions from the dance dataset, using all of the feature vectors for every frame of the data. Another tree is built and populated with the same 50 motions, but adaptive segmentation is performed to compress runs of identical feature vectors.

Table 5.1 shows the results regarding the timing and retrieval accuracy between the two cases. Both the per-label average and the overall average for each case are shown. The first thing we notice is that using adaptive segmentation does slightly speed up the average query, but since a query takes only a few seconds the difference is not especially important. However, retrieval accuracy does appear to be diminished somewhat by using the adaptive segmentation. For most of the motion types, the accuracy increases or decreases only slightly when using the segmentation, but for a few of the labels the performance suffers a large decrease. This leads to a performance around 3% worse on average over all labels. This indicates that removing the runs of identical feature vectors removes information that is useful in characterizing the dance motions. While it appears that the majority of this information remains intact based on the fact that the retrieval performance decreases only slightly, it seems that removing this information may lead to errors classifying motions that are already easily confusable.

	Label 1	Label 2	Label 3	Label 4	Label 5	Label 6	Label 7	Label 8	Average	Time
Unsegmented	67.7%	56.0%	23.1%	70.4%	51.9%	57.4%	34.3%	79.6%	55.0%	6.3s
Segmented	59.4%	53.0%	25.9%	67.6%	48.6%	42.6%	36.1%	82.4%	51.9%	4.6s

Table 5.1: Results comparing retrieval performance with and without adaptive segmentation. Each tree was built and populated with only 50 motions from the database, one with adaptive segmentation and one without. On average, using adaptive segmentation seems to degrade performance slightly, but it does speed up the retrieval time.

	Label 1	Label 2	Label 3	Label 4	Label 5	Label 6	Label 7	Label 8	Average
50 Motions	64.9%	49.3%	27.1%	55.7%	40.1%	35.7%	44.5%	77.3%	49.3%
100 Motions	61.6%	51.6%	24.0%	60.0%	43.5%	39.4%	42.1%	79.0%	50.1%
150 Motions	62.7%	48.0%	25.1%	57.9%	38.0%	39.2%	41.6%	64.4%	47.1%
200 Motions	60.5%	50.3%	25.2%	54.3%	39.6%	40.4%	42.1%	64.4%	47.1%

Table 5.2: Results comparing the impact of the number of motions used to construct the initial tree. Each tree is built using the indicated number of motions and then populated using all 288 motions from the database. Performance increases slightly when increasing from 50 to 100 motions, but then drops off for higher input motion counts.

While these results indicate that our indexing scheme performs better when adaptive segmentation is not activated, for most of the following tests we use it simply to speed the process of gathering results.

Dataset size

The second test is to determine what effect the number of motions used to construct the tree has on its retrieval performance. We build trees using the first 50, 100, 150, and 200 motions in the dance dataset and then populate them with all 288 motions. The per-label and overall average scores are presented in Table 5.2. We can see from these results that building the initial tree with more of the motions in the dataset does not make a large difference in retrieval performance. Among all 4 cases, the variation in the retrieval accuracy is only around 3%. This could be because the first 50 motions contain at least a few examples of all 8 motion types, so the necessary information for accurate retrieval is represented in the tree when only using these clips. What is interesting, however, is that the retrieval performance degrades slightly once we build the tree with more than 100 clips.

	Label 1	Label 2	Label 3	Label 4	Label 5	Label 6	Label 7	Label 8	Average
Even label instances	61.6%	51.6%	24.0%	60.0%	43.5%	39.4%	42.1%	79.0%	50.1%
Arbitrary label instances	62.0%	52.8%	25.3%	51.6%	46.1%	39.1%	47.2%	69.3%	49.2%

Table 5.3: Results showing the impact of using an equal number of each of the 8 label types when building the tree. The first row uses the first 7 examples of each motion type while the second row uses 56 arbitrary label types from the first 56 motions in the repository.

Label distribution

In the previous tests we were building the tree with subsets of the dance motions chosen arbitrarily from the front of the dataset. This leads to an uneven distribution of the different dance step labels in the subset, but a distribution that matches that of the full dataset. We thought it would be interesting to see how the performance is affected if the label counts are distributed evenly through the subset used to build the tree, so for this test we selected 7 instances of each motion type at random with which to build the tree. The results of this test compared to the tree built with 100 motions from the previous test are presented in Table 5.3.

