A Lexical Analogy to Feature Matching and Pose Estimation

John Horst
Intelligent Systems Division
National Institute of Standards and Technology (NIST)
Gaithersburg, Maryland, USA
E-mail: john.horst@nist.gov

Abstract— **We relate the problem of finding a correspondence between sensed and model features to that of finding a match between a random set of letters and words in a dictionary. The process is equivalent to hashing and the lexical perspective illuminates items such as design tradeoffs, computational complexity, and hashing function definition. A method for two-dimensional pose estimation based on this concept has been implemented. The method is local feature based and is robust to image warping, occlusion, illumination anomalies, and sensed feature generation errors. The method will work with certain modifications for three-dimensional data. The domain is restricted to translation and rotation invariant applications, since many pose estimation problems do not require scale and skew invariance. This non-affine constraint can reduce computational and storage complexity *vis-à-vis* a fully affine transformation invariant technique.**

Keywords— **pose estimation, hashing, hash tables, lexical, computer vision, feature matching, feature correspondence, pose clustering**

I. Introduction

THE cost of exhaustive search for the feature correspondence problem is fundamentally $O(m^s)$ where m and s are the numbers of model and sensed features, respectively [1][1]. This high cost is, in part, due to the fact that several sensed features can, and often do, match with a single model feature. Various types of noise further increase cost. Typically only a small percentage of the sensed features belong to the object of interest. For example, some features may belong to other artifacts in the image, some features may have been misinterpreted by the image processing engine, there may be errors in the imaging optics (*e.g.*, barrel distortion), and there may be errors due to illumination (*e.g.*, specular reflections).

The keys to pose estimation efficiency are reducing the search space, being robust to the various types of errors, designing the algorithm to exploit the asymmetry between on-line and off-line computing, and exploiting parallelism. Reducing the search space is greatly helped by a thoughtful representation of the model features before matching with sensed feature data is attempted. Hashing is such a representation. Tree searching techniques have the disadvantage of slower search but, if features are added to the model of a part, perfect hash tables (those that have no collisions) may need to be completely recomputed, whereas insertion and deletion is possible within a tree. Besides requiring recomputation, perfect hash functions are also hard to find. But if we allow collisions in our hashing functions, we also avoid the requirement of recomputation [2]. Recomputation is needed due to the fact that the output of perfect hash functions have a strong dependence on each individual element in the hash table; in contrast, adding an element to a tree affects the total tree structure very little.

Grimson [1] employs a tree search technique to do pose estimation. A complete global match of features is done by forming what is called an interpretation tree. Depth-first search is performed on that tree using unary and binary geometric constraints to reduce search while allowing for a certain amount of pairwise mismatches due to noise and feature occlusion. Further techniques are needed to achieve real-time computational speeds, since the tree search is exponential [3]. This is due to the fact that an interpretation tree is forming the $O(m^s)$ matches between model and sensed features. Incorporating these further techniques, the outline of Grimson's approach is as follows. Perform an initial pose clustering (as in [4]). Perform multiple interpretation tree searches on the reduced sets of matched features revealed through the clustering. Employ a mismatch tolerance threshold to terminate search to allow for sensor noise and occlusion.

The pose clustering method described by Stockman [4] does not reveal the additional combinatorial increase of the number of high level sensed and model "structures" that can be formed from lower level iconic features, *e.g.*, line segments and constant curvature arcs. For example, for m model features, we must form $m \cdot (m-1)$ structures from these features that would be sufficient for both matching and determining a pose estimate from each match. Consequently, if we have s sensed features, the matching problem is $O(m^2 s^2)$, assuming an exhaustive matching scheme, *i.e.*, one which forms all matches, equally weighted.

Several authors have discovered the advantage of hashing for pose estimation and object recognition [5] [6] [7] [8]. For example, Lamdan [5] defines interest point sets (three points per set) and expresses all model points in terms of affine transformation invariant parameters. These parameters are stored in a

[1] This complexity measure is derived in Appendix -A

hash table for use during online search. This hash table generation process (offline) is $O(m^4)$ for m model features (the *a priori* complexity of our non-affine method is always less than $O(m^4)$ for feature sets consisting of two, three, or four features as shown in Table II). Since the parameters are affine transformation invariant, if we happen to sense one or more of the same interest point sets, one can compute the same parameters and look them up in the hash table. This is followed by a voting procedure and automatic verification of object pose and recognition.

Most of these hashing papers say little about the particular hashing function, the handling of collisions, and the minimization of memory usage through quantization. A lexical analogy will help explicate these issues.

In our method, we match features by first forming a dictionary of "words." The "letters" in each word consist of quantized translation and rotation invariant geometric attributes for all possible unordered sets of r model features out of a total of m, making a total of $\binom{m}{r}$ possible sets. Each word in the dictionary is sorted by a canonical ordering of the letters within the word and the entire dictionary is indexed. The rules for canonical ordering are based on the type of feature set attribute and the value of the attribute. Each indexed location contains all model feature set words that match the range of letters in each dimension. The canonical ordering and subsequent indexing is a particular instance of hashing.

II. The Lexical Analogy

Our approach to pose estimation is based on search reduction through hashing. The challenge of hashing consists in generating a simple and efficient hashing function that minimizes complexity and memory usage. We will employ canonical symbol ordering as our hashing function. Such a hashing function is simple enough to describe for discrete valued items like letters in words. However, for the pose estimation problem, we will be dealing with geometric attributes of features, whose attributes are real valued quantities of nonuniform distribution in attribute space. Various types of errors further complicate pose estimation. Therefore, items such as design tradeoffs, computational complexity, and error handling are well illustrated if we start with a lexical analogy to feature matching.

