Best-First and Ten Other Variations of the Interpretation-Tree Model Matching Algorithm

Robert B. Fisher
Dept. of Artificial Intelligence, University of Edinburgh
5 Forrest Hill, Edinburgh EH1 2QL, Scotland, United Kingdom
Telephone: 44-(31)-650-3098
Fax:44-(31)-650-6899
email:rbf@aifh.ed.ac.uk

Abbreviated title: The Best-First Model Matching Algorithm

Abstract

The best known control algorithm for symbolic model matching in computer vision is the *Interpretation Tree* search algorithm, popularized and extended by Grimson, Lozano-Perez, Huttenlocher and others. This algorithm has a high computational complexity when applied to matching problems with large numbers of features. This paper examines eleven variations of this algorithm in a search for improved performance, and concludes that a best-first algorithm has greatly reduced theoretical complexity and runs much faster than the standard algorithm.

KEYWORDS: model-based vision, model matching, object recognition, search

1 Introduction

The most well-known control algorithm for symbolic model matching in computer vision is the *Interpretation Tree* (IT) search algorithm, as used by Grimson and Lozano-Perez [12]. The algorithm searches a tree of potential model-to-data correspondences, such that each node in the tree represents one correspondence and the path of nodes from the current node back to the root of the tree is a set of simultaneous pairings. This model matching algorithm is a specialized form of the general AI tree search technique, where branches are pruned according to a set of consistency constraints (usually related to the geometric structure of the problem). The goal of the search algorithm is to maximize the set of consistent model-to-data correspondences in an efficient manner. Finding these correspondences is a key problem in model-based vision, and is usually a preliminary step to pose estimation, identity verification or visual inspection.

Unfortunately, this algorithm has the potential for combinatorial explosion. This has prompted researchers to develop techniques for pruning the trees, thus limiting the number of matches considered. The main technique commonly used is based on **pruning constraints** [12] (which locally reject pairings that are inconsistent, and hence eliminate all of the search that might further extend this inconsistent pairing) and **early termination** [14]. The latter stops search: (1) completely at the first hypothesis with a given T pairings, or (2) on any path when it is impossible to make

T pairings with the remaining potential matches. (Grimson [13] - Section 12.3 page 350 - has given an analytic derivation of the termination threshold T as a function of the probability of false matches being accepted.) However, even with these effective forms of pruning, the algorithm still can have exponential complexity, making it unsuitable for use in scenes with many features.

This paper discusses and analyses eleven extensions to the standard IT algorithm that have the potential to reduce the search space (including three original algorithms and the others are adapted from either techniques in general use or from published reports):

1. switching from standard search to a geometric algorithm as soon as possible (with and without spatial indexing),
2. using alignment methods to expand the search tree once several features are matched,
3. partitioning the model features into a hierarchy, using either binary feature or subcomponent pose consistency testing,
4. use of "model invocation" methods to eliminate unmatchable combinations at the start,
5. re-ordering the search space,
6. only using model features once,
7. partitioning features by visibility (in the 3D case) and
8. use of typed features.

As reported by Grimson [14, 15], the main cause of the exponential complexity is the use of a "wildcard" match feature. Hence, we also:

9. explore a slightly different search tree using consistent pairs without using a wildcard and
10. extend this algorithm to form a "best-first" matching algorithm.

Finally, although this is a slight change to the problem assumptions, we also consider:

11. exploiting an ordering of the model and data features,

to test the effect of strengthening the problem information.

The results of the paper show that several of the variations produce greatly improved performance in both theory and as applied to real data, and in particular, the best-first algorithm is greatly superior to the standard interpretation tree. (All algorithms except for the best-first, which is new, were briefly compared in [9].)

2 The Standard Interpretation Tree Algorithm

Consider a set $\{ d_i \}$ of D data features and a set $\{ m_i \}$ of M model features. Then, the root of the interpretation tree has no pairings. The first level expands the root node to pair all of the M model features with data feature d_1. The second level in the tree expands each of these nodes to pair all model features with data feature d_2 (multiple use of m_i are allowed), and so on. The expansion continues for all D data features. At each node at level k in the tree, therefore, there is a hypothesis with k features matched.

If this IT were explored completely, there would be M^D "leaf" nodes at the bottom of the tree (i.e. these many complete interpretations) and

$$\sum_{i=0}^{D} M^i = \frac{M^{D+1} - 1}{M - 1} \doteq M^D$$

nodes in the full tree. If either M or D are of any reasonable size (e.g. larger than 5), then we can expect to have excessively large search trees.

An additional complication is that one usually wishes to include a "wildcard" feature that will match with any other feature. This is necessary because it may not always be possible to find a model feature that matches the data feature at the current level of the tree (because of fragmentation, bad segmentation, noise, unrelated features, etc.). If one were searching for a data feature to match a given model feature, there is also the possibility of occlusion. This increases the number of leaf nodes to $(M + 1)^D$.

One way to reduce the amount of searching is to 'prune whole branches of the tree', by showing that a given pairing or sequence of pairings is inconsistent. In consequence, all descendents from that node in the tree will also be inconsistent and need not be explored. The most common approach uses unary and binary pruning constraints. Unary constraints eliminate model-to-data pairings when some shared property is inconsistent. Binary constraints eliminate hypotheses when a relative property between a pair of model features is inconsistent with the same property between the corresponding pair of data features. For example, Grimson and Lozano-Perez [12] provide a set of *binary* constraints useful for three-dimensional scene analysis, based on *pairwise consistency constraints*, that compare quantities such as relative distance, orientation and direction. Similar constraints can be developed for higher-order consistency (e.g. vector triple products). Of particular importance is the local nature of the consistency tests, based on the assumption that a few simple, fast tests on partially generated hypotheses will eliminate large numbers of globally inconsistent hypotheses. Of course, it would be ideal if satisfying the set of local unary and binary tests would guarantee global consistency, but this is not always the case. (Freuder [10] considers k-consistent problems, wherein satisfying constraints over all sets of k-tuples of pairings guarantees global consistency).

The following quantities are used:

- There are M model features in the model.
- On average, $p_v M$ of these are visible in the scene (less than M by occlusion, being on the back side of the object, etc.). In 2D scenes, $p_v \doteq 1$ and, in 3D scenes, $p_v \doteq 0.5$ as about half of the features are back-facing and hence not visible.
- Of the visible model features, only p_r of these are recognizable (less than those visible because of segmentation failures, etc.) forming $C = p_r p_v M$ correct matchable data features. (If the model chosen for this scene is not present, $p_r = 0$.) Which C of the M model features are matchable is not known initially.
- There are also S spurious data features (e.g. noise features and unrecognizable visible model features). Altogether there are $D = C + S$ data features.
- The probability that a randomly chosen model feature matches with an incorrect random data feature is p_1 (correct pairings always match).

- The probability that a random pair of model features is consistent with an incorrect random pair of data features (given that the individual model-to-data pairings are consistent) is p_2.

- An acceptable set of model-to-data pairings must have at least $T = \tau p_v M$ non-wildcard correspondences ($\tau \in [0,1]$).

In the discussion that follows, the term *generated* refers to matches that are hypothesized prior to consistency testing, and *accepted* refers to matches that pass the consistency tests.

Grimson [13] analyzed the combinatorics of the standard algorithm, and showed that, <u>without wildcards</u>, the algorithm tends to accept (Proposition 5, pg 274) a single path with many pairings (i.e. the correct one), and generates (Proposition 6, pg 274) a number of nodes that is quadratic in the number of model features.

When wildcards are allowed, examination of the search process shows there are several sources of wasted effort. Now, the algorithm could accept an exponential number of correctly matchable features. One key term is 2^C, arising from the power set of the C matchable features. The complexity occurs because each matchable data feature can be either matched with the correct model feature or the wildcard. Examination of a typical search tree shows that most of the tree consists of paths containing either members of this power set or many wildcards, paired with one or two false matches. The best-first and consistent pair matching algorithms reduce this wasted exploration. Another source of wasted effort is the re-exploration of identical subtrees under each initial set of matches. The best-first and consistent pair, invocation, hierarchical and geometric matching algorithms reduce this re-exploration.

Many of the mainly wildcard paths can be eliminated by using the *termination threshold*, but this can only apply when the search is sufficiently advanced. Grimson [13] analyzed the consequences of using pruning and the termination condition and showed (Corollary 3.2, pg 367) that if $p_2 DM < 2$ then the expected number of nodes generated is bounded by:

$$\frac{MD^2}{C} < num\_generated < aT\frac{MD^2}{C}$$

where a is a small constant. Unfortunately, the $p_2 DM < 2$ condition given above does not always hold, in which case the algorithm again seems to be exponential. In the experiments described below, it only holds for the smallest test cases.

There might be some small problems with the exactness of these bounds, as:

1. In the bound $O(W(s))$ in Corollary 3.1 and $o(W(s))$ in Corollary 3.2 ([13], Page 367) $(1+\frac{\kappa^2}{M})^2$ should perhaps be $(1+\frac{\kappa^2}{M^2}) = (1+p_2)$, and a factor of p_1 seems to have been lost.

2. As p_1 and p_2 become very small, we expect only true matches to be accepted. Hence, a lower bound on the amount of search comes from having the first T data features (of D) to be true. If $\frac{M}{2}$ model features are tried on average for each of these, the lower bound is $o(W(s)) = T\frac{M}{2}$, or the the algorithm gets lucky and finds the corresponding matchable features in the first T model features, the lower bound is $o(W(s)) = \frac{T^2}{2}$. Either of these is smaller than the stated lower bound of $\frac{MD^2}{C}$.

However, the conclusion that the use of a termination condition improves performance is valid, although the published bounds on the effectiveness of that algorithm may not be correct.

3 The Algorithmic Variations

This section describes the different algorithm variations explored, starting with the consistent pair algorithm to lay the foundation for the best-first algorithm.

The algorithmic framework for most variations except the hierarchical algorithm (Section 3.11), the consistent pair algorithm (Section 3.1) and the best-first algorithm (Section 3.2) is similar to that of the standard IT algorithm, except with additional tests applied where appropriate. Pseudocode for the hierarchical IT algorithm is given in [8] and for an earlier version of the consistent pair search tree in [7]. Appendix A gives the generic C code for the best-first algorithm, including the location where statistics on generated and accepted nodes are collected (which are used in the testing reported in Section 4). For discussion purposes, the standard algorithm is also summarized here. The locations where modifications are made for the various algorithm variants are given with respect to the line numbering.

```
searchtree(model_featurelist, data_featurelist, treesofar)
{
    if no more data features, return fail

    // expand tree to next stage
    data_feature = head(data_featurelist)
    for each model_feature in model_featurelist plus wildcard
    {
        increment generated count

        // test new pairing for compatibility
        if compatible(model_feature, data_feature, treesofar)
        {
            increment accepted count
            if have enough matches, return success
            if can never get enough with this most recent
                    pairing, then skip this pairing

            // recurse to expand next level
            searchtree(model_featurelist, tail(data_featurelist),
                 append((model_feature,data_feature),treesofar))
            if success, then return success
        }
    }
    return fail
}

//
// test for compatibility of new pairing with rest of pairings:
//    - pairing satisfies pairwise constraints (compatible1)
//    - pairwise geometric consistency between filled slots
//      (compatible2)
//    - special cases for wildcards
//
boolean compatible(model_feature, data_feature, treesofar)
{
    if model_feature = wildcard then return true

    // test compatibility between model_feature and data_feature
    if not compatible1(model_feature, data_feature)
        then return false

    // check pairwise with previously filled slots of this hyp
    for each pair in treesofar
        if (model_feature(pair) != wildcard)
        {
            // test compatibility between
            // (model_feature,data_feature)
```

```
                // and each non-wildcard previously found pair
                if not compatible2(pair,model_feature, data_feature)
                    then return false
            }

        // passed all tests
        return true
    }
```

3.1 Non-wildcard Search Tree Algorithm

As many of the nodes in the standard interpretation tree algorithm arise because of the use of wildcards, an alternative search algorithm [7] explores the same search space, but it does not use a wildcard model feature to match otherwise unmatchable data features. The essence of the difference is that the search process skips over all data pairings that use a wildcard, to consider the next true data-model feature pairing. This results in a flattening of the search tree. The algorithm has two phases:

1. The set $\Omega = \{s_k\} = \{(m_{i(k)}, d_{j(k)})\}, k = 1..N$ of all pairs of features satisfying the unary pairing constraints is formed, such that if s_r is before s_s (i.e. $r < s$), then $j(r) \leq j(s)$. If $j(r) = j(s)$ then $i(r) \leq i(s)$.

2. A different search tree is explored, in which each extension of a branch is formed by appending new entries from Ω, subject to the constraints that (1) each data feature appears at most once on a path through the tree and (2) the data features are used in order (with gaps allowed).

Starting from a branch ending with pair s_λ (or nothing at the root of the tree), all pairs $s_{\lambda+1} \ldots s_N$ are possible extensions to the branch. Only extensions that satisfy the normal binary constraints are accepted. Extension stops when the termination number of matches is reached, or on branches where insufficient possibilities remain in the tail of Ω.

For example, if $\Omega = \{s_1, s_2, s_3, s_4\} = \{(m_2, d_1), (m_4, d_2), (m_1, d_2), (m_5, d_4)\}$, the tree:

X

s_1 s_2 s_3 s_4

s_2 s_3 s_4 s_4 s_4

s_4 s_4

is searched depth first following the leftmost branches first (no pruning is shown here to illustrate the shape of the tree).

The implemented algorithm has several additional work-saving ideas: (1) members of the set Ω are generated only when needed, which means that fewer unary tests need be done if the data is in a fortuitous order and (2) there is a recording of pairs s_i and s_j $(i < j)$ that pass the binary tests so that when s_i is added to an existing match, only consistent pairs s_j are considered as successors. New pairs may be generated and tested on demand after previously generated pairs have been explored.

As the second and third levels of the new search tree represent matches using several non-wildcard pairings, the binary constraints eliminate almost all false pairings quickly. The trade-off is that the branching factor of the new tree is $sizeof(\Omega)$ instead of M, but the depth of the tree for any false sets of matches is usually very shallow.

This search algorithm can produce the same set of hypotheses as the standard IT algorithm, with respect to the data features paired to non-wildcard model features. The order of generation may be different when the termination threshold is used. Pseudocode for this algorithm are given in [7].

3.2 Best-First Algorithms

Using the structure of the model-to-data pairing representation of the consistent pair approach, it is now possible to define a best-first matching algorithm. Formulation of such an algorithm is not easy when using the standard IT problem representation.