These results indicate that changing the distribution of motion types in the tree construction phase does not have much of an impact on the average retrieval performance. For example, motion types 1 and 2 made up a larger percentage of the original distribution than any of the other types, but in the even distribution the performance of retrieving type 1 motions decreases, while that for retrieving type 2 motions increases. When averaging the performance over all labels, there is essentially no difference between this tree and the tree generated from the first 50 motions.

Effect on performance from adding new motions

In the first test where we analyzed the performance of our index when using adaptive segmentation, we populated the vocabulary tree with the same dataset used to build its structure. In practical use, this would not make sense as the idea behind the vocabulary tree is to build a good tree from a representative subset of motion data, and then to populate the tree with as much data as possible, and to add more

Motions/Type	Label 1	Label 2	Label 3	Label 4	Label 5	Label 6	Label 7	Label 8	Average
50/segmented	58.9%	59.0%	24.1%	68.5%	51.3%	43.5%	34.3%	88.0%	53.4%
100/segmented	58.8%	58.1%	23.4%	67.9%	45.2%	39.4%	34.7%	87.3%	51.8%
150/segmented	59.7%	54.1%	20.9%	62.9%	46.3%	39.4%	42.5%	86.6%	51.5%
200/segmented	56.1%	52.4%	22.9%	60.3%	39.8%	40.2%	42.0%	78.2%	49.0%
250/segmented	60.7%	52.7%	25.9%	58.9%	42.8%	37.9%	44.3%	79.0%	50.3%
50/unsegmented	62.3%	63.2%	18.5%	65.7%	50.3%	43.5%	35.2%	88.9%	53.5%
100/unsegmented	59.3%	61.0%	22.6%	67.1%	42.9%	41.0%	37.6%	88.1%	52.4%
150/unsegmented	60.8%	55.1%	22.5%	61.0%	44.0%	42.6%	42.2%	86.7%	51.9%
200/unsegmented	57.1%	52.6%	24.5%	58.8%	37.6%	42.5%	40.9%	77.7%	49.0%
250/unsegmented	61.5%	53.1%	26.3%	57.0%	41.3%	41.3%	42.5%	79.0%	50.2%

Table 5.4: Results showing the change in performance when populating the index with an increasing number of motions, both with and without adaptive segmentation. Adaptive segmentation does not change the performance noticeably, but adding more motions actually decreases performance ratios slightly.

	Average retrieval time
50 motions segmented	7.2 s
250 motions segmented	11.9 s

Table 5.5: Results listing the average retrieval times for queries on indexes generated by building trees with 100 motions and populating them with either 50 or 250 motions.

motion data as it becomes available. In this test we build the tree using the best performing dataset seen so far, 100 clips not using adaptive segmentation. We then populate the tree with a varying number of motions both with and without adaptive segmentation, and calculate the performance as new motions are added. This is to simulate the situation where our index is used as intended, where it is built with a good representative set of motions and new motions are inserted over time to make the index more robust. The values from this test are presented in Table 5.4.

These results are surprising. In this test, adaptive segmentation does not seem to have much of an influence on the average performance of the system. Average scores between corresponding motion populations are nearly identical for the segmented and unsegmented tests.

The surprising result is that adding more motions to the index population actually seems to lower performance somewhat. This does not mean that the index is broken, as performance still hovers around 50%, meaning that the performance ratio says nearly the same. With an equal performance ratio, as more motions are added to the index more relevant motions will be retrieved for every query, and so the practical utility of the indexing system does increase. The most likly explanation for this result is the fact that with more motions in the database to search for a particular label type, it becomes more likely that correct clips could get confused with incorrect clips. The way we are scoring retrievals, when there are more instances of the query label type in the index, we require more correct retrievals in the ranked list to generate a good score. But our system seems to be good at placing correct motions near the top of the ranked list, and less adept at finding those near the bottom. The most obvious instance of this is the fact that if the query motion itself exists in the index, our system will always return it in the top spot on the ranked list. Therefore in an index with fewer instances of a given motion label, we score fewer spots in the list and the retrieval has a better chance at getting a good score.