Consider the following scenario. Select a set of letters at random from an alphabet and form all possible "words" from those letters. Or more formally, randomly select exactly s letters with replacement from an alphabet of m letters (m^s possible combinations with ordering). For each random selection of s letters, we have $s!$ orderings. Determine which of the $s!$ orderings match with words in a dictionary of n words, each of length s. We will now sketch four methods for accomplishing this scenario.

The first method is basically a brute force search. We form all $s!$ orderings (of the randomly selected letters) and do $s!$ searches through the entire dictionary for matches. This method requires $O((n+1)\cdot s!)$ operations to complete the search. This consists of $O(n \cdot s!)$ operations to search through the dictionary and $O(s!)$ operations to form the candidate words. $O(1)$ operations are required prior to search, and the only memory needed is that for storing the n words of the dictionary and the $s!$ candidate matches.

As a second method, index the dictionary so that each word in the dictionary has a unique location in a

$$\overbrace{m \times m \times \cdots \times m}^{s \text{ times}}$$

array. Develop indices for these words via canonical symbol ordering and determine the indices for each of the $s!$ orderings. Use those indices to see if there is a match in the indexed dictionary. For example, if $s = 3$, "`bat`" would be stored in location (2,1,20) and "`tab`" would be stored in the location (20,1,2). If we received the randomly ordered letters "`tba,`" we would determine the indices of each of the 3! orderings of these three letters, and find the matches in the indexed dictionary. In general, this requires $O(s!)$ operations for on-line search, *a priori* complexity is $O(n \log n)$ for creating the indexed dictionary, and $O(n + m^s)$ storage locations are required.

A third method is to form a new indexed dictionary, which first sorts the letters of each word in canonical (alphabetical) order. These "sorted letter" words are then indexed as in method two. This canonical ordering and indexing constitutes the hashing function. Collisions are now highly probable. For example, both "`bat`" and "`tab`" would be stored in the location (1,2,20), or "`abt`". However, collisions are of little concern, since it is known that finding perfect hash functions (functions that avoid all collisions) are usually not worth the effort [2]. Collisions will be of more concern when we look at feature matching in pose estimation. Resolving collisions involves both local and global consistency checking in pose estimation (see Figure 3. Search is accomplished by simply ordering the randomly selected letters alphabetically, forming the indices for those letters, and finding all the words associated with those letters in the indexed dictionary. Search via this method can be done in $O(n/m^s)$, which is $O(1)$ since $n < m^s$ and $n < m^s$ because each of the n words are of length s and there are only m^s total possibilities for forming a word of length s from an alphabet of m letters[2]. *A priori* complexity is $O(ns \log s + n \log n)$ for creating the ordered and indexed dictionary. This is because we need to sort the letters in each word, $O(ns \log s)$, and finally sort the whole dictionary, $O(n \log n)$. $O(n + m^s)$ storage locations are required[3].

Methods one to three increase search efficiency, but also increase in both memory usage and amount of *a*

[2]This complexity measure is derived in Appendix -B
[3]This complexity measure is derived in Appendix -C

TABLE I

Complexity of search methods 1, 2, 3, and 4 where s is the number of letters randomly selected, n is the number of words in the dictionary, and m is the number of letters in the alphabet.

	method 1	method 2
on-line search	$O((n+1)\cdot s!)$	$O(s^2)$
a priori	$O(1)$	$O(n \log n)$
memory	$O(n+s!)$	$O(n+m^s)$

	method 3	method 4
on-line search	$O(1)$	$O(1)$
a priori	$O(ns \log s + n \log n)$	$O(ns \log s + n \log n)$
memory	$O(n+m^s)$	$O(n+(m/p)^s)$

priori complexity. A summary of the quantitative complexity of the three methods is found in Table I.

It is easy to see that the number of memory locations required under methods 2 and 3 can be large indeed for certain m and s. For example, if $s = 8$ and $m = 26$ (8 letters picked out of the alphabet with replacement), we have roughly 2×10^{11} storage locations required. There is, however, a tradeoff between the amount of storage and the search required, if we allow more collisions. If we are willing to do some limited search for matches among the collisions, we are not forced to provide a unique location for every one of the m^s possible combinations of letters. For example, maintaining 2×10^{11} storage locations for a $n = 1000$ word dictionary (containing words of length $s = 8$) can make for a very sparse array.

A simple solution to this inefficiency of storage is to modify the indexed array so that the number of locations is closer to the number of words to store. To accomplish this we can design the locations to correspond to a range of combinations of letters. Such a range can be applied in each dimension of the array, (i.e., each letter position in the word. If we keep the resolution constant over all dimensions, we get $O(n+p^s)$ locations where $p \le m$. This gives a

$$\overbrace{p \times p \times \cdots \times p}^{s \text{ times}}$$

array to store the words of the dictionary. An application of a uniform range of letters ($p = 6$) to a few $s = 3$ letter words is shown in Figure 1. If the n words are distributed evenly throughout the s-dimensional array of m^s locations, a good choice of p is a p such that $(m/p)^s \approx n$. If the n words are not so evenly distributed but exhibit some "clumpiness," we will either choose a non-uniform partition or else choose a p such that $(m/p)^s > n$.