Assume that it is possible to evaluate how well sets of model features match sets of data features, and also to estimate the benefit of adding additional feature matches to an existing set of matches. This evaluation can then be used as the basis of a best-first matching algorithm that investigates hypotheses in order of the best estimated evaluation. As any real problem is likely to provide some useful heuristic ordering constraints, the potential for speeding up the matching process is large. As formulated below, the algorithm is an instance of the branch-and-bound general search technique

[17]. It is also very similar to the A^* search algorithm (e.g. [20], except that we are interested in both the cost of a path to a solution (i.e. we wish to minimize the time to finding a solution) as well as the quality of the solution. In the general framework of the A-type algorithms, the evaluation function $f() = g() + h()$ is specialised to use $g()$ to represent the quality and size of a partial match set so far and $h()$ to represent an estimate of how much an addition to the match set would enhance the evaluation of the partial match set.

The core algorithm initially evaluates all model-to-data feature pairings. These pairings are then sorted into a best-first list. Larger sets of pairings are found by adding new members of the sorted list in a best-first order as defined below.

As with the standard algorithm, exploration terminates whenever a sufficiently large set of consistent matches is found, or whenever it is impossible to extend the current set of matches to the required number.

One major computational cost is the initial exhaustive model-to-data comparison, which, in most cases, is unnecessary. It is rather unlikely that false hypotheses achieve more than 2 (or 3) matched features. Hence, any initial set of 2 false matches will reach the termination criteria before needing any model features of the remaining $M-(T-2)$. Moreover, assuming a reasonable distribution of the correct matches, it is likely that the required T correct matchable features will be in the first $\lceil \frac{T}{p_r p_v} \rceil + 3$ model features matched, and certainly in the first $M-(T-C)$ model features. Hence, we need only initially explore

$$W = max(M-(T-2), min(M-(T-C), \lceil \frac{T}{p_r p_v} \rceil + 3))$$

model features. Doing less initial exploration has 3 key benefits: 1) less work on the initial comparisons, 2) fewer matches so faster sorting of the matches and 3) a lower probability of false matches with an evaluation higher than the first correct match (so less exploration of false leads, before the well-focussed search after an initial correct match is found.) Of course, the algorithm needs to be able to generate additional model-to-data matches if they are needed.

Another improvement on the basic algorithm is to temporarily reduce the amount of search allowed when looking for an extension to a given set of matches. Assuming that there are C correct model-to-data pairs in the list of consistent pairs (of length L), then, on the average, a correct pair should be encountered periodically. Hence, if one has searched for a new extension without success for some time (i.e. we are probably on a false path that cannot be extended), it is probably more productive to start exploring other possible matches. Let:

$$F = \lceil \frac{p_1 D}{p_r p_v (1 - \tau)} \rceil$$

The method used to temporarily halt exploration of a path is to reduce the priority of successor extensions. After F potential extensions to a node are tried at level 1 and after $2F$ potential extensions are tried at level 2, the evaluation of subsequent potential extensions are penalized. This lowers their position in the priority queue. Terminating the extensions is incorrect, as there is always some probability that the current set of matches is the correct one, whereas lowering the priority ensures that the set of matches will be reconsidered eventually.

The main data structures this algorithm uses are:

- a list of consistent model-to-data pairs, initially sorted, but any entries created later are added at the end in unsorted order (when new entries are dynamically added):

  $$pair_i = (model_i, data_i, A_i)$$

  where A_i is the compatibility measure of features $model_i$ and $data_i$. The pair list is initially sorted with larger A_i values at the top.

- A priority queue of entries of the following form, sorted by the estimated evaluation of the next potential extension of the set $f()$.

  $$(S_i = \{pair_{i_1}, pair_{i_2}, \ldots pair_{i_n}\}, g(S_i), m, f(S_i \bigcup \{pair_m\}))$$

  where S_i is a set of n mutually compatible model-to-data pairs, $g(S_i)$ is the *actual* evaluation of S_i, m indicates that $pair_m$ is the next extension of S_i to be considered, and $f(S_i \bigcup \{pair_m\})$ is the *estimated* evaluation of that extension. The priority queue is sorted with larger $f()$ values at the top. The queue has a finite maximum length. Whenever a new entry is to be added that requires more than the allocated space, either it is deleted (if its $f()$ value is too low), or the bottom-most entry is deleted and the new entry inserted in the appropriate location.

The core of the matching algorithm is:

1. compare first W model features to all D data features, evaluate match and sort to form initial consistent pair list.

2. initialize the priority queue with entry $(\{\}, 0.0, 1, A_1)$

3. while no satisfactory match found, do:

   (a) pop priority queue top $(S_i, g(S_i), m, f(S_i \bigcup \{pair_m\}))$

   (b) if $consistent(S_i, pair_m)$

      i. if $size(S_i \bigcup \{pair_m\}) \geq T$, then return success.

      ii. if not rejected by early termination, generate* next descendent of successful extension:

      $$(S_i \bigcup \{pair_m\}, g(S_i \bigcup \{pair_m\}), m+1, f(S_i \bigcup \{pair_m\} \bigcup \{pair_{m+1}\}))$$

      and insert* into priority queue.

   (c) update old state: if not rejected by early termination, generate* next descendent of original popped node: $(S_i, g(S_i), m+1, f(S_i \bigcup \{pair_{m+1}\}))$ and insert* into priority queue.

The generate* function returns the next entry in the current model-to-data pair list, if there are any more. Otherwise, it may attempt to generate a new entry if all model-to-data comparisons have not been done. If all model-to-data comparisons have been done, then nothing gets generated and no new queue entry is made. The insert* function estimates an evaluation of the new node (taking account of any temporary reductions of the evaluation as described above), and then enters the new node in the appropriate priority position.

The early termination rejection tests compare the number of current matches plus the estimated remaining possible matches from the initial model-to-data pair list and the remaining potential models still to explore in the initial model-to-data pairings.

Using a finite length priority queue may result in matching failures, which might occur when the correct set of matches has a low intermediate evaluation and thus falls off the priority queue bottom. However, no failures have been observed with sufficiently long queues (with at least $2Mlog(M)$ entries), although low priority bad matches do fall off of the queue.

The algorithm needs two evaluation functions, $f()$ for the estimated new state evaluation and $g()$ for the actual state evaluation. In the algorithm comparisons given in Sections 4.2 and 4.4 below, we used these evaluation functions:

$$f(S_i \bigcup \{pair_m\}) = g(S_i) + (size(S_i) - 1) + A_m$$

with the special case:

$$f(\{pair_m\}) = A_m$$

If the last pair in S_i is $pair_x$ and $m - x \geq F$ and $size(S_i) = 1$, then the $f()$ value is reduced by 1. If $m - x \geq 2F$ and $size(S_i) = 2$, the $f()$ value is reduced by 3.

The initial A_i values are calculated from the degree to which the model feature is compatible with the data feature. It is assumed that the evaluation for model-to-data pairing i is e_i, which lies in the range (0,1]. Then:

$$A_i = 2log(e_i) + 1$$

Finally, here, the $g()$ function was set to be equal to the $f()$ function. However, in a real matching problem, $g()$ should reflect the consistency of the whole set of matches, and might be something like a scaled least-square error value.

In the simulations, the model-to-data pairing evaluations used the function

$$2^{1-\frac{x^2}{9}} - 1$$

where $x \epsilon U[-3, 3]$. In the algorithm variant compared with the other algorithms in Section 4.4, both the correct and false matches were evaluated from the same distribution (to allow comparison with the other algorithms); however, in the case of real matching problems, it is expected that the evaluation function would produce higher evaluations for correct matches. The ideal $g()$ evaluation function would take account of how well the current set of features are matched to the model as a whole, not just how well their properties match. That is, some form of geometric verification should be used, coupled with some metric on how much evidence has been found. With such a function, then the rankings should well represent the degree to which the model is recognized.

3.3 Re-ordering The Tree

In the most common formulation of the algorithm, one expands the IT one *data* feature at a time, leading to a maximal tree of size M^D. However, one might also expand the IT one *model* feature at a time, leading to a maximal tree of size D^M. In a typical scene, one usually expects $D > M > 2$, so in this case $D^M < M^D$. The difference in the number of nodes generated might be quite substantial. For example, if $M = 20$ and $D = 40$, then $M^D = 10^{52}$, whereas $D^M = 10^{32}$. These are both ridiculously large, of course, but it illustrates the point that substantial savings might be achievable by expanding the search tree in the other order.

Besides the complexity difference, the two algorithms also explore a different search space. In the extreme, the standard algorithm might pair all data features to the same model feature. This cannot occur with the re-ordered algorithm, but it might pair all model features to the same data feature. Hence, another determinant of which algorithm to use is the nature of data errors expected. If multiple data features can pair with one model feature (i.e. a feature may get fragmented), then one should use the standard algorithm. If multiple model features can pair with one data feature (i.e. a feature may be undersegmented), then one should use the re-ordered algorithm.

Both algorithms generate the same subset of combinatorial pairings when all features can be used only once, but there is also a substantial difference in those paired with the wildcard. Since both algorithms require a given number of matched features, the matched results will be similar; however, the number of spurious hypotheses maintained can be quite different.

Algorithms that mix data and model levels are possible, opportunistically expanding likely sequences of matches. However, the book-keeping required to ensure that all possible matches are explored is complex. Moreover, as the results in Section 4 do not show a marked advantage to the reordered algorithm, there is probably not much benefit to be obtained from this extension.

The reference to model and data features should be swapped in lines 6, 7 and 20/21 in the standard algorithm given above.

3.4 Unique Use of Features

The standard algorithm allows the use of all model features at each level in the search tree, irrespective of whether the model feature might have previously been matched to another data feature. This is reasonable when data features might be fragmented; however, it need not always be the case, and it might also be reasonable to allow only the first match to occur.

A variation on this algorithm is described in [19], where verification is based on the number of unique groups in a bi-partite graph. If model features are separated sufficiently with respect to sensor error, then the number of matched model features gives the same results.

A test for unique use of model features should be added after line 9 in the standard algorithm.

3.5 Visibility Subgroups

When enough non-wildcard features are matched, it is possible to estimate a 3D orientation for the model. We can then predict which other model features are visible (here assumed to be $p_v M$), and the search tree is expanded only for the visible features. This reduces both the number of generated features and the number of false matches.

A test for whether orientation can be determined is inserted before lines 20/21 in the standard algorithm and if the test succeeds, then non-visible model features are removed from the set of model features used in the recursion.

3.6 Typed Features

If we use typed features (e.g. only matching “straight” lines to “straight” lines), then we need only consider pairs where the model feature type is the same as that of the data. If there are N types each with equal likelihood, this reduces the probability of a false feature correspondence by a factor of $\frac{1}{N}$. On the other hand, the factors that determine the type can be included in the unary constraint testing. So, this extension can be seen as the same as the standard algorithm, except possibly with a different value of p_1.

3.7 Geometric Matching Algorithms

Once two (or more, depending on the type of feature and problem dimension) oriented model-to-data pairings have been formed, it is (usually) possible to estimate a pose for the model [2]. Then, the exponential portion of the search algorithm stops for that branch. From the pose estimate, it is possible to predict the image position of unmatched model features including which are back-facing and hence not visible [6], and hence not searched for further. For the other unmatched model features, direct search is possible, testing each data feature to see if its position is consistent with the predicted model feature position. The effort required to do each comparison is assumed to be comparable to that of standard algorithm doing the unary and binary tests when considering a new match, and is thus added to the complexity of the algorithm in the experiments given below.

This variation still has an exponential element, because it is not possible to guarantee that the correct solution is found early in the search. Hence, many paths containing mainly wildcards still are generated. Further, although the direct search phase requires only about $\frac{D}{2}$ searches for each unmatched visible model feature, D must be checked for each non-visible model feature (unless some visibility pruning is done — see Section 3.5). And, for each initial false set of pairings, $O(MD)$ searches must be done, which adds a considerable effort.

If spatial indexing of data features is possible, then the direct search phase need only match against data features directly indexed, instead of all features. This reduces the verification work to $O(M)$. In the experiments described below, we assume that spatial indexing is sufficiently good that, for each prediction, only 1 other incorrect data feature is considered (besides any correctly matching, if nearby). Further, in the experiments, we investigate requiring both 2 and 3 non-wildcard matches in the IT before going to the geometric matching phase.

A test for whether pose can be determined is inserted after line 7 and if so the algorithm passes to the geometric search phase.

3.8 Alignment Methods

Huttenlocher and Ullman [18] described a variation on the interpretation tree search where, after several levels of the interpretation tree were explored, a geometric pruning operation was applied. When sufficient numbers of non-wildcard model-to-data features are paired that the model position can be estimated, then this position is used to predict the position of the remaining unmatched model features. Then, using some form of image indexing, potentially matching data features for the fan-out on subsequent branches of the interpretation tree are located directly. Only these data features are used in the remaining exploration of the search tree. In this case, the interpretation tree is searched by expanding model levels, rather than data levels (as in the reordered algorithm described in Section 3.3). The intuition behind this algorithm is that there are unlikely to be many candidate features near the predicted position, so the branching factor of the remaining matches is likely to be small. In the experiments below, we assume that direct search starts after 2 non-wildcard features are matched and that spatial indexing is sufficiently good that, for each prediction, only 1 other incorrect data feature is considered (besides any correctly matching, if nearby).

A test for whether orientation can be determined is inserted before lines 20/21 in the standard algorithm and if the test succeeds, then the position of the remaining candidate data features is made for each remaining model feature.

3.9 Model Invocation Methods

The standard combinatorial matching model presumes that every possible model and data feature could be paired with each other. If some very limiting pre-classification of the data features or *model invocation* ([4], Chapter 8) occurs, then only the pre-selected model-to-data correspondences need to be considered, resulting in a greatly reduced search space. The pre-classification does not affect the number of nodes accepted (as exactly the same nodes as before are accepted), but it reduces the fan-out at each node and hence the number of nodes generated. The pre-classification process requires a comparison between each model and data feature with complexity $O(MD)$. Once in search, the search tree is the same as for the standard IT algorithm, except that at level λ, only model features (or the wildcard) known to be unary compatible with data feature d_λ are compared. No unary tests are needed as compatibility is ensured, but the binary feature compatibility tests still apply.

In the standard algorithm, line 7 is replaced by a generation that uses only compatible model features.

3.10 Ordered Search

If some way of ordering the features existed, then a more efficient algorithm could be developed. Suppose that both the model and data features could be given a linear order and valid pairings were required to follow this ordering. Then, whenever we have successfully matched a model feature, for subsequent matches we need only consider model features after this feature in the ordering [1]. This might lead to a large reduction of the search space, as the number of candidate matching features could shrink quite quickly. However, as there is always a chance of a spurious match with any data feature, some false matches are still expected, and this will limit the performance gains. In this case, early termination can occur whenever there are either insufficient model or data features remaining.