We could compensate for this problem by scoring a fixed number of positions in the ranked list for every retrieval, but we chose a scheme that expects more out of an index with a larger population. We still believe this is the correct scoring scheme, and perhaps with further improvements on the system (allowing for partial matches, optimizing the input parameters), the benefits of our system could be demonstrated further.

Table 5.5 displays the average retrieval times for two of the above cases. These results indicate that populating the index with more motions does increase retrieval time, but a factor of 5 increase in the number of motions only leads to a factor of 1.6 increase in retrieval time.

5.2 Analysis

From these results, the general conclusion that can be drawn seems to be that our index method performs well as a content based retrieval system, but that varying the parameters of the index does not significantly affect performance. This is both an advantage and a disadvantage, as it means that our system will perform fairly well regardless of the conditions under which the index is generated, but conversely that it is not simple to change settings to make it perform significantly better.

In addition, it seems that using our current scoring metric and parameters, our index loses a bit of performance as more motions are added to the database. When going from 50 motions to 250, the performance drops by around 3%, based on our scoring method. This is not a large decrease, so it seems that our system will work decently well regardless of the number of motions indexed. We would need a larger annotated dataset to determine if our system can maintain its performance with a truly large database.

In nearly all cases our retrieval seems to be scoring around 50% on average using our scoring metric. Intuitively, this can be thought of as returning a correct result in every other position in a ranked list of results. In reality, many of the retrieved motions that were technically incorrect based on their label appeared very similar to the query motion when viewed on screen, but we must score based on the standard we have decided upon. A 50% retrieval rate seems like a good performance level when considering the quality of many other search engines, so we hope that with further development an index scheme such as this could be of value in solving the motion querying problem.

Chapter 6

Conclusions and Future Work

We have introduced a new technique for content-based retrieval of human motion capture data based on a vocabulary tree index of geometric motion features. Our index uses relational, binary geometric features to represent poses in the frames of motion, which allows the resulting index to be invariant to pose size, skeletal topology, and orientation since geometric features depend only on pose characteristics internal to each skeleton. The form of the index itself is that of a vocabulary tree consisting of a hierarchical set of nodes, each node representing the cluster center of a group of similar poses. The hierarchical nature of the vocabulary tree permits fast retrieval and insertion of new motions into the index. Retrieval tests on a dataset consisting of 8 different dance steps annotated by hand show that our index is able to recognize logically related motions and return a ranked list of the most similar motions in just a few seconds.

Since we have shown our indexing scheme to be effective at motion retrieval, an interesting goal for future work would be to incorporate the index into an actual database application allowing practical use by those who deal with motion capture data on a regular basis. To make the system more practical, it would be nice to be able to retrieve subsequences within longer motions when querying for particularly short actions. As it stands, our system merely returns the whole clips that are deemed to be the most similar to the input motion. Finally, in order to build a system that performs optimally, it would be advantageous to run a large battery of experiments to determine the best setting for each of the possible parameters. These would include finding the most influential features for parameterizing the motions, determining optimal clustering parameters, and generating the best possible representative set of motions for constructing the index.