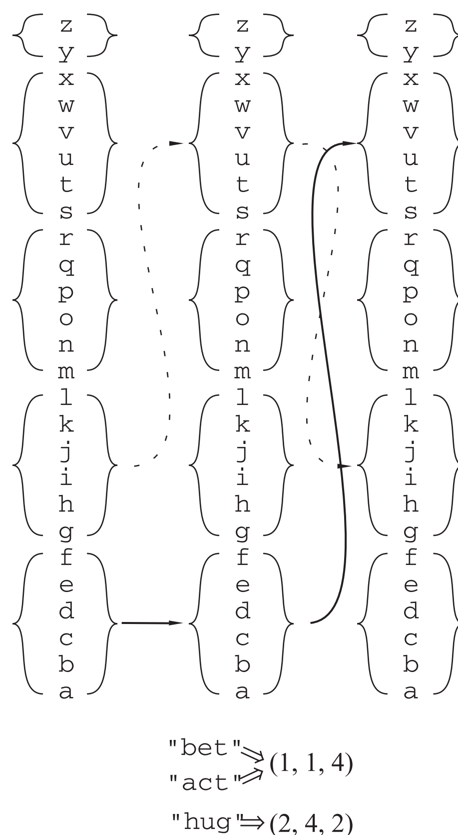

Fig. 1. Forming indices for several example words given a uniform range, $p = 6$, for a dictionary of $s = 3$ letter words. The words "bet" and "act" are stored in the location (1,1,4) and the word "hug" is stored in the location (2,4,2). To store all three letter words for $p = 6$, we need a block matrix of size $p \cdot p \cdot p = 216$. This is much less than the $m \cdot m \cdot m = 17576$ storage locations that would be required if we did not choose some $p \ll m$ as our range.

A non-uniform partition is one in which either the resolution varies over each dimension or the resolution is also non-uniform within each dimension. In the former case, we have an $p_1 \times p_2 \times \cdots \times p_s$ array to store the words of the dictionary.

Now that we have negotiated a tradeoff between search and memory storage, we note that if we design things well, we can lose little in on-line search and gain much in memory usage. Since on-line search will have approximately $n/(\lceil (m/p))^s$ operations and since $1 \le m/p \le m$, $(m/p)^s < m^s$[4]. This gives on-line search cost as $O(n/(m/p)^s)$. But, if $n \approx (m/p)^s$, $O(n/(m/p)^s) \approx O(1)$[5]. So, by choosing p such that $n \approx (m/p)^s$ and if the n words are evenly distributed within the uniform array, we have a near optimal solution for the uniform partitioning case.

Table I also includes the costs of method 4. Compared to methods 1, 2, and 3, method 4 significantly

[4] $\lceil$ is the ceiling operator

[5] This complexity measure is derived in Appendix -D

reduces storage, without greatly increasing either on-line or off-line effort. A search for an exact match must proceed through the words stored in each indexed location. In the case of pose estimation, exact matches are not desired, so we will need to utilize all the values in the indexed location for further processing. The number of words in each storage location will vary widely from location to location throughout a single dictionary and over different types of dictionaries. The number of words in any single location can be bounded by choosing non-uniform letter ranges. In the case of pose estimation we bound the search at each location in two ways, 1) by using non-uniform partitioning of the "word space," and 2) by allowing Type I errors (*i.e.*, the system misses a correct match), as we will explain in the next section.

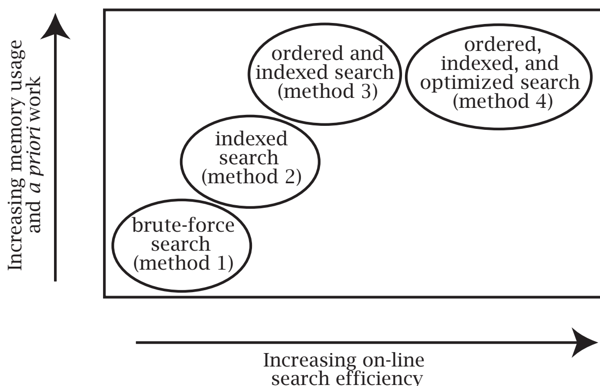

Fig. 2. Qualitative view of the on-line efficiency and memory usage of different search methods

The relationship between on-line complexity, *a priori* complexity, and memory use for the four methods is illustrated qualitatively in Figure 2.

It should be clear now that a non-uniform partitioning can be chosen off-line that suits the "clumpiness" of our dictionary. In regions where there are many ordered words not far away from each other, we are able to choose a non-uniform partitioning with on-line search complexity of $O(1)$. The distance metric for words is the following. If $P = p_1 p_2 \cdots p_s$ and $Q = q_1 q_2 \cdots q_s$ are words, p_i and q_i are letters, and the metric is $\sum_i |p_i - q_i|$.

We have been assuming up until now that we only match words of length s in our dictionary. We have also assumed that our dictionary only contains words of length s. However, we may also want to find matches in a more normal dictionary containing words of length less than or equal to s contained in the set of s randomly chosen letters. The addition of the NULL letter to the list of n symbols will accomplish this. In this case, we need to form all the possible words of length $i \leq s$, $i = 1, 2, \cdots, s$. Since we are selecting i letters out of s without regard to ordering and without replacement, we have to perform the following number of searches in the ordered and indexed dictionary, $\sum_{i=1}^{s} \binom{s}{i} = 2^s - 1$ As in method 3 described above, we order the letters in each of these words in alphabetical order and search for each ordered word in our new ordered and indexed dictionary. This new dictionary contains NULL letters to allow for words of varying length. However, the manner we have chosen to do pose estimation does not require the use of the NULL letter. We will employ two dictionaries of different symbol sets and each of those dictionaries contains words of length exactly equal to s.

III. Pose Estimation

In the previous section, we matched randomly selected letters to words in a dictionary by creating an array of locations organized so that the words contained in each location have the same set of letters. We expanded this basic concept to define each location to contain words that have sets of letters that are all within a local range of one another. For example, if our range is six letters (as in Figure 1), “`bet`” and “`cat`” are stored in the same location, but “`bat`” and “`but`”, are not, since “`a`” and “`u`” are not within the range.