3.11 Subcomponent Hierarchies

Suppose that the $M = K^L$ model features can be decomposed into K^{L-1} primitive subcomponents each containing K features [8]. The algorithm (with special cases for incomplete branches) also works if M is not an exact power of K, but the description below is simpler if we assume it is an exact power. Then, each of the subcomponents are grouped into K^{L-2} larger models, each containing K^2 features, and so on hierarchically until we have one top-level model containing all K^L model features. Let each group at each level now define a new type of model feature representing its particular set of subcomponents. The matching algorithm described below generates hypotheses of these submodel types, and only these model types can be matched together to create hypotheses of the next larger model type. We note that it is not necessary for the features to have any natural model hierarchy — the algorithm described here assumes that any set of model features can be grouped into any arbitrary hierarchy.

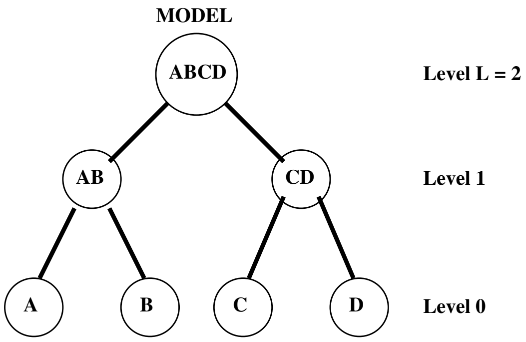


Figure 1: Simple Model Hierarchy

We describe here a binary subcomponent hierarchy matching algorithm (i.e. $K = 2$). As an example, consider the simple model hierarchy consisting of four model features A - D organized into two larger subcomponents AB and CD, which are combined into a larger model $ABCD$ (see Figure 1). We use a top-down matching strategy (e.g. look for instances of $ABCD$, which recursively looks for instances of AB and CD, etc.). Then, at level λ from the bottom of the model hierarchy, we need only compare the hypotheses that were successfully generated at level $\lambda - 1$.

The algorithm records hypotheses accepted at any given point in the model hierarchy. Then, for example, if the algorithm needs to backtrack from level $ABCD$ to find a new hypothesis for AB, it is not necessary to later re-explore previously explored portions of the matching space for CD. The algorithm recalls and retries previously verified matches, and then, if there were no successes, generates additional hypotheses starting from where the matching last stopped for this model. This provides a substantial savings over the standard IT algorithm, which re-explores the latter portion of the interpretation tree for each different initial match.

At the lowest levels of the hierarchy, many consistent hypotheses are composed mainly of wildcards. Hence, the hierarchical algorithm uses a "largest-subhypothesis-first" search algorithm (i.e. having the most matched non-wildcard data features). The algorithm determines the largest possible hypothesis size, then attempts to generate hypotheses of that size before considering smaller hypotheses. For example, if N were the currently desired hypothesis size, for a binary tree it would attempt to find two sub-hypotheses of sizes $(\frac{N}{2}, \frac{N}{2})$, then $(\frac{N}{2}+1, \frac{N}{2}-1)$, then $(\frac{N}{2}-1, \frac{N}{2}+1)$, then $(\frac{N}{2}+2, \frac{N}{2}-2)$, etc. After all possible subhypotheses of this size have been generated and if more matches still need to be considered, then the algorithm then attempts to generate hypotheses having $N-1$ matched features. Since the subhypotheses may be of different sizes (although generated in order of size), the algorithm for pairing hypotheses at this level uses a sensible ordering:

- all hypotheses that have N matched data features are generated before any hypothesis of size $N-1$,
- amongst all hypotheses with the same size, the algorithm generates the hypotheses in the order that keeps the subhypothesis sizes as similar as possible (e.g. it chooses the pair of sizes (3+2) over the pair of sizes (4+1))

At higher levels, when a new subhypothesis is needed, the algorithm may: (1) return a previously generated subhypothesis of the desired size, (2) generate a new subhypothesis that is the same size as that returned at the last call (if there was a previous call), (3) generate the largest possible new subhypothesis of a smaller size or (4) fail when no more subhypotheses are possible.

This algorithm works because, by induction on the previous level, the subhypotheses generated for the two subcomponents at the next lower level are also recursively generated in a largest-first order, and are then combined in the order that produces the largest hypotheses first.

When the recursive request for hypotheses reaches the lowest level 0, the "leaf" level of the subcomponent tree (e.g. model feature A in the above example), the algorithm matches the original data features to the original primitive model subcomponents. New pairings are tested on demand from above, so not all given model-data feature pairings need always be tested before a successful match is found. Consistency is tested using the standard unary feature matching tests. After all possible matches with data features have been attempted, then a match using the wildcard is generated. This promotes filled hypotheses over the proliferation of empty hypotheses.

When testing subhypothesis consistency at level 1 (i.e. pairings involving two model and data features), the standard algorithm's binary feature matching tests are used. At level 2 and higher, consistency is based on the same binary compatibility tests, only tested using primitive model-data feature pairings that come from different subhypotheses.

The approach generates hypotheses that do not need wildcards before those needing wildcards. Thus large consistent hypotheses will succeed before hypotheses containing a few false matches and large numbers of wildcards. This ordering limits backtracking and pursuit of dead-ends, for example, when the first match is a false one, and many subsequent matches have to be tried before it is realized that the first match must be rejected. On the other hand, the algorithm still must generate most of the same low-level hypotheses as the standard algorithm, pairing individual model and data features, so there will still be a lot of matching required.

Also important to the efficiency of the algorithm is early termination testing. Whenever the algorithm determines that a hypothesis at a given level cannot generate enough matches in the rest of the tree to satisfy the termination condition, that hypothesis is rejected and no further hypotheses for that node are generated (as they will only be the same size or smaller). The maximum hypothesis size at a given point is equal to the number of matched features to the left of the current node in the tree (can be passed down the tree as matching recurses to lower levels), plus the number matched at the current point (i.e. after either the left or the right subnodes have been generated), plus the maximum matchable nodes in the remainder of the tree to the right. This last term is either the size of the first actual hypothesis generated for the right portion of the (as generation is largest first) or the number of model features to the right of the tree. Recording the maximum sizes of hypotheses to the right of each point allows easy determination of this value.

The order of features in the hierarchy is not important to the success of the algorithm, and the algorithm does not require the model to have any natural binary decomposition. However, efficiency is improved if highly likely matches are in the leftmost nodes, thus preventing the algorithm from having to back-track through false starts.

There are special cases when M is not an exact power of K and for matches consisting of all wildcards (which becomes a wildcard at the next higher level). Pseudocode for this algorithm and an example of a matching with the model shown above are given in [8].

Matching Subcomponent Hierarchies But Using Pose Consistency

We also experimented with only testing compatibility between subcomponent positions, rather than testing binary compatibility at all levels. In this variation of the hierarchical algorithm, when a new hypothesis is tested for consistency, if the two subcomponent hypotheses have sufficient features matched that their poses have been estimated, then rather than doing a combinatorial binary test comparison between the features of the two subcomponent hypotheses, instead only the positions of the subcomponent hypotheses are checked for consistency relative to the position of their "parent" hypothesis. This reduces the number of binary tests being checked dramatically resulting in a substantial reduction in computing time. This algorithm was found to allow false matches to result, because the subcomponent testing required only the one position test to succeed, whereas the binary testing approach tested many pairs of hypotheses. The binary testing approach described in the previous subsection requires more data to be kept and tested, but allowed false complete matches to result with only a very low probability.

4 The Experiments

4.1 Method

To demonstrate the effectiveness of the different search algorithms, we use the following simulated experimental problem, based on on experiments described in Grimson [14, 15]. (Real matching problems also follow below.) The approach is designed to allow comparison of methods for which no formal complexity measure has yet been determined, and also to allow comparison of algorithms within the same complexity class. The experiments use simulated data, but Grimson showed that the model and simulation gave a reasonable characterization of real matching problems. The use of the simulated problems then allows us to compare the algorithm performance on data sets of varying sizes.

Based on the problem model given in Section 2, each model-match experiment of the standard algorithmic variations consisted of:

1. Initially determining a random selection of C of the D data features to be the solution.

2. For each generated model-to-data pairing, a correspondence that is not part of the solution and does not use a wildcard is accepted if:

   - the new correspondence is individually satisfied with probability p_1 and
   - the new correspondence is pairwise satisfied with each previously filled non-wildcard feature with probability p_2.

   Correspondences that are part of the solution or use the wildcard are accepted.

The experiments of the best-first and consistent pair algorithms are straight-forward extensions of this framework, except they obviously do not use wildcards. The geometric pruning algorithm was run for the cases when 2 and 3 matches were needed before the geometric matching phase.

For the experiments described in this paper, we used:

PARAMETER	NOMINAL	RANGE
M	40	5 to 100 by 5
S	20	0 to 100 by 5
p_1	0.1	0.05 to 0.75 by 0.05
p_2	0.01	0.001, 0.002, 0.004, 0.008, 0.01, 0.02 to 0.20 by 0.02, 0.25
τ	0.5	0.2 to 0.9 by 0.1
p_v	0.5	no variation
p_r	0.95	no variation

In each experiment described in this section, one parameter was varied over the range given above and all others were set to the nominal value. All experiments were run 200 times and the value reported is the mean value.

4.2 Comparison of Variants of the Best-First Algorithm

In the main algorithm comparison in Section 4.4, we use one variant of the best-first algorithm for comparison to the other algorithms. Here, we look at seven variations of the best-first algorithm:

1. the basic algorithm: initial model-to-data comparison over *all* model features, no limits on the fan-out from any sequence of matches, uniform distribution of a single property.

2. limit the initial model-to-data comparison to the number expected to be needed for successful termination.

3. limit the fan-out to a reasonable number to terminate search on unproductive nodes.

4. combine the limits on initial comparisons and fan-out.

5. use a property that has a gaussian (rather than uniform) distribution for correct matches (and uniform for incorrect matches).

6. use the average of three gaussian property evaluations rather than a single property evaluation, to reflect the fact that model-to-data comparison is usually not just over a single property.

7. combining the techniques of cases 2, 3 ad 6.

The graphs in Figure 2 show how the number of nodes generated and accepted varied with seven variations of the best-first algorithm. In the graphs, the curves for the different algorithms are labeled by:

Label	Algorithm
basic	Basic algorithm
ilimit	Initial comparison limits
flimit	Fan-out limits
bestu	Using both limits and a single uniform property comparison
gauss1	Single gaussian property in basic algorithm
gauss3	Average of 3 gaussian properties in basic algorithm
bestg3	Using both limits and average of 3 gaussian properties
norm	Standard Interpretation Tree (Section 2)

As can be seen in the generation results, both the fan-out (flimit) and the initial generation (ilimit) techniques make some improvement and combine to produce a

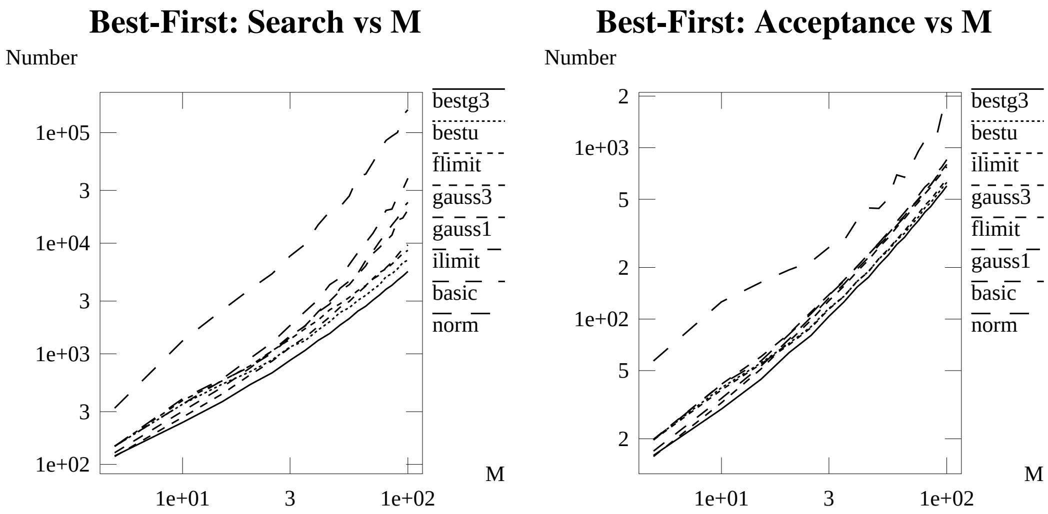


Figure 2: Best-first algorithms: Generated and Accepted Nodes versus Number of Model Features (M) with $S = 20$ $p_r = 0.95$ $p_1 = 0.1$ $p_2 = 0.01$ $p_v = 0.5$ $\tau = 0.5$. The number of nodes generated for $M = 100$ for the left graph is: bestg3:5556 bestu:7030 flimit:8619 gauss3:9633 gauss1:20001 ilimit:23434 basic:38576 norm:161236. For the right graph the number of accepted nodes at $M = 100$ are: bestg3:598 bestu:627 ilimit:657 gauss3:775 flimit:799 gauss1:799 basic:851 norm:1667.

good algorithm (bestu). Using a single gaussian property (gauss1) improves over the basic algorithm by about a factor of 2 and using the average of 3 gaussian properties (gauss3) adds another factor of 2. Using 3 gaussian properties (bestg3) also improves on the combined (bestu) algorithm; however, as this variant uses additional information, only the bestu algorithm will be compared to the other algorithms in Section 4.4. In any case, all variants greatly improve on the standard IT algorithm.

4.3 Comparison of the Variants of the Geometric Algorithm

The graphs in Figure 3 show how the number of nodes generated and accepted varied with the changed parameter for the four variants of the geometric matching algorithms (Section 3.7). In the graphs, the curves for the different algorithms are labeled by:

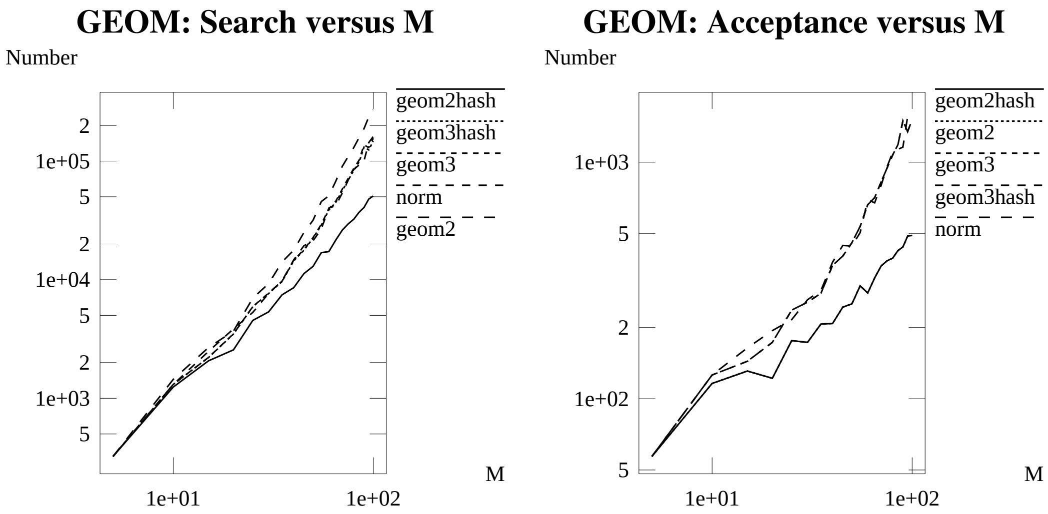


Figure 3: Geometric Algorithms: Generated and Accepted Nodes versus Number of Model Features (M) with $S = 20$ $p_r = 0.95$ $p_1 = 0.1$ $p_2 = 0.01$ $p_v = 0.5$ $\tau = 0.5$. The number of nodes generated for $M = 100$ for the left graph is: geom2hash:50773 geom3hash:147727 geom3:155508 norm:161236 geom2:271452. For the right graph the number of accepted nodes are: geom2hash:490 geom2:490 geom3hash:1516 geom3:1516 norm:1667.