Appendix A

Geometric motion features

Feature	Description	Threshold
F_1/F_2	Right/left hand moving forwards, in the direction of the normal to the plane spanning the neck, right hip, and left hip	1.2 hl/s
F_3/F_4	Right/left hand above the plane anchored at the neck, with normal in the direction from the root to the neck	0.2 hl
F_5/F_6	Right/left hand moving upwards, in the direction from the root to the neck	1.2 hl/s
F_7/F_8	Right/left elbow bent	110°
F_9	Hands far apart in the dimension between the left shoulder and right shoulder	2.5 sw
F_{10}	Hands approaching each other along the dimension between the left and right hands	1.2 hl/s
F_{11}/F_{12}	Right/left hand moving away from the root	1.2 hl/s
F_{13}/F_{14}	Right/left hand moving fast, absolute velocity	1.8 hl/s
F_{15}/F_{16}	Right/left foot behind left/right leg: right/left foot behind plane spanning the root, left/right hip and left/right toes	0.38 hl
F_{17}/F_{18}	Right/left foot raised above the ground plane	1.2 hl
F_{19}	Feet far apart in the dimension between the left hip and right hip	2.1 hw
F_{20}/F_{21}	Right/left knee bent	110°
F_{22}	Feet crossed over: right ankle closer than left ankle to plane anchored at left hip, with normal in the direction from the right hip to the left hip	0 hl
F_{23}	Feet moving towards each other along dimension between the left and right hip	1.2 hl/s
F_{24}	Feet moving apart along dimension between the left and right hip	1.2 hl/s
F_{25}/F_{26}	Right/left foot moving fast, absolute velocity	2.5 hl/s
F_{27}/F_{28}	Right/left humerus abducted: angle between spine and upper arm greater than threshold	25°
F_{29}/F_{30}	Right femur abducted: angle between spine and upper leg greater than threshold	50°
F_{31}	Core of body in bent position: root behind plane spanning right ankle, left ankle, and neck	0.5 hl
F_{32}	Spine nearly horizontal: angle between spine and vertical axis greater than threshold	70°
F_{33}/F_{34}	Right/left hand lowered: right/left hand close to ground plane	1.2 hl
F_{35}/F_{36}	Shoulders rotated right/left: test whether right/left shoulder closer than left/right shoulder to plane spanning right hip, left hip, and neck	0 hl
F_{37}	Body compressed vertically: lowest and highest extents of body are close together	3.0 hl
F_{38}	Body has wide horizontal coverage: project all joints onto ground plane and test whether diameter is above threshold	4.0 hl
F_{39}	Root moving fast, absolute velocity	1.6 hl/s

Table A.1: This table describes the 39 binary geometric features we use to parameterize our motions. They are adapted from [11]. Some features employ a threshold value that depends on skeleton size. These values are denoted by 'hl', 'sw', and 'hw', which stand for humerus length, shoulder width, and hip width, respectively.

Bibliography

[1] Marc Cardle, Michail Vlachos, Stephen Brooks, Eamonn Keogh, and Dimitrios Gunopulos. Fast motion capture matching with replicated motion editing. In *SIGGRAPH 2003, Sketches and Applications*. ACM Press, July 2003.

[2] Chih-Yi Chiu, Shih-Pin Chao, Ming-Yang Wu, Shi-Nine Yang, and Hsin-Chih Lin. Content-based retrieval for human motion data. In *Journal of Visual Communication and Image Representation*, volume 15, pages 446–466, 2004.

[3] CMU. Carnegie-mellon motion capture database, 2003.

[4] Christos Faloutsos. *Searching Multimedia Databases by Content.* Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[5] K. Forbes and E. Fiume. An efficient search algorithm for motion data using weighted pca. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 67–76, New York, NY, USA, 2005. ACM Press.

[6] Brendan Frey and Delbert Dueck. Clustering by passing messages between data points. *Science*, 315(5814):972–976, 2007.

[7] Eugene Hsu, Sommer Gentry, and Jovan Popović. Example-based control of human motion. In *Symposium on Computer Animation (SCA)*, pages 69–77, July 2004.

[8] Eamonn Keogh, Themistoklis Palpanas, Victor B. Zordan, Dimitrios Gunopulos, and Marc Cardle. Indexing large human-motion databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB)*, pages 780–791, 2004.

[9] Lucas Kovar and Michael Gleicher. Automated extraction and parameterization of motions in large data sets. *ACM Trans. Graph.*, 23(3):559–568, 2004.

[10] Guodong Liu, Jingdan Zhang, Wei Wang, and Leonard McMillan. A system for analyzing and indexing human-motion databases. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on management of data*, pages 924–926, New York, NY, USA, 2005. ACM Press.

[11] Meinard Müller and Tido Röder. Motion templates for automatic classification and retrieval of motion capture data. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 137–146. Eurographics Association, 2006.

[12] Meinard Müller, Tido Röder, and Michael Clausen. Efficient content-based retrieval of motion capture data. *ACM Transactions on Graphics*, 24(3):677–685, August 2005.

[13] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2, pages 2161–2168, June 2006. **oral presentation**.

[14] Alexander Sturn. Cluster analysis for large scale gene expression studies. Master's thesis, Graz University of Technology, 2001.