We exploit this technique for pose estimation by equating “location” of letters in the lexical analogy to “feature set attribute” in pose estimation. Our feature set attributes are roughly analogous to the unary, binary, and tertiary geometric constraints found in the literature [1]. We generate pose invariant feature set attributes and call them “letters.” The ordered attributes of a feature set form the “word” for that feature set. The canonical ordering of the letters in the feature set word is equivalent to the alphabetical ordering of letters in each word in the lexical analogy. The canonical ordering of the letters in the feature set word (along with quantization) constitutes the hashing function for feature matching. We begin our discussion of pose estimation with an overall view of the component parts of the pose estimation task.

Our feature-based pose estimation method can be divided into the following subtasks: sensed and model feature generation, model dictionary generation, sensed feature set word generation, feature matching (or feature set word search), globally consistent pose checking, and pose clustering. Only model dictionary generation, sensed feature set word generation, and feature matching employ the lexical analogy of the previous section. We will focus our discussion on word generation and feature matching, but also describe the other subtasks in some detail. We illustrate the subtasks and data flow of the pose estimation task in Figure 3.

The goal of model and sensed feature generation is to generate features that are in the same format and are sufficient for pose invariant feature set word generation. Line segment and constant curvature arc features are sufficient. The parameters of these features are passed to model dictionary generation and sensed feature set word generation subtask.

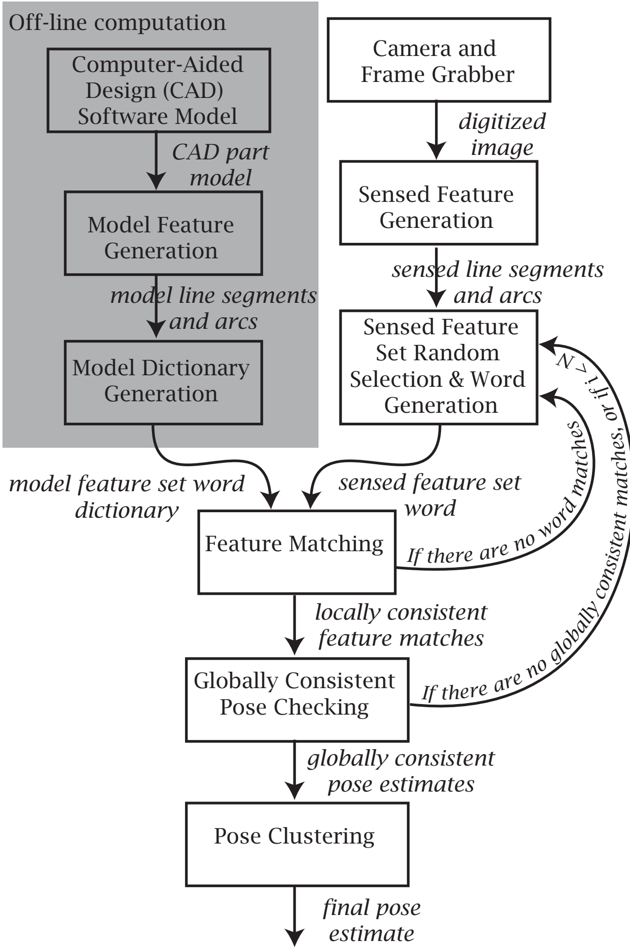

Fig. 3. The overall pose estimation task

A. Model dictionary generation

Model feature set word dictionary generation is accomplished as follows. We form all possible feature sets of r features out of the total of m model features, which is a total of $\binom{m}{r}$ feature sets. Pose invariant feature sets of s attributes are generated for each feature set. This is analogous to the canonical ordering of the letters in each word. We generate four separate dictionaries. One for sets containing all line segments, one for sets containing exactly one arc, one for sets containing all arcs, and one for all other sets. The four separate dictionaries are necessary, if feature set attributes are different for the four dictionaries, as is so in this case. This is equivalent in the analogy to having four dictionaries with different alphabets.

The goal in choosing these dictionaries is to create appropriate transformation-invariant attributes that preserve or amplify true differences between sets of features. Another possibility is to create a uniform transformation that is independent of the nature of the features in each set. This is the approach taken by Lamdan [5] and has the advantage of greater simplicity of feature set attribute "alphabets."

For the “line-segments-only” dictionary, we form all possible differences in orientation between pairs of line segment. This gives $\binom{r}{2}$ differences for r features in the set. We order these differences by magnitude to form the letters of our model feature set word for the line-segments-only dictionary. This is equivalent to alphabetical ordering of the letters in the lexical analogy.

For the “exactly one arc” dictionary, we first form the vector of shortest distances from the arc center to the lines formed from each of the line segments. The first letter is the arc radius plus the sum of these distances. The next $r-1$ letters form the ordered vector of these distances. This is again equivalent to alphabetical ordering of the letters in the lexical analogy. Each word in this dictionary has $s = r$ letters.

The “all arcs” dictionary also has $s = r$ pose-invariant feature set attributes. For the center of each arc, we sum the radius plus the sum of the distances from the center of each of the $r-1$ other arcs. This gives r values, which we use to form the ordered vector of these distances. The ordered vector forms the $s = r$ letters in the word.

In the “otherwise” dictionary, the first letter is the number of arcs in the set. Then, for each arc, we sum the radius plus the sum of the distances from each of the other arcs. We form an ordered vector of these values. We then choose the arc that has the smallest radius plus the sum of distances, and we form the ordered vector of distances from this arc center to the lines formed from each of the line segments. The number of arcs and these two vectors form the $s = r+1$ letters in each word of this dictionary.