Label	Algorithm
geom2	Geometric pruning with 2 starters
geom3	Geometric pruning with 3 starters
geom2hash	Geometric pruning with 2 starters and spatial indexing
geom3hash	Geometric pruning with 3 starters and spatial indexing
norm	Standard (Section 2)

The comparison shows that the hashing variations have improved the performance, particularly when starting with two matches before going to geometric verification (as compared to starting with three matches). Altogether, the geom2hash algorithm is substantially better than the standard algorithm.

The main problem with the geometric algorithms is the amount of work required to refute bad initial matches. In the geom2 algorithm, requiring only two matches before proceeding to the verification stage means that many false initial matches are generated. Using spatial indexing reduces the verification cost per match from $O(MD)$ to $O(M)$, which explains the considerable improvement in the geom2hash algorithm over the geom2 algorithm. In the case of geom3, so much more work is done in the initial interpretation tree stage that few false hypotheses are generated and so the difference between the verification work of geom3 and geom3hash is small.

The geom2hash algorithm will be compared to the other main algorithms in the next section.

4.4 Comparison of the Main Algorithm Variants

The graphs in Figures 4–13 show how the number of nodes generated and accepted varied with the changed parameter for the different algorithms. In the graphs, the curves for the different algorithms are labeled by:

Label	Algorithm
align	Alignment (Section 3.8)
bestu	Best-first (Section 3.2)
geom2hash	Geometric pruning with 2 starters and spatial indexing (Section 3.7)
hier	Subcomponent hierarchy with standard pruning (Section 3.11)
hiersubc	Subcomponent hierarchy with subcomponent pruning (Section 3.11)
invoke	Model invocation methods (Section 3.9)
consist	Consistent pair matching (Section 3.1)
norm	Standard (Section 2)
reorder	Re-ordering the tree (Section 3.3)
sort	Ordered search (Section 3.10)
uniq	Unique use of feature (Section 3.4)
vis	Visibility subgroups (Section 3.5)

As we look over the results, which explore a substantial portion of the parameter spaces likely to be encountered in visual matching problems, there is no algorithm which is always better. However, the bestu algorithm is often better or approximately equal for both generation and acceptance rates over most normal ranges of parameters. We summarize key observations about the various algorithms:

align: Distinctly worse search as M increases and not so good for small S. Initially not so good for small p_1 and p_2, but grows more slowly as p_1 and p_2 get large.

bestu: This algorithm generally has the best generation performance over most ranges of parameters. Performance declines as the termination parameter τ in-

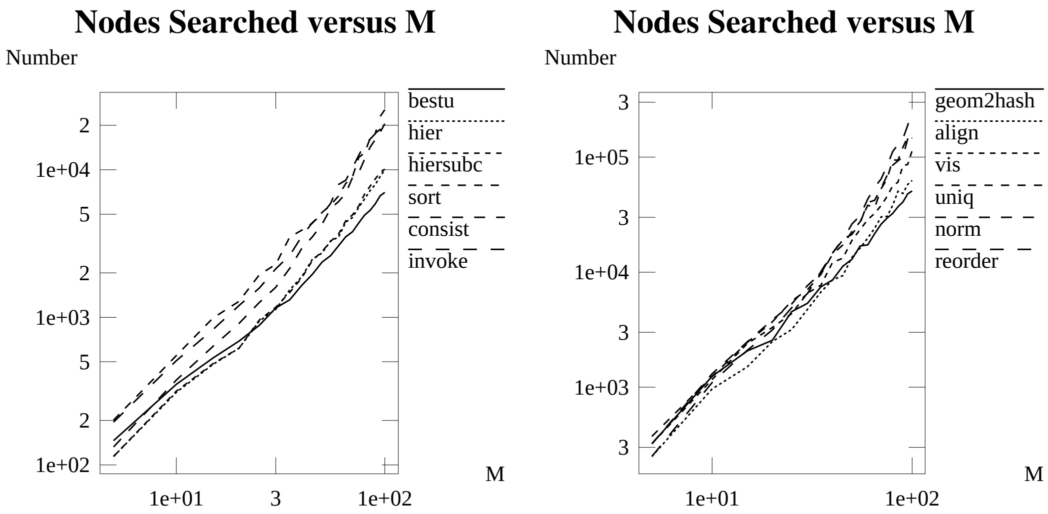


Figure 4: Generated Nodes versus Number of Model Features (M) with $S = 20$ $p_r = 0.95$ $p_1 = 0.1$ $p_2 = 0.01$ $p_v = 0.5$ $\tau = 0.5$. The number of nodes generated for $M = 100$ for the left graph is: bestu: 7030 hier:9994, hiersubc: 10101, sort:17363 consist:20487 invoke:25535. For the right graph the numbers are: geom2hash:50773 align:62568 vis:113710 uniq:146563 norm:161236 reorder:258389.

creases (in part because more initial model-to-data comparisons are performed and in part as the fan-out limit increases), but still has better performance over most of the useful range of τ. Bestu's performance degenerates a bit more quickly as p_2 grows, but other algorithms also degenerate quickly. The acceptance performance is about the same as the consistent pair algorithm, and the sort and geom2hash algorithms are somewhat better when M or τ are large.

geom2hash: Good acceptance performance when M is large (acceptance grows with a different order). Initially not so good for small p_1 and p_2, but grows slower as p_1 and p_2 get large.

hier: Generally a good algorithm with regards to both number of nodes searched and accepted. Tends to degenerate as p_2 grows, but more stable as p_1 grows. Gets worse as termination threshold increases.

hiersubc: Very similar performance as the hier algorithm. This algorithm allows incorrect matches to appear for larger p_1. (All algorithms allow some when M is small, as then only a few matches are needed to terminate and in this case there is little pruning.) This occurs when most of the correct matches

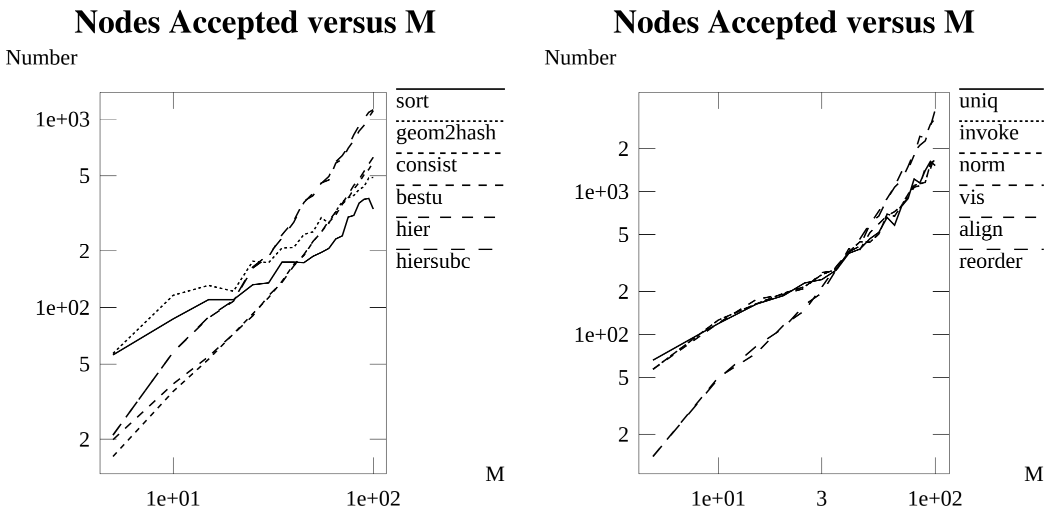


Figure 5: Accepted Nodes versus Number of Model Features (M) with $S = 20$ $p_r = 0.95$ $p_1 = 0.1$ $p_2 = 0.01$ $p_v = 0.5$ $\tau = 0.5$. The number of nodes accepted for $M = 100$ for the left graph is: sort:333 geom2hash:490 consist:590 bestu:627 hier:1107 hiersubc:1122. For the right graph the numbers are: uniq:1517 invoke:1667 norm:1667 vis:1815 align:3236 reorder:3735.

are in one half of the subcomponent hierarchy and a false match occurs on the other branch. Then, there is some probability that the false match will have a position that is compatible with the true position calculated for the other half of the subcomponent hierarchy. Note that it is always possible for the other algorithms to produce a false match, but the probability of this is low because more features are checked.

invoke: Somewhat of a middle range algorithm - with decent search as M, S and τ grow. The invoke algorithm has the same number of acceptances as the standard algorithm (as it is essentially that with preprocessing to reduce the generation phase), but has a greatly reduced generation rate.

consist: Generally a good algorithm with regards to both number of nodes searched and accepted. However, tends to degenerate as p_1 and p_2 grow.

norm: Nothing much to distinguish the standard algorithm.

reorder: Distinctly worse acceptances as M large and also not so good at small S. The reorder algorithm also does not produce a dramatic improvement, al-

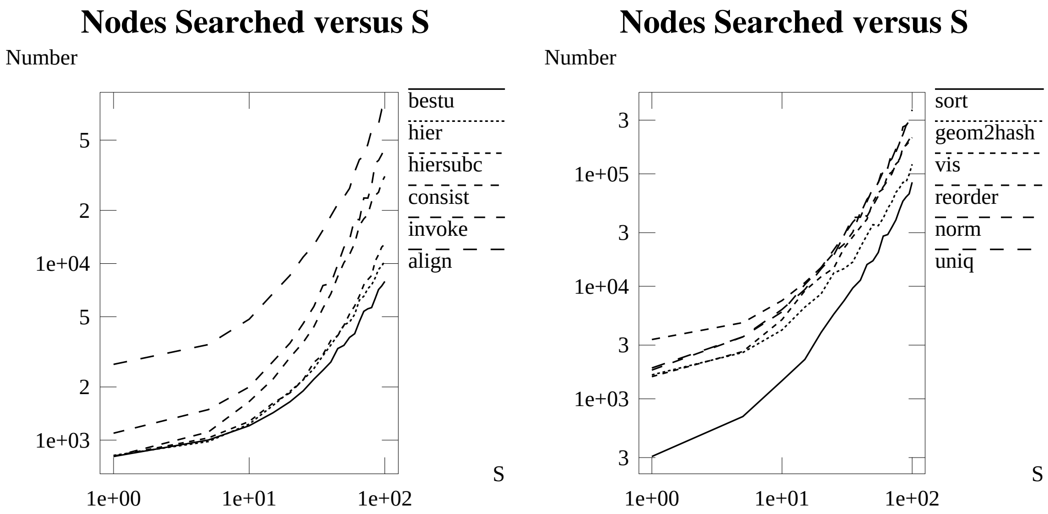


Figure 6: Generated Nodes versus Number of Spurious Features (S) with $M = 40$ $p_r = 0.95$ $p_1 = 0.1$ $p_2 = 0.01$ $p_v = 0.5$ $\tau = 0.5$. The number of nodes generated for $S = 100$ for the left graph is: bestu:7919 hier:10323 hiersubc:12875 consist:31176 invoke:44967 align:72047. For the right graph the numbers are: sort:83996 geom2hash:121459 vis:209770 reorder:221009 norm:320274 uniq:371581.

though it does have somewhat better performance as the number of spurious data features increases.

sort: Good search and acceptance as M and p_1 grow. Appears to have a different order of growth. Degenerates as S increases.

uniq: Nothing much to distinguish this algorithm, although usually better than the standard algorithm.

vis: Nothing much to distinguish this algorithm, although better than the standard algorithm.

Generally, the standard IT, the unique (Section 3.4), visibility (Section 3.5) and geometric (Section 3.7) algorithms are significantly worse than the best-first (Section 3.2), the hierarchical (Section 3.11), the invoke (Section 3.9) and the consistent pair (Section 3.1) algorithms. The real comparison is between these last four algorithms, and the choice depends on the problem parameters. The ordered search algorithm (Section 3.10) often has good performance, but makes an additional assumption about the data properties. This assumption does not give a dramatic improvement in performance, but it is a marked improvement over the standard algorithm. The align-

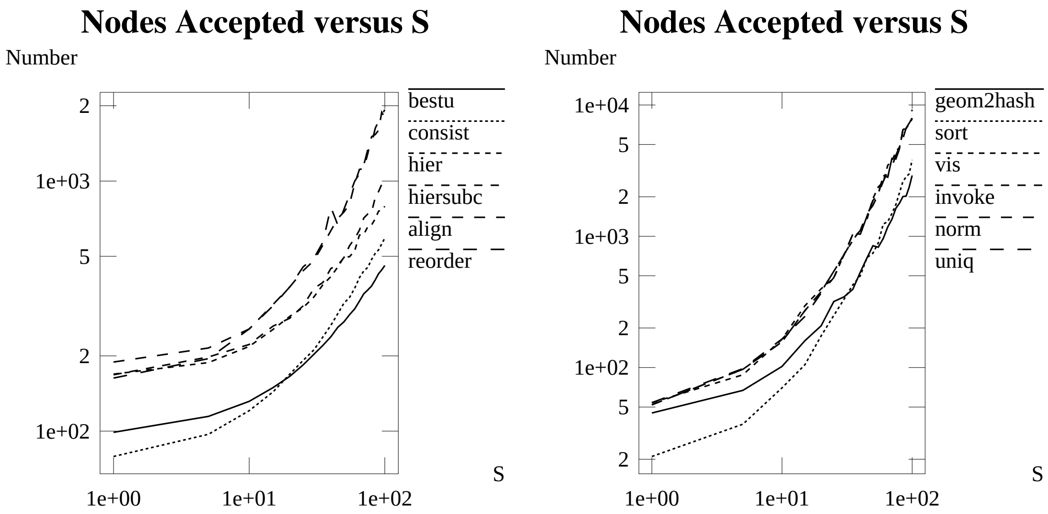


Figure 7: Accepted Nodes versus Number of Spurious Features (S) with $M = 40$ $p_r = 0.95$ $p_1 = 0.1$ $p_2 = 0.01$ $p_v = 0.5$ $\tau = 0.5$. The number of nodes accepted for $S = 100$ for the left graph is: bestu:459 consist:591 hier:792 hiersubc:979 align:1796 reorder:1929. For the right graph the numbers are: geom2hash:2904 sort:3816 vis:7841 invoke:7939 norm:7939 uniq:9211.

ment algorithm uses the reordered search algorithm, and it does perform much better than the reorder algorithm, but neither have particularly good performance. The geom2hash algorithm requires a large amount of work to refute bad initial matches.

Broadly speaking, the best-first algorithm is best or equal when a match is possible. The consistent pair algorithm is not bad for most problems, but its performance deteriorates when p_1 or p_2 is large (this increases the number of possible matches to consider at each stage). For acceptances, the best-first and consistent pair algorithm are usually the clear best, as they do not allow proliferating wildcard hypotheses. The invoke algorithm is also a possibility as it has a similar amount of search, but it has a much worse acceptance rate than the consistent pair algorithm.

One might also consider how the various algorithms perform when there is no instance of the object in the scene. In this case, it is unlikely that the early success conditions would occur, and thus almost all of the search space would have to be explored. Figure 14 shows the number of nodes generated and accepted in this case for the algorithms that showed the best results in the above experiments. When there is no true match possible, the best-first, hierarchical, invoke and consistent pair algorithms have good search performance (although both are much worse than when

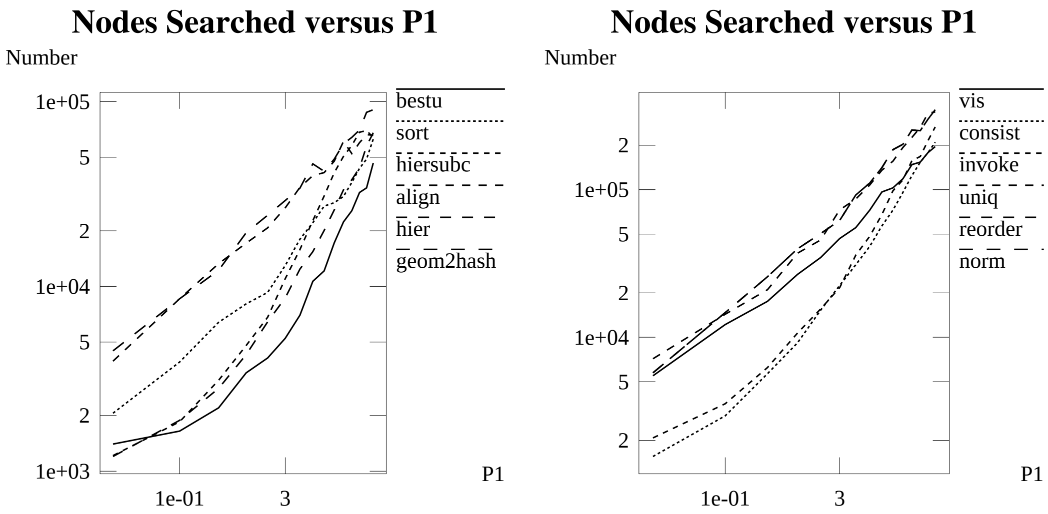


Figure 8: Generated Nodes by Unary Match Probability (p_1) with $M = 40$ $S = 20$ $p_r = 0.95$ $p_2 = 0.01$ $p_v = 0.5$ $\tau = 0.5$. The number of nodes generated for $p_1 = 0.75$ in the left graph is: bestu:46563 sort:62724 hiersubc:63921 align:65886 hier:67819 geom2hash:90416. For the right graph the numbers are: vis:195772 consist:209837 invoke:266019 uniq:336562 reorder: 348414 norm:348414.

a match is possible). The best-first, consistent pair and hierarchical algorithms have much better acceptance rates. Grimson ([13], page 389) shows that the standard algorithm is also much worse when no match is possible. Here, the results show 3-27 times more search is required in all cases.

The results displayed above showed the mean results of the various algorithms. Figure 15 shows the minimum and maximum number of nodes generated for the best-first algorithm compared to the standard algorithm. As seen here, while the mean results show the new algorithm has much better performance, and it's worst performance is better than the mean performance of the standard algorithm, it is always possible for the standard algorithm to have better results on a particular data set (i.e. if the search gets lucky with data and model feature orders).

Overall, given the problem as formulated here, only the best-first, hierarchical and consistent pair algorithms seem like real alternatives to the standard algorithm, and these algorithms give a factor of about 3–77 improvement (in search) over the standard algorithm. The choice between these should be based on the relative costs of the generation and acceptance processing. Further, the cost of real implementations needs to be considered, as the number of actual binary property tests is a key factor

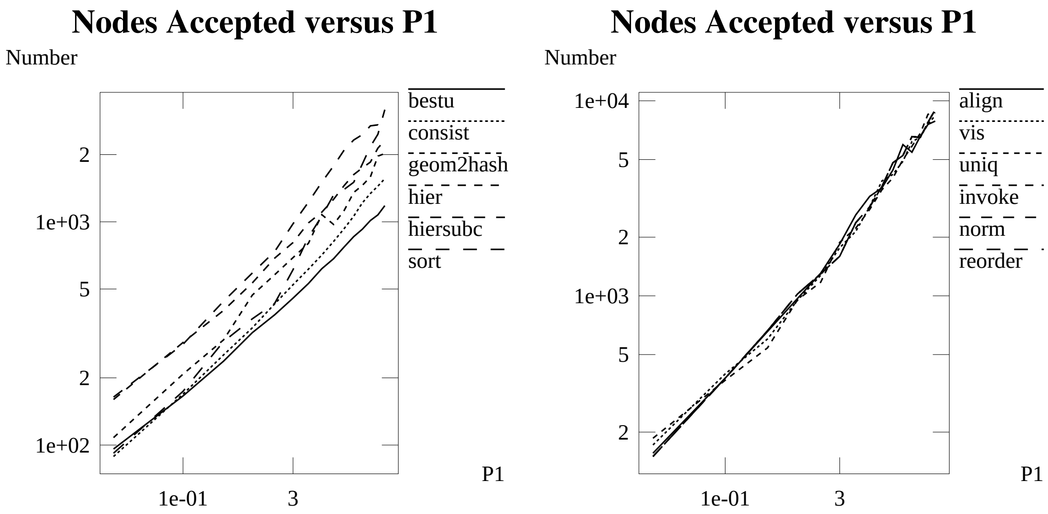
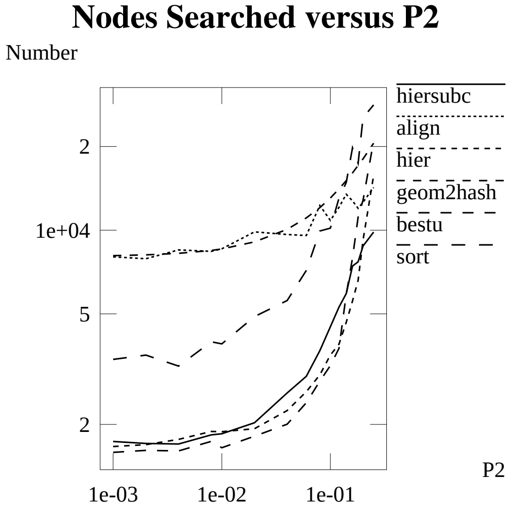


Figure 9: Accepted Nodes by Unary Match Probability (p_1) with $M = 40$ $S = 20$ $p_r = 0.95$ $p_2 = 0.01$ $p_v = 0.5$ $\tau = 0.5$. The number of nodes accepted for $p_1 = 0.75$ in the left graph is: bestu: 1180 consist:1560 geom2hash:2021 hier:2313 hiersubc:2537 sort:3177. For the right graph the numbers are: align:7868 vis:8414 uniq:8702 invoke:9008 norm:9008 reorder:9008.

in the actual run-time costs.

Even with very large problems (e.g. involving $M = 100$ model features to match), the number of potential matches generated (7030 for the best-first algorithm, 9994 for the hierarchical algorithm and 20487 for the consistent pair algorithm, as compared to 161236 for the standard algorithm) and number of acceptances (590 for the consistent pair algorithm, 627 for the best-first algorithm and 1107 for the hierarchical algorithm, as compared to 1667 for the standard algorithm) are low for the new algorithms. Both factors are important, because, depending on the particular matching algorithm, the savings achieved depend on relative costs of each action (e.g. the pairwise consistency checking costs may be high relative to final verification costs).

It is possible that the relative speed difference of the matching algorithms might overcome this reduction in search complexity, because the alternative algorithms are more complex than the standard algorithm. However, again using the simulated data $M = 100$ case from Figure 4, matching requires 0.28 seconds for the best-first algorithm, 2.1 seconds for the consistent pair algorithm and 0.93 seconds for the hierarchical algorithm, as compared to 5.4 seconds for the standard algorithm (on a SparcStation 1+, code in C++). Hence, the speed of the best-first, consistent pair and



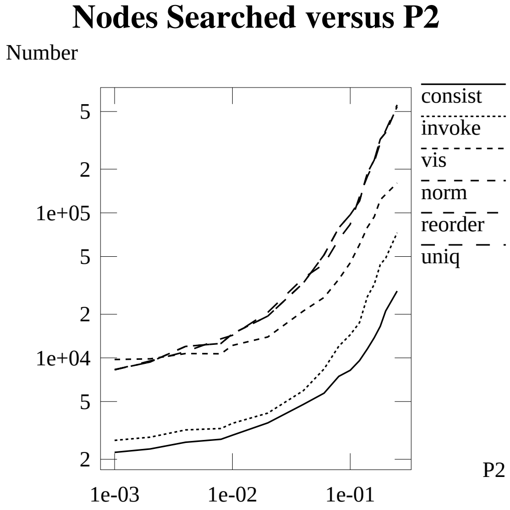


Figure 10: Generated Nodes by Binary Match Probability (p_2) with $M = 40$ $S = 20$ $p_r = 0.95$ $p_1 = 0.1$ $p_v = 0.5$ $\tau = 0.5$. The number of nodes generated for $p_2 = 0.25$ in the left graph is: hiersubc:9844 align:14236 hier:15742 geom2hash:20559 bestu:20948 sort:28234. For the right graph the numbers are: consist:28887 invoke:72920 vis:160584 norm:540186 reorder:540186 uniq:555177.

hierarchical matching algorithms are also significantly better, in practice, than the standard algorithm. Note that these times were obtained only from the simulation, which used a random number to test the feature compatibility. When in use, real, time-consuming tests will be applied, in which case the bare-bones algorithm time may be small compared to the testing time, in which case the complexity results would be relevant (i.e. if the execution time is proportional to the number of testings, which is equal to the search complexity).

5 Real Matching Examples

To assess the performance on real data, the best-first, hierarchical and consistent pair algorithms were compared on the edges shown extracted in Figure 16 (part a). Because the algorithms are sensitive to data feature order, the algorithms were run 100 times with the model and data features permuted randomly. The effective probabilities in this scene were $p_1 = .235$ and $p_2 = 0.017$ and the number of features were $M = 13$ and $D = 129$. Seven of 13 model edges match true data edges in the test scene using the given tolerances. The average time taken for the matching

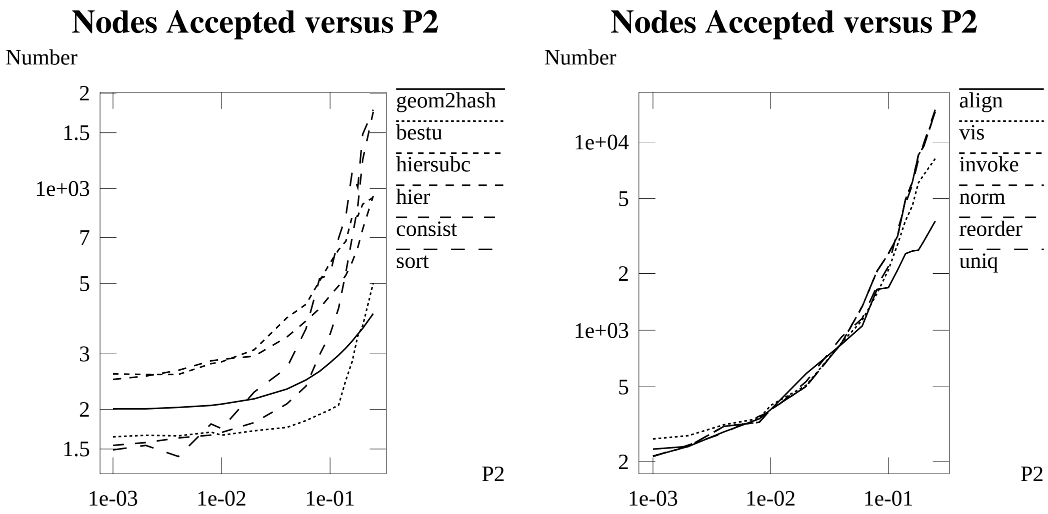


Figure 11: Accepted Nodes by Unary Match Probability (p_1) with $M = 40$ $S = 20$ $p_r = 0.95$ $p_1 = 0.1$ $p_v = 0.5$ $\tau = 0.5$. The number of nodes accepted for $p_2 = 0.25$ in the left graph is: geom2hash:402 bestu:502 hiersubc:937 hier:947 consist:1746 sort:1774. For the right graph the numbers are: align:3796 vis:8157 invoke:14490 norm:14490 reorder:14490 uniq:14863.

algorithms on a SparcStation 1+ was 0.56 sec for the best-first algorithm, 0.96 sec. for the consistent pair algorithm, 1.69 seconds for the hierarchical algorithm and 5.88 sec. for the standard algorithm. The mean number of nodes generated and accepted was 6048 and 28 for the best-first algorithm, 52686 and 1632 for the hierarchical algorithm, 64412 and 845 for the consistent pair algorithm and 544171 and 39711 for the standard algorithm. On another test scene containing 10 instances of one of these parts (Figure 16 part (b)), the average times required for a match was best-first 2.9 sec, consistent pair 20.4 sec., hierarchical 21.4 sec. and standard 419 sec. The effective probabilities in this scene were $p_1 = .288$ and $p_2 = 0.011$ and the number of features were $M = 28$ and $D = 191$. Interestingly, while the multiple instances of the part were simultaneously examined by the hierarchical algorithm, its performance was comparable to the consistent pair. The extra memory costs of recording the successful submatches in the hierarchical algorithm was about 1M bytes and about 0.5M bytes for the best-first algorithm. The hierarchical algorithm's longer average running time was a consequence of having to consider more (expensive) binary tests before hypothesis rejection, even though it generated fewer tests.