The complexity of dictionary construction (*i.e.*, *a priori* complexity) is $O(\binom{m}{r} s \log s + \binom{m}{r} \log \binom{m}{r})$. As long as $r = 2$, 3, or 4, $\binom{m}{r} s \log s + \binom{m}{r} \log \binom{m}{r}$ is always less than the *a priori* (dictionary building) complexity of m^4 found in [5]. We used $r = 3$. This reduction in *a priori* complexity is in part due to the fact that we are using a non-affine constraint (translation and rotation only).

We experimented with various values for r and found that $r = 3$ seems to give the best combination of speed, manageable dictionary size, and pose estimation accuracy.

We now have four dictionaries that consist of unordered lists of words with ordered letters. Following search method 4 in the lexical analogy, we need to form the indexed list, in which we store the words. For each dictionary, we must choose what the range of values will be for determining the storage location size and extent. In the lexical case we have discrete letters and we assume that there is no error in the transmission and interpretation of the letters. In pose estimation, we have two key differences, 1) the presence of errors and 2) the “letters” are real valued (excepting the number of arcs letter). The use of a range of values for the letters becomes essential for pose estimation. So we have a nearly exact analogy with method 4 described in the previous section. The key difference is the presence of errors in transmission and interpretation.

Handling errors effectively translates to our choice of range of feature set attribute (letter) values in each dictionary. This choice is guided by the following tradeoff. If we make our range of letters too large, we will have too many potential matches at each location where the true match lives, increasing search. However, if the range of letters is too narrow, the location may not contain the correct match due to various types of measurement errors. In order to avoid too large a range, we bound the search at each location in two ways, 1) by using non-uniform partitioning of the "word space," and 2) by allowing Type I errors (the system misses a correct match). In order to avoid too small a range, we simply make sure the range is not too small by experimentation. The range of values is obtained experimentally. There are many sources of error in our system, including camera calibration error, feature distortion error (*e.g.*, image warping), coordinate system transformation parameter measurement error, lighting errors (*e.g.*, spurious reflections), image processing errors, and pose estimation averaging errors. The search for an optimal range computation as a function of all the errors was not within the scope of this research. Therefore, we select a range for each dimension of each indexed list that seems to give successful pose estimates while minimizing execution time.

B. Sensed feature set word generation

Sensed feature set word generation is done exactly as model feature set dictionary generation, except that we do not generate words for all possible combinations of sensed feature sets. Once we have generated our indexed model dictionaries, we randomly select a set of r sensed features out of the total of s features. Random selection implies that no attempt is made to weight certain features to be more likely candidates than others. We form the sensed feature set word and order the letters in the sensed word just as we did for all the words in the model dictionary. We then compute the indices for that word using the same ranges employed in generating the indexed model dictionaries.

C. Word search

Using the sensed word indices, we get all the model words stored in the dictionary at that location. If there are no words at that location, we randomly select another set of sensed features and compute another set of sensed word indices. If there are words at the location, we send these words to the globally consistent pose checking phase.

Global consistency means that we must examine the pose transformations feature by feature within each candidate match. We check if the pose transformations required to put each of the r sensed features in the matched set into correspondence with the candidate model feature matches are the same, or nearly so. In order to achieve bounded time execution, we also require that we put a bound on the number of candidate matches that are input to this subtask. If we miss a match, it is of little consequence, since we will loop back and generate another sensed word from a randomly selected feature set.

If we find a non-empty set of globally consistent pose estimates, we return and randomly select a new sensed feature set of r features, until we receive N non-empty sets of globally consistent pose estimates. $5 \leq N \leq 20$ seems to work well for our experiments.

D. Pose clustering

The final list of globally consistent pose estimates for all N cycles is input to the pose clustering subtask. In pose clustering, because we often expect more wrong answers than right, we cannot use common averaging, such as the mean or median of the data. This situation is dauntingly typical in computer vision, where we often encounter a preponderance of “replacement errors,” *i.e.*, utterly wrong errors. For example, with rectangular shaped objects, we often measure orientation estimates that are off by $\pm\pi/2$ rad or π rad. Pose clustering allows us to find the correct answer even when common averaging would miss it altogether.

The pose clustering technique is simple and is designed for efficient computation. Since the best pose estimates have been culled already, there is no need for an exhaustive pose clustering algorithm. All candidate poses have three real-valued measurements for x, y, and θ. We find the largest and next-to-largest clusters of measurements in each dimension, x, y, and θ, independently. A cluster is defined as the collection of points lying with each of the bins of a simple histogram. For example, if there are N pose estimates, then for the set measurements, x_i for $i = 1, 2, \ldots N$, we find two subsets, x_{i_j} for $j = 1, 2, \ldots L \leq N$ and x_{i_k} for $k = 1, 2, \ldots M \leq N$. The first subset is the set with the largest number of points fitting into one cluster. The second subset is the set with the second largest number of points fitting into one cluster. For two clusters in each of three dimensions, we have $2 \cdot 3 = 6$ clusters in all the dimensions. With these we form the $2^3 = 8$ possible pose clusters in three dimensional pose space (x, y, and θ). This greatly reduces the amount of space in three dimensions over which we must search. We find the cluster in three dimensional pose space with the most pose estimates in it, compute the mean of the points in that cluster, and declare the mean as the final pose estimate. This simple and efficient pose clustering technique depends on the fact that the matching effort has already eliminated many of the erroneous estimates.