The much improved performance of the best-first algorithm as compared to the

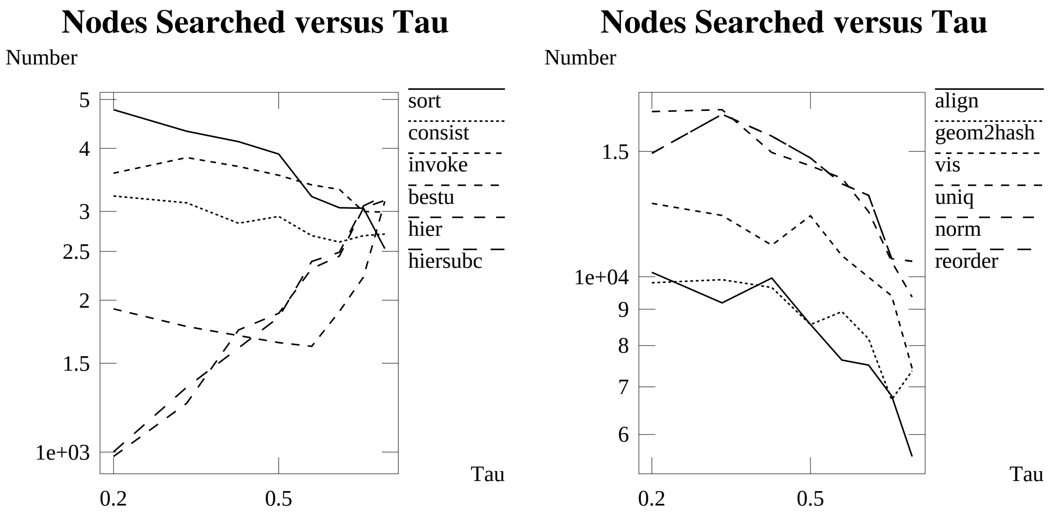


Figure 12: Generated Nodes by Acceptance Threshold (τ) with $M = 40$ $S = 20$ $p_r = 0.95$ $p_1 = 0.1$ $p_2 = 0.01$ $p_v = 0.5$. The number of nodes generated for $\tau = 0.90$ in the left graph is: sort:2532 consist:2704 invoke:2990 bestu:3139 hier:3154 hiersubc:3224. For the right graph the numbers are: align:5588 geom2hash:7373 vis:7431 uniq:9364 norm:10512 reorder:10512.

consistent-pair and hierarchical algorithms is due to the use of a good ranking function (here ranking consistent pairs by the product of average length and one minus the relative difference in the paired edge lengths). This allowed many good pairs to appear earlier in the initial priority queue than would have been the case with the uniformly distributed evaluation that occurred in the simulation experiments.

One needs to consider also the impact that these algorithm improvements make in the context of the full recognition process. Timings of the other stages of processing are: Canny edge detector: 14.3 sec, connectivity and tracking: 2.1 sec, segmentation: 1.2 sec, merging/description: 1.9 sec and pose estimation and verification: 0.7 sec. Hence, using the improved algorithms reduces the complete time from 26 seconds to 20 seconds in the first case and from 511 seconds to 23 seconds in the latter case.

Because the best-first algorithm sorts its initial consistent pairs into a best-first order, and because all model-to-data pairs were initially evaluated, each run of the real matching example above produced nearly identical results (varying only where two pairings had identical evaluations). It may be the case that we were lucky with the data set. To assess this question, the algorithms were also tested on a stereo matching problem, because of the different nature of the problem. In this case, the

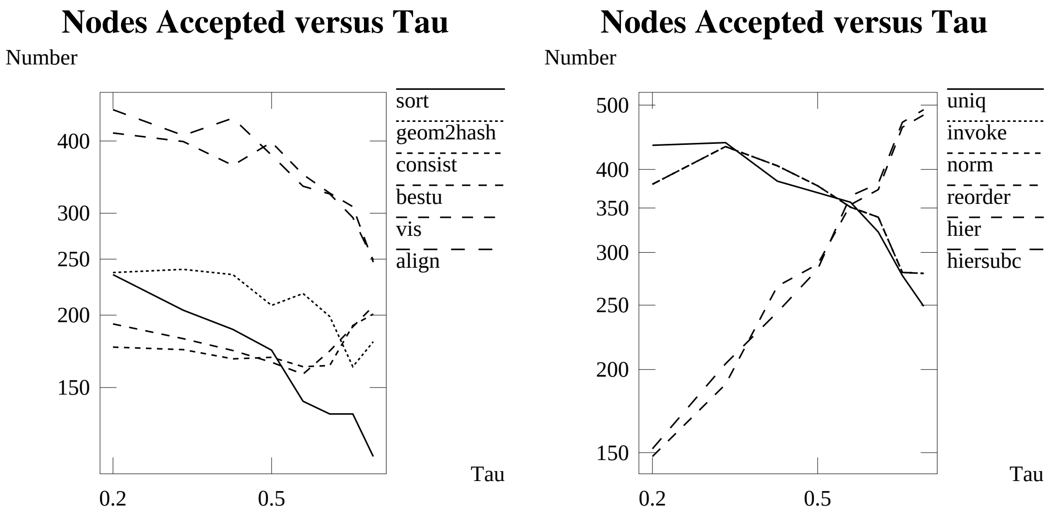


Figure 13: Accepted Nodes by Acceptance Threshold (τ) with $M = 40$ $S = 20$ $p_r = 0.95$ $p_1 = 0.1$ $p_2 = 0.01$ $p_v = 0.5$. The number of nodes accepted for $\tau = 0.90$ in the left graph is: sort:114 geom2hash:180 consist:200 bestu:207 vis:247 align:249. For the right graph the numbers are: uniq:249 invoke:279 norm:279 reorder:279 hier:483 hiersubc:492.

transformation between the two images (here treated as model and data images) is affine rather than rigid. Hence, the problem produces a number of subsets of edges that are within the unary and binary tolerances. Unfortunately, because of the perspective distortion, not all edges match simultaneously within tolerances, and so the matching algorithms can thrash a while with a subset that can almost satisfy the termination criteria before finally rejecting it and proceeding to a subset that successfully terminates. Here, the number of left image edges was $M = 82$ and the number of right image edges was $D = 98$. The number of matches required was 12 and the average time (over 100 permutations of the edges) to compute this match on a SparcStation 1+ was 0.74 sec. for the best-first algorithm, 0.99 sec. for the consistent pair algorithm, 1.29 for the hierarchical algorithm and 6.52 secs. for the standard algorithm. Figure 17 shows the left scene with the edges being matched overlaid. This is a particularly difficult test data set, because there are many subsets of the edges that almost satisfy the termination threshold, so there is a lot of thrashing with incorrect initial solutions. This problem is observed clearly in another stereo dataset from this scene with $M = 155$ and $D = 174$, where the best-first algorithm took 8.7 sec to find a match, but the consistent pair algorithm did not even after an hour of

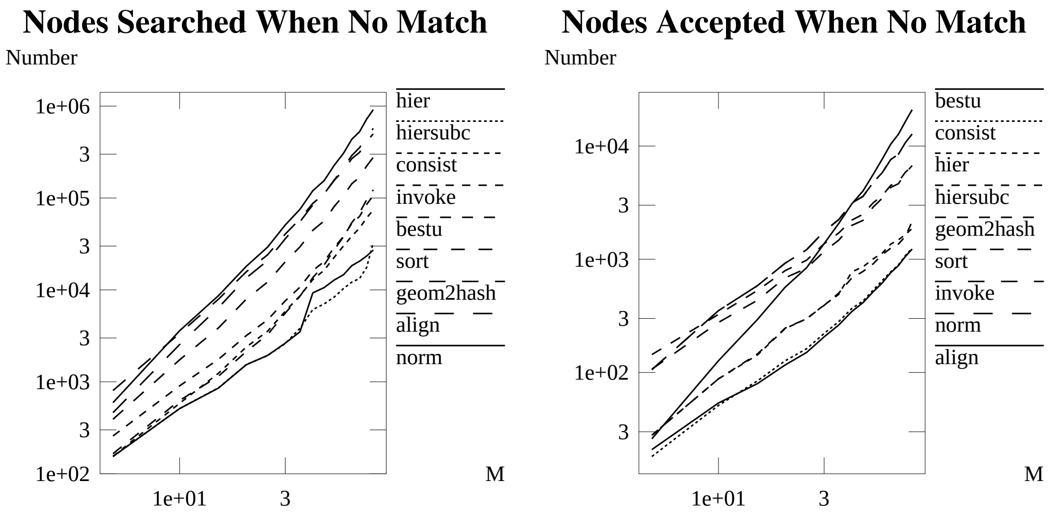


Figure 14: Generated and Accepted Nodes versus Number of Model Features (M) When No Instance of the Model is Present with $S = 20$ $p_r = 0.95$ $p_1 = 0.1$ $p_2 = 0.01$ $p_v = 0.5$ $\tau = 0.5$. The number of nodes generated for $M = 75$ is: hier: 27135 hiersubc:32741 consist:73884 invoke:106384 bestu:122586 sort:276427 geom2hash:498951 align:571904 norm:913044. The number of nodes accepted for $M = 75$ is: bestu:1230 consist:1251 hier:1872 hiersubc:2162 geom2hash:6639 sort:6725 invoke:12753 norm:12753 align:20936.

Sparc10 time (and the other algorithms were not even tried).

One observation from the real matching examples is that the matching performance is greatly dependent on the order of the features to be matched (as was suggested from the minimum, mean and maximum generated matches reported in Figure 15). A second observation is that there is a fundamental problem with match sequences that start with a bad, but nearly plausible initial match. This causes the matching algorithms to exhaustively explore an unproductive region of the search tree before proceeding onto a more productive region. The fan-out limits in the best-first algorithm help to overcome or avoid this problem, but in general it is still a problem with symbolic matching algorithms.

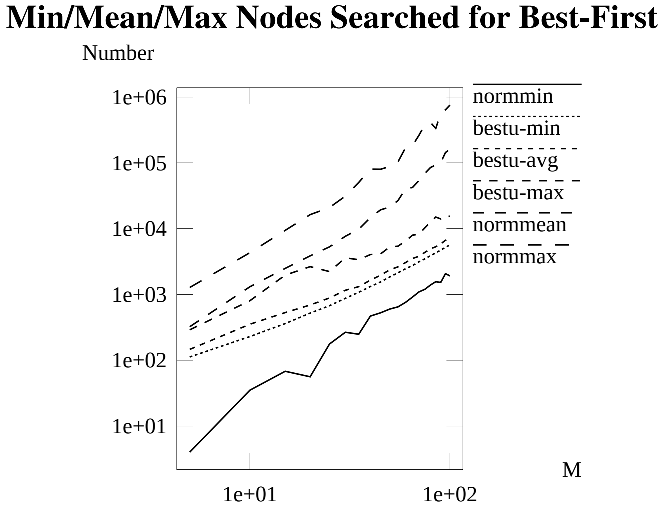


Figure 15: Minimum, Mean and Maximum Generated Nodes versus Number of Model Features (M) for the Best-first and Standard algorithms.

6 Computational Complexity of the Hybrid Best-First Matching Algorithm

Grimson [13] has mainly concentrated on estimating upper and lower bounds for the standard algorithm. As seen in the results from Section 4, the hybrid best-first search algorithm looks very promising. Hence, we give here a complexity analysis for that algorithm, except that we derive the *mean* performance of the algorithm.

Theorem 1 (Mean Complexity of Hybrid Best-First Algorithm) *Assume the problem definitions from Section 2 and also that M and D are very large, so that the effect of matching a few features does not significantly affect the statistics of the pool of matchable features. Assume that the correct model-to-data pairings are uniformly distributed amongst the false pairings. Also assume that no false hypotheses containing 3 or more pairings survive the pruning tests (i.e. $p_2F < 1$).*

Then, let:

$\hat{M} \quad = max(M-(T-2), min(M-(C-T), 3+\lceil \frac{T}{p_r p_v} \rceil))$

(number of model features expected to be needed to terminate match).

$\hat{C} \quad = \lceil \frac{\hat{M}C}{M} \rceil$

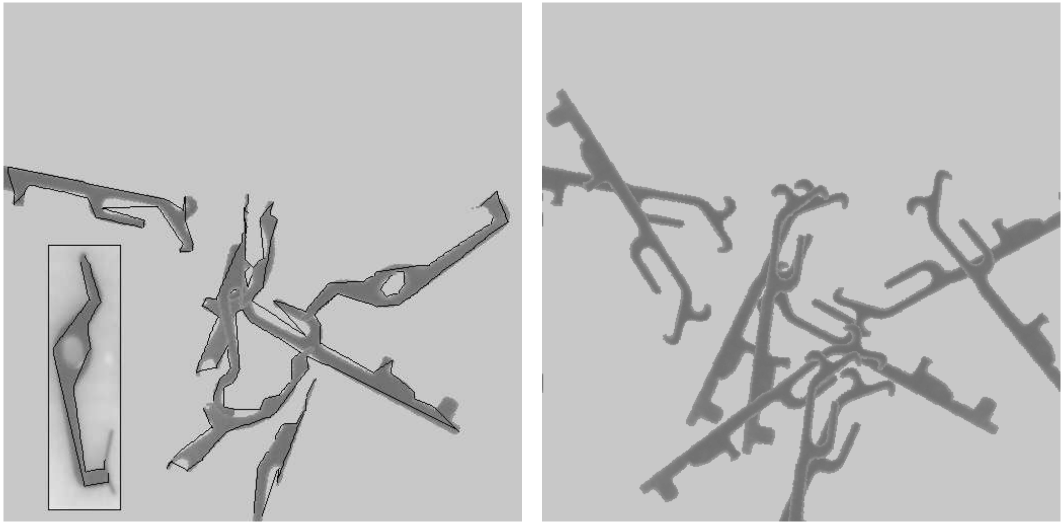


Figure 16: (a) The first example scene with extracted edges highlighted and model used shown in the box (b) The second example scene with multiple instances of the part.

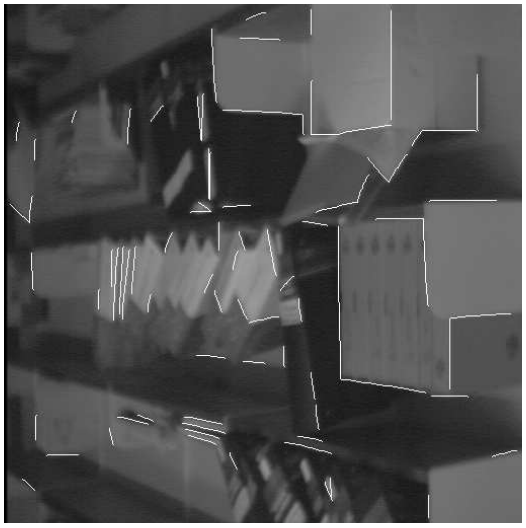


Figure 17: Stereo Scene with Edges Being Matched Highlighted

(expected number of matchable features to be observed before termination.)

$$\begin{aligned}
\hat{F} &= p_1(\hat{M}D - \hat{C}) \\
&\textit{(expected number of false pairings passing unary tests)} \\
\delta &= \lceil \tfrac{p_1 D}{p_r p_v (1-\tau)} \rceil \\
&\textit{(average number of false matches per correct match)} \\
\lambda &= \tfrac{\hat{F}}{\hat{F}+\hat{C}} \\
&\textit{(probability of next pairing being false.)} \\
\mu &= \lambda^{\delta} \\
&\textit{(probability of not getting a correct pairing in } \delta \textit{ tries.)} \\
\rho &= \tfrac{\lambda}{1-\lambda} \\
&\textit{(average number of false pairings before first true pairing.)}
\end{aligned}$$

The expected amount of search (i.e. the number of nodes generated) is approximately:

$$GEN = g_0 + g_f + g_{fx} + g_{fxx} + g_t + g_{tf} + g_{tfx} + g_{tt} + g_{ttx}$$

where each of the g_X terms refers to the number of potential matches generated along each path in Figure 18, and has the value:

$$\begin{aligned}
g_0 &= \hat{M}D \\
g_f &= \rho + \tfrac{\mu}{1-\mu}(\delta + \rho) \\
g_{fx} &= \delta a_f \\
g_{fxx} &= 2\delta a_{fx} \\
g_t &= \tfrac{1}{1-\mu} \\
g_{tf} &= \rho a_t \\
g_{tfx} &= 2\delta a_{tf} \\
g_{tt} &= 1 \\
g_{ttx} &= (T-2) * (\rho + 1)
\end{aligned}$$

As the dominant terms are g_0, g_{fx}, g_{fxx} and g_{ttx}, which are of order M^2, M^2, M^3 and M^2, the overall complexity is of order $O(M^3)$ for large M. However, for many matching problems, the g_0 dominates, which is $O(M^2)$.

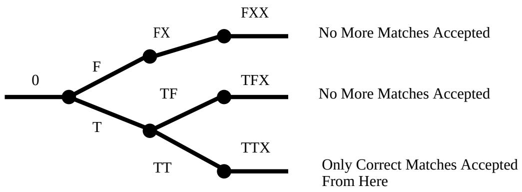


Figure 18: Prototypical Search Tree for Best-First Matching

The expected number of hypotheses accepted is approximately:

$$ACC = a_0 + a_f + a_{fx} + a_{fxx} + a_t + a_{tf} + a_{tfx} + a_{tt} + a_{ttx}$$

where each of the a_X terms refers to the number of matches accepted along each path in Figure 18, and has the value:

$$\begin{aligned}
a_0 &= \hat{C} + \hat{F} \\
a_f &= g_f \\
a_{fx} &= p_2 g_{fx} \\
a_{fxx} &= (p_2)^2 g_{fxx} \\
a_t &= g_t \\
a_{tf} &= p_2 g_{tf} \\
a_{tfx} &= (p_2)^2 g_{tfx} \\
a_{tt} &= g_{tt} \\
a_{ttx} &= (T-2)
\end{aligned}$$

As the dominant terms are a_0, a_{fx} and a_{fxx}, which are of order M^2, M^2 and M^3, the overall complexity is of order $O(M^3)$ for large M. However, for many matching problems, the a_0 term dominates, which is approximately $p_1 M^2$.

Proof:

Figure 18 shows the three prototypical types of hypothesis subtrees that can develop. The top subtree shows an initial false pairing (of which many are possible), followed by another true or false pairing. This terminates after the second attempted match (by the assumption that the probability of getting a match at this point is negligible). The second path is a true pairing followed by a false pairing, which also terminates after next pairing. The third is a sequence of true matches which is extended until the termination threshold is achieved.

The proof considers the mean case, so all occurrences are "on the average". The term "matching pairs" refers to model-data pairs that successfully match, even if false.

The g_0 term is determined by the initial pre-pass, which pairs a given number of model features to each data feature. The number of model features is the larger of { the number of model features to be encountered along the top false search path before early termination occurs, the number of model features needed to be checked before the number of required matches is reached}. The a_0 term is the percentage of the g_0 terms that match successfully (i.e. p_1 of the false matches and all true matches).

On the average, ρ false matching pairs are initially encountered along path F before the first true matching pair is encountered. Then, as occasionally the first true pair cannot be extended to a second true pair before the fan-out limit is reached (requiring testing δ pairs), further returns to exploring the F path for an additional ρ pairings occur. On the average $\frac{\mu}{1-\mu}$ of these occur.

On the F path, compatibility testing is already done in the unary comparisons, so all g_f pairs are accepted. For each of these, all subsequent pairs are explored until the fan-out limit is reached, in an attempt to form the second level FX match. This requires exploring δ hypotheses. (Hypotheses to be explored after the first δ could be encountered later at lower priority, but they are ignored here. From experience, there are not many of these.) The ones that pass the binary test (a_{fx}) are matched with a further 2δ to try to extend the match.

Once the initial F paths are explored, the T path is then selected. Then, approximately ρ false matching pairs are encountered before the second true matching pair is encountered. Of these, p_2 are accepted, and these are compared against 2δ matching pairs before this path is terminated unsuccessfully.

Finally, on the TT path, approximately $\rho + 1$ matching pairs are compared before accepting each new true matching pair, and we need $T - 2$ additional matches to achieve successful termination.

end proof

The complexity formulas do not take account of the limitation of using each data feature only once in a consistent hypothesis, which will reduce the number of matches attempted and achieved. Neither do they account for matches that have had their priority reduced (after the fan-out limit is reached) being later encountered as their priority causes them to reach the top of the queue. This effect will increase the number of matches attempted and achieved. The proof also assumed that no false hypotheses with three pairs would be accepted, which is clearly false, so a real matching will have a few extra matches of this type. Thus, the number of generated and accepted hypotheses resulting from the actual matching algorithm will be slightly different than those given in the bounds above. Further, the proof was essentially based on a removal with replacement algorithm (i.e. the probability of getting either a true or false match did not change as matches were made), so again the real algorithm will be more efficient, particularly when the number of model and data features is small.

To test the complexity bounds, we ran the simulated matching according to the procedure given in Section 4. Figure 19 part (a) compares the number of nodes actually searched by the hybrid best-first algorithm in the simulation experiments with the number estimated by GEN. Part (b) compares the number of nodes actually accepted with the number estimated by ACC. We can see that there are small differences between the predicted and actual number of nodes generated and accepted, but the mean complexity results above characterize the performance well.

7 Discussion and Conclusions

From the experiments, it is obvious that the best-first, consistent pair and hierarchical algorithms produce better performance than the more straightforward variations of the standard matching algorithm. Even for a small number of model features (e.g. $M = 10$), the amount of work of the worst algorithm is about 3 times that of the best. It is also the case that the choice of which algorithm to use depends on the nature and parameters of the problem, although the best-first algorithm seems to be generally better. There are also situations where the performance of an algorithm might suffer dramatically because of the nature of the problem. One example of this is with the hierarchical algorithm when used with problems containing parts with symmetry or

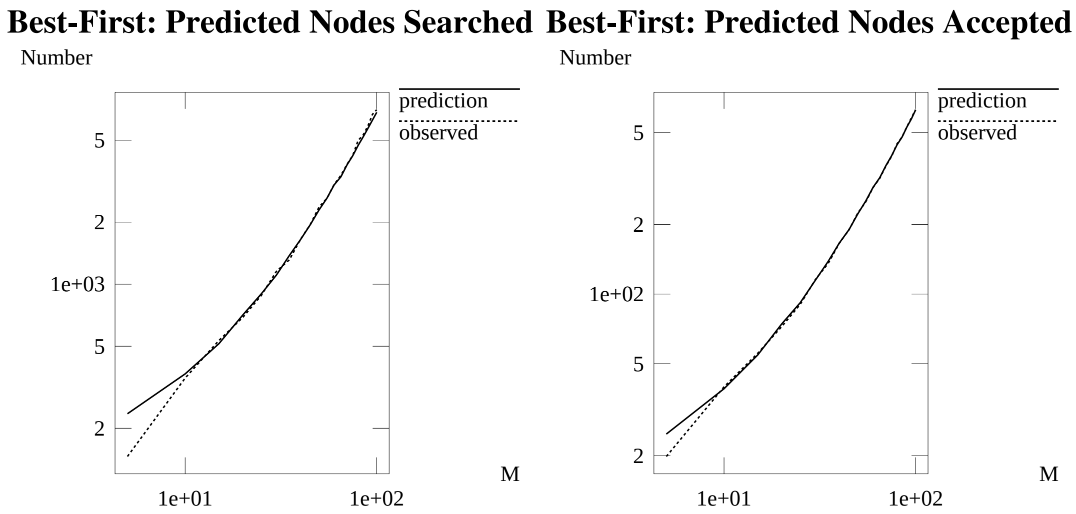


Figure 19: Part (a): Predicted and Observed Generated Nodes versus Number of Model Features (M) for the Hybrid Best-First algorithm with $S = 20$ $p_r = 0.95$ $p_1 = 0.1$ $p_2 = 0.01$ $p_v = 0.5$ $\tau = 0.5$. Part (b): Predicted and Observed Accepted Nodes versus Number of Model Features (M).

duplicated instances of the matched parts (because the bottom-up approach of the hierarchical algorithm will propagate upward multiple instances of the recognized features until finally a single complete match is achieved at the top level).

It is possible that the performance in any particular problem could be improved further by a combination of techniques. For example, once the best-first algorithm has matched 2-3 features, a visibility pruning stage could be applied, removing pairs involving model features that cannot possibly be seen. The uniqueness constraint could also be applied. In the case of the best-first algorithm, the amount of search can be reduced to almost nothing beside the initial model-to-data feature comparison, if a good evaluation function can be found. The simulation results given above showed the best-first algorithm could achieve better results if it knew nothing special about the data and if the correct model-to-data feature pairs were distributed uniformly amongst the false model-to-data pairs. If an evaluation function can be found that is good at selecting likely model-to-data pairs, then the algorithm could have much better performance. If the evaluation function were perfect, then only the correct matches would be selected first. However, the cost of applying such a function needs to be considered. Here, we assume that the initial evaluation function must be cheap to apply.

Another form of reduction can be achieved by pre-expanding the search tree [11] according to what model features can still be matched after particular model features have already been matched. This is similar to the visibility algorithm described in Section 3.5, in that each match further constrains the viewer's position, and hence the features that remain at each level are still potentially visible from that position.

However, for all of these algorithms, a comparison between nearly all model and data features is required, which can be the source of much of the real work. As any model feature might be an explanation for any data feature, it is hard to avoid this complexity, which results in MD initial comparisons and roughly p_1MD false acceptances. This effectively provides a lower bound on the amount of work required. After that, a reduced search space needs to be considered, but the initial effort is substantial. For example, except where the problem parameters are extreme (e.g. where p_2 is large), in the best-first algorithm most (i.e. approximately 70%) of the search complexity is in the initial model-to-data comparison.

There does not seem to be much possibility of reducing this amount of effort, unless some additional aspect of the particular problem can be exploited. For example, one might only consider the largest features first, and only perform additional comparisons as more features are needed. Some sort of property hashing scheme might be able to link directly from the properties of a data feature to compatible model features. This substitutes increased memory usage and offline processing for on-line processing, but could improve the performance by another factor of 2-3.

The other main sources of unproductive matches in the best-first algorithm are incorrect initial model-to-data matches that are encountered before the first correct initial model-to-data match. These can be eliminated using a more well-tuned initial evaluation function; however, as the initial model-to-data comparisons should be done quickly, there may be some limits to how much can be compared at the initial stage (where many hypotheses are tested).

To improve on these results, I believe that real benefits can be gained by reducing the number of features that need to be considered at a time. If the data features can be reliably partitioned into K subsets, which can be matched independently, and the model features can also be partitioned into some corresponding L subcomponents, then the brute-force version of the matching algorithm is reduced from $(M+1)^D$ to:

$$KL(\frac{M}{L}+1)^{\frac{D}{K}}$$

which is considerably less. To do this, some form of perceptual organization [21] is needed, such as a region or surface patch grouping algorithm (e.g. [16], [4] Chapter 5). This step will ultimately be necessary for most real visual matching problems. If the model and data features features could be organized into corresponding groups with no more than (e.g.) 5 features, then almost any algorithm is reasonable.

The performance of the geometric pruning search algorithms (Section 3.7) is a disappointment. The direct search phase (after the geom2 algorithm initial interpretation tree search) requires the additional search (assuming that the initial matches are correct):

$$\frac{1}{2}(T-2)(D-2)+(\frac{1}{p_v}-1)(T-2)(D-2)$$

The first term is the amount of search needed to match the $T-2$ remaining required model features, and the second term is the amount of search needed for checking other model features that were not visible. However, the real work comes from attempts to verify incorrect initial hypotheses, which has search cost:

$$(D-2)(M-T+1)$$

for each spurious hypothesis (of which there might be many). This complexity analysis assumes that all data features must be checked, but any form of spatial hashing that can allow direct access to a small subset of the data features using the predicted feature position has potential for dramatic increases in performance, as then the verification search cost would be $O(M)$ instead of $O(MD)$. However, in spite of its generally poorer performance, the geometric matching algorithms have other advantages in that they predict the position of sought-for features. This provides the possibility to overcome feature mismatch arising from over- and under-segmentation, and occlusion.

The complexity analysis and simulations done here and by Grimson [13] assume that the probability that three falsely matched features pass the binary matching constraint is $(p_2)^3$. However, M. Orr (private communication) finds that this has not been the case in his experience. A possible explanation is that when two features pass the binary tests and incorrectly match, the event of finding a third incorrect match compatible with either of the first two matches is not independent. The net result of this is an increase in the number of incorrectly matched hypotheses.

The analysis assumed that the amount of work was proportional to the number of nodes searched. This is only partially true, in that extending a hypothesis might require a number of binary tests equal to the number of non-wildcard matches currently in the hypothesis, which might require a lot of computational effort. If the hypothesis is valid, then all tests will be done. However, if the hypothesis is invalid, then it will be rejected, and the odds are that most will be rejected after a single binary test. As there are likely to be a much larger number of invalid hypotheses tested, the average number of binary tests should be close to one per hypothesis, so the modeling assumption is usually valid.