E. Pose estimation computational complexity

To describe the complexity of the pose estimation algorithm we have only to look at the lexical analogy, since the heart of the pose estimation algorithm is based on it. A few additions to complexity must be considered. On-line search will have approximately $n/(n/p)^s$ =

$\binom{m}{r}/(\binom{m}{r}/p)^s$ operations, where n is the number of words (model feature set attributes) in the dictionary. To optimize on-line search and storage efficiency, we want to select a p such that $\binom{m}{r} \approx (\binom{m}{r}/p)^s$. If we select $p \approx \binom{m}{r}/\sqrt[s]{\binom{m}{r}}$, we have optimum values for on-line search and memory usage. Since $s \geq 2$, this value for p also meets the required constraint, $1 \leq p \leq \binom{m}{r}$. Therefore, on-line search complexity is $O(1)$ just as it was in method 4 of the lexical analogy[6]. On-line complexity, *a priori* complexity, and memory usage for pose estimation are summarized in Table II.

TABLE II

COMPLEXITY AND MEMORY USAGE OF METHOD 4 APPLIED TO POSE ESTIMATION WHERE s IS THE NUMBER OF FEATURE SET ATTRIBUTES (LETTERS), r IS THE NUMBER OF FEATURES IN THE ATTRIBUTE SET, m IS THE NUMBER OF MODEL FEATURES, AND p IS THE NUMBER OF LETTERS IN THE PARTITIONING RANGE

	pose estimation
on-line search	$O(1)$
a priori	$O(\binom{m}{r} s \log s + \binom{m}{r} \log \binom{m}{r})$
memory	$O(\binom{m}{r} + (\binom{m}{r}/p)^s)$

The overall pose estimation algorithm has two loops as can be seen in Figure 3. The inner loop is necessary, because we will often encounter incorrect sensed features of various types. The bad features may be due to features from other parts, poor lighting, optical effects such as image warping, camera calibration errors, or correct sensed features not existing in the model set. The outer loop is necessary as well. Even though we will encounter word matches, most of them are not globally consistent, *i.e.*, the pose transformations required to put each of the features in the set into correspondence are not equal.

Furthermore, even globally consistent pose estimates can be wrong. Wrong estimates typically arise from two sources. One, due to replacement errors, where the pose estimate is off by a large amount, and the other, due to small errors due to random noise in the measurement process. Therefore, we need to collect more than one globally consistent pose estimate to guarantee success in the pose clustering phase and get an accurate pose estimate.

Additionally, depending on the nature of the data, we often find a very large set of candidate matches in a single hash table location that need to be checked for global consistency. To assure bounded execution time, we must limit the number of matches checked for global consistency. However, this may cause us to miss the correct globally consistent match. We solve this problem by looping back, namely, selecting another sensed feature set at random and trying again.

[6]This complexity measure is derived in Appendix -E

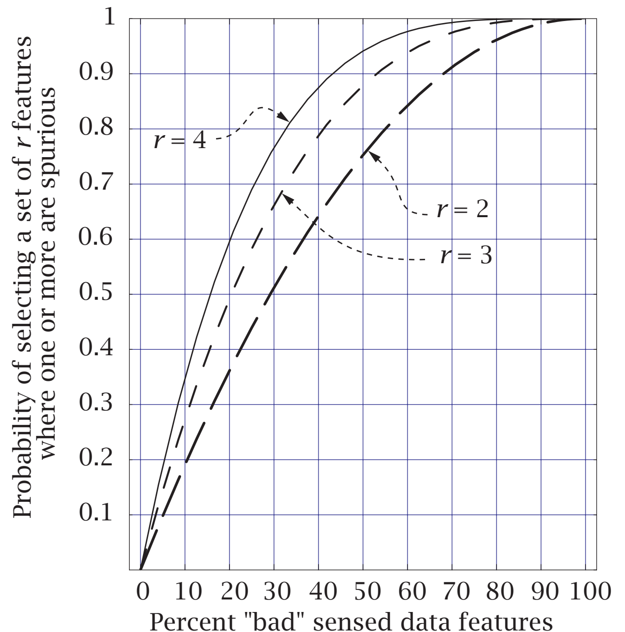

Fig. 4. The probability of selecting a set of r features out of 100 total features as a function of percentage of spurious features for a few values of r.

F. Random selection and computational complexity

Random selection affects algorithm complexity. If we randomly select features, search time can be unbounded, if the sensed features sets containing attributes find no match in the model dictionary. Clearly, we must bound search to declare failure when we are unable to find matches within sufficient time. If we have enough features in the total set of sensed features that have real matches in the total model feature set, we can guarantee statistically that we will find a correct pose estimate within bounded time. To illustrate this, Figures 4 and 5 show the probability of getting a sensed feature set containing one or more "bad" features, *i.e.*, sensed features having no true match with features in the model feature set. In Figure 4, this probability is a function of the total percentage of bad features in the sensed feature set. A family of curves is generated for, $r = 2$, 3, and 4 (r is the number of randomly selected features). In Figure 5, this probability is a function of r and a family of curves is generated for certain values of the total percentage of bad features in the sensed feature set.

IV. CONCLUSION

The lexical perspective illuminated many of the trade-offs inherent in the feature matching phase of the pose estimation problem. It not only allowed us to see clearly

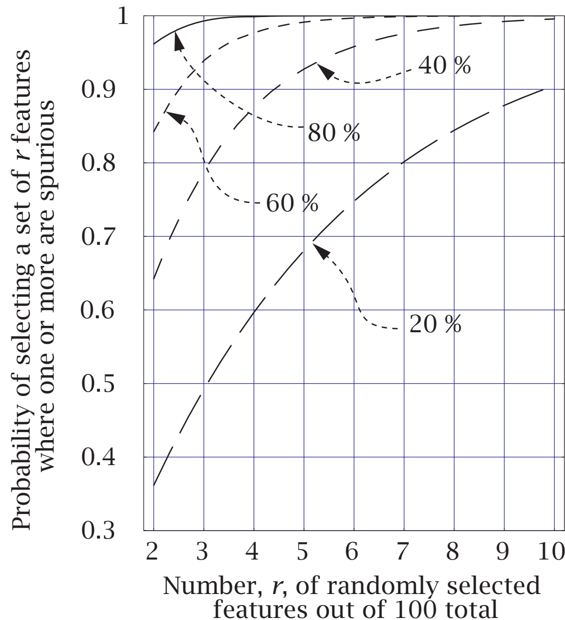

Fig. 5. The probability of selecting a set of r features as a function of r for a few different percentages of bad sensed features.

that hashing is efficient, but that certain tradeoffs are required to achieve quasi-optimal memory usage. Based on this analysis, we have described an asymmetrical pose estimation system that places more computational and complexity burden on the off-line algorithms in order to speed up the on-line computations. Efficient use of indexing can be done only when the algorithm is designed to allow the effective ordering of the search space (via the hashing function) prior to the matching phase. A summary of the analogies between the lexical perspective and pose estimation is given in Table III.