The analysis above also assumed that only one model (i.e. one set of model features) was considered when matching. If all models must be considered, then the computational complexity will be high, as the results in Section 4 showed that the performance of the best search algorithms on an incorrect model is about 3-17 times worse than with the correct model. Hence, some form of model invocation method is needed to reduce the number of candidate models (e.g. [4] Chapter 8, [13] Chapter 15).

However, in spite of the complexity of the analysis, the key conclusion is that by using the best-first algorithm as an alternative to the standard interpretation tree visual matching algorithm, it is possible to reduce the amount of search, the number of partial interpretations accepted and the computation time by about a factor of 10-20, where the precise amount of improvement depends on the problem parameters.

Acknowledgements

This research was funded by SERC (IED grant GR/F/38310, ACME grant GR/H/86905). Other facilities provided by University of Edinburgh. This paper benefited greatly from discussions with Dibio Borges, Sheila Glasby, John Hallam, Howard Hughes, Mark Orr, Kristian Simsarian, Martin Waite, Manuel Trucco and Mike Uschold. Particular thanks go to Andrew Fitzgibbon for the suggestion of

investigating the best-first algorithm which was designed, in part, through discussions with him and Josef Hebenstreit.

References

[1] D. W. Murray, B. F. Buxton. Experiments in the machine interpretation of visual motion. MIT Press, Cambridge, Mass. 1990.

[2] O. D. Faugeras, M. Hebert. A 3-D Recognition and Positioning Algorithm Using Geometric Matching Between Primitive Surfaces, Proceedings 8th Int. Joint Conf. on Artificial Intelligence, pp996-1002, 1983.

[3] R. B. Fisher. SMS: A Suggestive Modeling System For Object Recognition. Image and Vision Computing, 5(2), pp 98-104, 1987.

[4] R. B. Fisher. From Surfaces to Objects: Computer Vision and Three Dimensional Scene Analysis. John Wiley and Sons, Chichester, 1989.

[5] R. B. Fisher. Reducing Viewsphere Complexity. Proc. 1990 European Conf. on Artificial Intelligence, pp 274-276, 1990.

[6] R. B. Fisher. Determining Back-facing Curved Model Surfaces By Analysis At The Boundary. Proc. 3rd Int. Conf on Computer Vision, pp 296–299, Osaka, 1990.

[7] R. B. Fisher Non-Wildcard Matching Beats the Interpretation Tree. Proc. 1992 British machine Vision Conf., Leeds, pp 560-569, 1992.

[8] R. B. Fisher. Hierarchical Matching Beats The Non-Wildcard and Interpretation Tree Model Matching Algorithms. Proc. 1993 British Machine Vision Association Conf., pp 589–598, Surrey, 1993.

[9] R. B. Fisher Performance Comparison of Ten Variations on the Interpretation-Tree Matching Algorithm. Proc. 1994 European Conference on Computer Vision, Stochholm, Sweden, 1994.

[10] E. Freuder. Backtrack-Free and Backtrack-Bounded Search. in L. Kanal and V. Kumar (eds), Search in Artificial Intelligence. pp343-369, New York:Springer-Verlag, 1988.

[11] C. Goad. Fast 3-D Model Based Vision. in Pentland (ed), From Pixels To Predicates. pp371-391, Ablex Publishing, New Jersey, 1986.

[12] W. E. L. Grimson and T. Lozano-Perez. Model-Based Recognition and Localization from Sparse Range or Tactile Data. International Journal of Robotics Research, Vol. 3, pp 3-35, 1984.

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

[14] W. E. L. Grimson. The Combinatorics of Heuristic Search Termination for Object Recognition in Cluttered Environments. Lecture Notes in Computer Science, ECCV-90, Springer-Verlag, pp 552-556, 1990.

[15] W. E. L. Grimson. The Combinatorics of Object Recognition in Cluttered Environments Using Constrained Search. Artificial Intelligence, Vol 44, No. 1–2 pp 121–166, 1990.

[16] A. Guzman. Decomposition of a Visual Scene into Three-Dimensional Bodies. Proceedings Fall Joint Computer Conference, pp291-304, 1968.

[17] F. S. Hillier and G. J. Lieberman. Operations Research. Holden-Day, San Francisco, 1967.

[18] D. P. Huttenlocher and S. Ullman. Recognizing Solid Objects by Alignment with an Image. Intl. Journal of Computer Vision, vol. 5, no. 2pp. 195-212, 1990.

[19] D. P. Huttenlocher and T. A. Cass. Measuring the Quality of Hypotheses in Model-Based Recognition. Proc. European Conf. Comp. Vision, London, pp 773–777, 1992.

[20] N. J. Nilsson. Principles of Artificial Intelligence. Tioga Publishing, 1980.

[21] A. P. Witkin and J. M. Tenenbaum. What Is Perceptual Organization For? Proceedings 8th Int. Joint Conf. on Artif. Intel., pp1023-1026, 1983.

A Generic Best-First Matcher

```
// best-first algorithm
//
//      no re-use of data feature
//      termination on enough matches
//      bounded expansion of non-matches
//
#include <stdio.h>
#include <math.h>

#define MAXFEATURES 200
#define MAXSTATE    2000
#define MAXCOMPAT   MAXFEATURES*MAXFEATURES
#define LISTEND     ((state_record*) -1)

// data features
struct feature {
  int properties;       // some abstract properties
};
struct feature modelfeature[MAXFEATURES]; // model features
int nummodelfeat;             // number of model features
struct feature datafeature[MAXFEATURES];  // data features
int numdatafeat;              // number of data features

// state queue
struct state_record
{
  short state[MAXFEATURES]; // pairs added to state
  int numpairs;       // number of pairs added to state
  float stateeval;    // evaluation of state so far
  int nextpair;       // next entry to add to this state
  float nexteval;     // evaluation of the next addition
  struct state_record *next, *prev;  // doubly linked list
} st[MAXSTATE];
int stateused;
struct state_record listhead, listtail; // queue binding
struct state_record *freelist;

// run input parameters
float perreqd;     // percent of matched features required
int numfeatneeded; // number of features needed before termination
```

```
// derived run parameters
int searchlimit;   // amount searched before probability reduction
int modelsweep;    // number of model features initially explored

// statistics
int sumgenerated;
int sumnodecount;

// compatible pairs
struct compatstr
{
  short model,data;
  float plaus;
} compat[MAXCOMPAT];
int numcompat,allcompatgenerated,lastmodel,lastdata;
struct compatstr tmpcompat;

// unary and binary compatibility tests
int unarytest(int data, int model, double *eval)
{
  if (!compatible1(data, model)) return 0;
  *eval = compatibility_evaluation(data, model);
  return 1;
}
int binarytest(int data1, int model1, int data2, int model2)
{
  // unique data use
  if (data1 == data2) return 0;

  // relative property tests
  if (!compatible2(data1, model1, data2, model2)) return 0;
  return 1;
}

// estimate merit of adding this feature to current set of matches
// high plausibility matches add positive weight (up to 1)
// low plausibility matches subtract weight, down to -infinity
double fevalfunct(state_record *ste)
{
  double tempvalue;

  if (ste->numpairs == 0) tempvalue = compat[ste->nextpair].plaus;
  else tempvalue = ste->stateeval + ste->numpairs - 1
                      + compat[ste->nextpair].plaus;

  // decrement if too large of a search passed without success
  if (ste->numpairs == 1
    && ste->nextpair > ste->state[ste->numpairs-1] + searchlimit)
```

```
      tempvalue -= 1;
  else if (ste->numpairs == 2
    && ste->nextpair > ste->state[ste->numpairs-1] + 2*searchlimit)
      tempvalue -= 3;
  return tempvalue;
}

// get a new node for insertion
void getnode(state_record **newstate)
{
  // decide if 1) there is room for the newstate
  // and if not 2a) whether it falls off of the queue
  // bottom or 2b) the current queue bottom falls off
  if (!empty_freelist()) *newstate = nextfree();
  else
  {
    // last state falls off of Q bottom
    *newstate = listtail.prev;
    ((*newstate)->prev)->next = &listtail;
    listtail.prev = (*newstate)->prev;
  }
}

// insert a new queue entry.
// order by estimated next best state
void insertqueue(state_record *ptr)
{
  register state_record *here,*herenext;

  // insert ptr into queue
  // for now, do the dumb thing of searching from the top
  // could use some sort of tree sort, but statistics shows
  // that average walk in queue is about 1-2, so maybe not
  // worth the overhead of constructing and maintaining the
  // tree (although sometimes have to walk
  // several 100 nodes when demoting a hypothesis)
  //
  here = &listhead;
  do {
    herenext = here->next;
    if (ptr->nexteval > (herenext)->nexteval)
    {
      // insert here
      ptr->next = herenext;
      ptr->prev = here;
      (ptr->next)->prev = ptr;
      here->next = ptr;
      break;
```

```
    }

    // get next
    here = here->next;
  } while (1);
}


// pop the next top of state off
int poptop(state_record **top)
{
  if (listhead.next == &listtail) return 0;
  *top = listhead.next;
  listhead.next = (*top)->next;
  (listhead.next)->prev = &listhead;
  return 1;
}

// check for a compatible new state extension
int nextcompatible(state_record *teststate)
{
  for (int k=0; k<teststate->numpairs; k++)
  {
    if (!binarytest(
      compat[teststate->state[k]].data,
      compat[teststate->state[k]].model,
      compat[teststate->nextpair].data,
      compat[teststate->nextpair].model))
      return 0;
  }
  return 1;
}

// generate a new mode-to-data compatible pair, if possible
int get_next_unary_compatible()
{
  double evaluation;

  if (allcompatgenerated == 1) return -1;
  while (lastmodel < nummodelfeat)
  {
    while (lastdata < numdatafeat-1)
    {
    lastdata++;
    sumgenerated++;
    if (unarytest(lastdata,lastmodel,&evaluation))
    {
      if (numcompat >= MAXCOMPAT)
```

```
        {printf("Out of compat space\n"); exit(0);}
      compat[numcompat].plaus = 2*log(evaluation)+1;
      compat[numcompat].data = lastdata;     // data feat
      compat[numcompat].model = lastmodel;   // model feat
      sumnodecount++;
      return numcompat++;
    }
    }
    lastdata = -1;
    lastmodel++;
  }
  allcompatgenerated = 1;
  return -2;
}

// generate successor state to the given one and insert into
// priority queue if appropriate
void update_state(state_record *state)
{
  // see if any exist
  if (++state->nextpair < numcompat)
  {
    if (state->numpairs + (nummodelfeat-lastmodel+1)
        + (numcompat-state->nextpair+1) >= numfeatneeded)
    {
      // more states to try
      state->nexteval = fevalfunct(state);
      insertqueue(state);
    }
    else {state->next = freelist; freelist = state;}
  }
  // else make some more if possible
  else if (state->numpairs+(nummodelfeat-lastmodel) >= numfeatneeded)
  {
    if (get_next_unary_compatible() >= 0)
    {
      // more states to try
      state->nexteval = fevalfunct(state);
      insertqueue(state);
    }
    else {state->next = freelist; freelist = state;}
  }
  else {state->next = freelist; freelist = state;}
}

// main search algorithm
int searchtree()
{
```

```
  register int k,i;
  register struct state_record *newstate,*oldtop;

  do {
  // pop top of list and copy for new state
  if (!poptop(&oldtop)) return 0;

  // test new state
  sumgenerated++;
  if (nextcompatible(oldtop))
  {
    sumnodecount++;

    // check for early termination
    if (oldtop->numpairs + 1 >= numfeatneeded)
    {
      for (k=0; k<oldtop->numpairs; k++)
        printf("Model %d paired with Data %d Pair %d\n",
        compat[oldtop->state[k]].model,
        compat[oldtop->state[k]].data,
        oldtop->state[k]);
      printf("Model %d paired with Data %d Pair %d\n",
        compat[oldtop->nextpair].model,
        compat[oldtop->nextpair].data,
        oldtop->nextpair);
      return 1;
    }

    // copy new state
    getnode(&newstate);
    newstate->numpairs = oldtop->numpairs;
    newstate->stateeval = oldtop->stateeval;
    newstate->nextpair = oldtop->nextpair;
    newstate->nexteval = oldtop->nexteval;
    for (i=0; i<oldtop->numpairs; i++)
        newstate->state[i] = oldtop->state[i];
    newstate->state[newstate->numpairs++] = oldtop->nextpair;
    newstate->stateeval = oldtop->nexteval;

    // add new state into queue
    update_state(newstate);
  }

  // update old state
  update_state(oldtop);
  } while (1);
}
```

```
main(int argc,char **argv)
{
  float perreqd;    // percent of matched features required
  double evaluation;
  int i,j;

  /* check arguments */
  if (argc != 2)
  {
        printf("usage: bestmatch <\% required>\n");
        exit(0);
  };
  sscanf(argv[1],"%f",&perreqd);

  // load data
  nummodelfeat = load_model();
  numdatafeat = load_data();

  // calculate termination threshold
  numfeatneeded = (int)ceil(perreqd*nummodelfeat);

  // calculate search fan out limit
  modelsweep = nummodelfeat;

  // make up initial compatible pairs
  numcompat = 0;
  for (j=0; j<numdatafeat; j++)
    for (i=0; i<modelsweep; i++)
    {
      if (unarytest(j,i,&evaluation))
      {
        if (numcompat >= MAXCOMPAT)
          {printf("Out of compat space\n"); exit(0);}
        compat[numcompat].plaus = 2*log(evaluation)+1;
        compat[numcompat].data = j;     // data feat
        compat[numcompat++].model = i;   // model feat
        sumnodecount++;
     } }
  lastmodel = modelsweep-1;
  lastdata = numdatafeat-1;
  allcompatgenerated = 0;
  searchlimit = (int)ceil((double)(numcompat
                      / (nummodelfeat*(1.0-perreqd))));
  if (searchlimit > numcompat) searchlimit = numcompat;

  sort(compat);     // sort initial list

  // initialize the first element
```

```
  st[0].numpairs = 0;
  st[0].stateeval = 0.0;
  st[0].nextpair = 0;
  st[0].nexteval = compat[0].plaus;

  // initialize the search queue
  st[0].next = &listtail;
  st[0].prev = &listhead;
  listhead.next = &st[0];
  listtail.prev = &st[0];
  listtail.nexteval = -HUGE;

  initialize_the_freelist();

  // statistics to accumulate as tree explored
  // accounting for initial pre-comparison pass
  sumgenerated = numdatafeat*modelsweep;
  sumnodecount = 0;

  // do model match
  if (!searchtree())
        printf("*** NO SOLUTION FOUND ***\n");

  printf("Nodes unpruned:%d nodes generated:%d\n",
        sumnodecount,sumgenerated);
}
```