Our pose estimation method is not invariant to scale, since in many applications the scale of the sensed features can readily be gotten from other measurements, namely, height of the camera from the object. We have implemented and integrated this algorithm in an inspection system for the purpose of automating part set-up in manufacturing.

We performed tests on a part with many orthogonal and symmetrical features. In this case, errors in pose estimation were most often found when the estimated orientation was off by integer multiples of $\pi/2$ rad.

The algorithm degrades when the percentage of part features in the image is low, as illustrated in Figures 4 and 5.

To make this method work for three-dimensional pose estimation, one must choose a set of feature set attributes that are fully affine. Lamdan has described such attributes [5]. However, this system we've described is independent of the particular type of feature set attributes chosen.

The pose estimation system is independent of the particular characteristics of a single object or family of objects (*e.g.*, prismatic). This is done by defining parameters that can be adjusted according to various minimization and optimization criteria such as speed of execution, number of potential matches, and *a priori* effort. However, we adjusted these parameters manually. To automate this system we would need to create adaptive parameter adjustment through some cost function of the minimization and optimization criteria.

All operations are coded in Mathematica™[7]. A version of the on-line portion of the code is in C++. The C++ version executes in about 0.1 s for a fairly uncluttered image. Figure 6 shows sensed features in blue overlaid onto model features in green after the computation of pose via our method. Note the presence of occlusion, warping, spurious data, and missing data in this data set, but not very much clutter.

It is clear that hashing helps achieve efficient pose estimation. However, the choice of feature types and feature set attributes varies widely depending on the class of object for which we desire a pose estimate. The choice of these attributes and feature types is *ad hoc*, and much work needs to be done to understand how to automate this choice to achieve algorithm design efficiency.

TABLE III

A SUMMARY OF THE ANALOGIES BETWEEN THE LEXICAL PERSPECTIVE AND POSE ESTIMATION.

Lexical Problem	Pose Estimation
Choose an alphabet of symbols	Choose feature set attribute types
Choose a set of words	Form the feature set attributes from all possible model feature sets
Order letters in each word in alphabetical order	Sort feature set attributes (the letters) in each model feature set attribute word in canonical order
Sort ordered words in alphabetical order	Form and sort (in canonical order) model feature set attribute words
Choose the letter range, p, (see Figure 1) and store all words into locations in a 2D array	Choose ranges for each feature set attribute, quantize and store all model feature set attribute words into locations in a 2D array
Randomly pick letters from the alphabet, sort the letters, determine the indices in the 2D array for this "word"	Randomly select a set of r sensed features, form the sensed feature set attributes, order them into a word, quantize the attribute values, determine the appropriate indices for this word in the 2D array
Check to see which of the potential word matches at the appropriate location in the array actually match perfectly	Check the global consistency of the pose transformation of model features (in each candidate set from the matching location in the array) to each randomly selected sensed feature set

[7]Certain commercial products are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply any judgement by the National Institute of Standards and Technology concerning these products, nor is it intended to imply that they are necessarily the best available for the purpose.

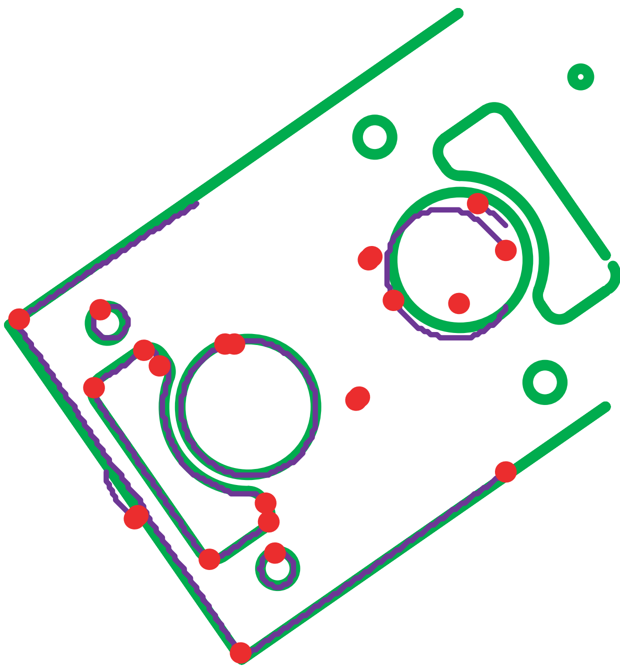

Fig. 6. An example pose estimate. The blue features are sensed constant curvature arcs and line segments. The red dots are the start of each sensed feature. The model features are green.

References

[1] W. E. L. Grimson, *Object Recognition by Computer: The Role of Geometric Constraints*, MIT Press, 1990.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 3, Sorting and Searching*, Addison Wesley Longman, second edition, 1998.

[3] W. E. L. Grimson, "The combinatorics of local constraints in model-based recognition and localization from sparse data," *Journal of the ACM*, vol. 33, pp. 658–686, 1986.

[4] G. Stockman, "Object recognition and localization via pose clustering," in *Computer Vision: Advances and Applications*, Rangachar Kasturi and Ramesh C. Jain, Eds. 1991, IEEE Computer Society Press.

[5] Y. Lamdan, J. Schwartz, and H. Wolfson, "Affine invariant model-based object recognition," *IEEE Transactions on Robotics and Automation*, vol. 6, no. 5, October 1990.

[6] A. S. Wallack, J. F. Canny, and D. Manocha, "Object localization using crossbeam sensing," *1993 IEEE International Conference on Robotics and Automation*, vol. 1, pp. 692, 1993.

[7] J. Edwards and R. Shoureshi, "Recognition of multiple objects using geometric hashing techniques," *Proceedings of the 32nd IEEE Conference on Decision and Control*, vol. 2, pp. 1617, 1993.

[8] S. Verreault, D. Laurendeau, and R. Bergevin, "Pose determination for an object in a 3-d image using geometric hashing and the interpretation tree," *Canadian Conference on Electrical and Computer Engineering*, vol. 2, pp. 755, October 1993.

Appendix

A. Derivation of $O(m^s)$ for feature correspondence problem

We assume that model features will not be broken up; multiple model features will not match with the same sensed feature, *i.e.*, model features are unique and complete. However, multiple sensed features may match to a single model feature. This is common in computer vision when there is occlusion and other errors. This is best revealed through an example. Consider Figure 7. Since every sensed feature must be matched with each model feature, the following sets of matches are necessary to determine pose, $\{m_0s_0, m_0s_1, m_0s_2\}$, $\{m_0s_0, m_0s_1, m_1s_2\}$, $\{m_0s_0, m_1s_1, m_0s_2\}$, $\{m_0s_0, m_1s_1, m_1s_2\}$, $\{m_1s_0, m_0s_1, m_0s_2\}$, $\{m_1s_0, m_0s_1, m_1s_2\}$, $\{m_1s_0, m_1s_1, m_0s_2\}$, $\{m_1s_0, m_1s_1, m_1s_2\}$. A total of $2^3 = 8$ sets of matches are needed, which is generalizable to m^s. This is the same as the number of unique ways to pick s symbols with replacement out of a bin of m unique symbols. Note that only the set $\{m_0s_0, m_1s_1, m_1s_2\}$ contains the correct match. If there are spurious sensed features (features that don't belong to the model), we still need to match with it, but just need some way to be robust to these erroneous matches. Our solution is to choose a small, random subset of all features, s_i, and match with all model features; matching now being done with a hash table of feature set attributes.

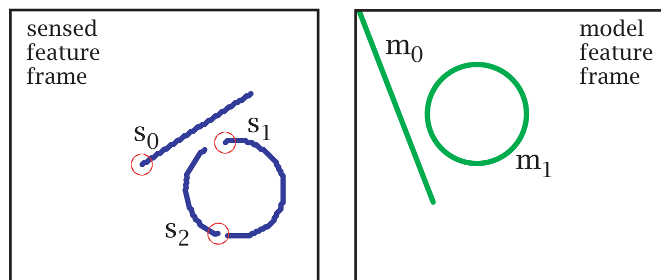

Fig. 7. Variables m_i are model features and s_j are sensed features.

B. Derivation of $O(1)$ on-line search complexity for method 3

If n is the number of words (all of length s) in the dictionary and m is the number of letters in the alphabet, then, an s-dimensional array of m locations per dimension is required to store the n words. This array has m^s locations. Now it is clear that no language will have words defined for every possible combination of letters, in fact, $n \ll m^s$, typically. Since our array is indexed, we can immediately access the location in the array we seek. However, there may be more than a single word at some locations, but it will never be greater than $s!$ and usually much less than that, particular for large s. Therefore, it can be concluded that we have $O(1)$ complexity.

C. Derivation of *a priori* complexity for method 3

We order the letters in each word in the dictionary alphabetically (canonically) and then we must sort canonically this new dictionary of words with canonically ordered letters. Each new word has pointers to the $< s!$ words that contain the same letters as the canonically ordered word, *e.g.*, "`bat`" and "`tab`" would both be stored in the location (1,2,20) or "`abt`". It is known that the best known sorting methods are $O(n \log n)$ and we also must order the letters in each of n words, so the total complexity is $O(ns \log s + n \log n)$.

D. Derivation of on-line search complexity for method 4

As before, n is the number of words (all of length s) in the dictionary, m is the number of letters in the alphabet, and p is the size of the partition. An s-dimensional array of $\lceil(m/p)$ locations per dimension is required. This array has $(\lceil(m/p))^s$ locations. We want to distribute the n words of the dictionary evenly throughout the dictionary. Now the occurrence of letters in an English dictionary are not equally likely, but, even so, if we choose p such that $n \approx (m/p)^s$, we will get not too many words in each location. Since on-line search will have approximately $n/(\lceil(m/p))^s$ operations and since $1 \leq m/p \leq m$, $(m/p)^s < m^s$. This gives on-line search cost as $O(n/(m/p)^s)$. But, if $n \approx (m/p)^s$, $O(n/(m/p)^s) \approx O(1)$. Therefore, it can be concluded that we have $O(1)$ complexity.

E. Derivation of on-line search complexity for pose estimation

Again, n is the number of words (all of length s) in the dictionary, m is the number of letters in the alphabet, p is the size of the partition, and r is the number of features in a feature set. This gives the same derivation as in the section above which derives the on-line search complexity for method 4, with the exception that $n = \binom{m}{r}$. $\binom{m}{r}$ is the number of words in the model dictionary of feature set attribute words.