A New Model Matcher for the IMAGINE II Object Recognition System

Josef Hebenstreit

MSc Information Technology: Knowledge Based Systems
Department of Artificial Intelligence
University of Edinburgh
1994

Abstract

The new model matcher discussed here counters the combinatorial explosion in searching the whole interpretation tree by using a best first search strategy and exploiting constraints to prune the search tree. Several heuristics have been investigated to guide the search. Parameter for constraint evaluation are related to parameter expressing performance of the algorithm. The matching algorithm has been tested on two objects with different degree of difficulty. In addition the relationship between heuristic evaluation and performance has been investigated.

Acknowledgements

I would first of all like to thank my first supervisor, Dr R. B. Fisher, for his guidance during the course of the research reported here. Special thanks for my second supervisor, Andrew Fitzgibbon, for the excellent support during my project. Without his help and knowledge the project wouldn't have been finished in time.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problems with the current Model Matcher

The current model matcher in IMAGINE II employs the standard interpretation tree search and explores the tree in depth-first order using backtracking. Constraints are used to prune parts of the search tree and thus to avoid exhaustive search of the whole interpretation tree. Pruning is a standard technique to cope with search complexity.

Although the performance of the standard interpretation tree search has been improved by taking advantage of data and model ordering there are fundamental problems that the search method can't cope with. The main problems are:

1. non-exhaustive interpretation tree exploration

2. non-directed interpretation tree exploration

3. redundant search

The first case occurs when a partially consistent path includes the correct hypotheses. If all paths containing the true match are partially consistent but not globally consistent then the algorithm can never find the correct hypotheses and simply fails.

The second case points to the problem that only locally best candidates are explored first that is the current model matcher doesn't take advantage of the possibility that the globally most plausible data model pairing could be explored first. This is clearly a disadvantage of the current algorithm and one would like to

have an algorithm which is able to exploit a global ordering of the pairings rather than a local one.

An additional problem strongly related to the second case is the search method itself and the way the geometric consistency test works. The exploration of the tree is fixed and determined by depth-first search. There is no way to collect evidence for more promising candidates while exploring the interpretation tree. The consistency test applied to each node is only a pass or fail test which means that candidates which are not promising are completely ignored and there is no way to return to them in a later stage of the tree search.

The third case is a deficiency introduced by the wildcard model feature. Wild cards are introduced to cope with spurious data. This causes redundant search by exploring all possible combinations of wildcards and data features. Although the redundant search effort can be minimized it can't be completely avoided.

1.2 Results with the New Model Matcher

The new algorithm originated in discusssion with R.B. Fisher and A. Fitzgibbon has been implemented and tested on several objects. In addition an SMS model for a new test object has been built and tested as well.

Several criteria have been used to measure the performance. Parameters extracted from the total heuristic function were related to parameters expressing the performance. The experiment was done on the new test object, the "Widget Part" and the "Renault Part". As a result the Widget Part could be recognized in only one raytracing operation whereas the Renault Part, known as the most complex object in vision literature, failed to find the correct hypotheses. The reason for the failure is based on bad heuristic estimates and not the algorithm in principle.

1.3 Structure of the thesis

The background chapter starts with motivation for object recognition, gives psychological and biological evidence about the importance of shape-based object recognition, states the goal of object recognition and introduces IMAGINE II. It continues with the current model matcher in IMAGINE II, analyses the standard interpretation tree, explains the pruning technique, states the interpretation tree search algorithm with wildcard and finally gives a short survey of other model matchers.

In chapter three the new algorithm is discussed in detail. It starts with the new interpretation tree, states and proves complexity measures, illustrates it on the basis of a simple example and introduces the basic principle of best-first search. It emphasizes the importance of heuristic evaluation, outlines the main points of the new algorithm, explains each point, especially the important issue of node expansion, presents the best-first search algorithm and gives a simple but complete example showing how the algorithm works.

In chapter four the total heuristic function is explained in detail. It starts with the geometric consistency function, explains its components, analyses each component in detail, gives geometric interpretations of each geometric consistency evaluation function and states the total heuristic functions used in the experiment. It explains each parameter and ranges of the parameter and the method used to accomplish the experiment. Finally the results of the experiment with two objects are reported.

Chapter five summarizes the key results, discusses the problems involved and gives suggestions for future work.

Chapter 2

Background

2.1 Short Survey of Object Recognition

2.1.1 Motivation

Although we humans beings take vision for granted we can hardly imagine how difficult it is to build a vision system of comparable performance to that of humans or animals. Biological vision has taken several hundred million of years to evolve to the stage seen today. Why is one interested to mimic biological vision? Well, the answer is easy. As a long term goal one would like intelligent autonomous robots that do all the jobs we like them to do and vision plays therein a crucial role. Of course there is strong economical and military interest in the exploitation of a successful vision system. There would then be a huge number of applications that are within reach of current technology and knowledge.

Object recognition is one of the most important aspects of visual perception. To get an idea why it is difficult let's look at an simple example. Suppose you have a cartoon line drawing. Young children have no difficulty recognizing it but there isn't any existing computer recognition scheme which could do it. We often recognize objects (a car, a familiar face, a printed character) visually on the basis of its characteristic shape, but there are also other ways to recognize objects. The recognition of trees is more based on texture properties, branching pattern and colour than on precise shape. Also certain animals can be recognized mainly on the basis of texture and colour pattern (e.g a giraffe or a tiger). The relative location of other objects can be used to recognize objects e.g an unusually shaped door knob can be recognized on the basis of its location relative to the door (

Global matching techniques use global features or parameters, such as area, volume, perimeter, higher order moments, spatial frequency description, distance measure etc, that clearly depend on the entire object, in order to find a comparison between model and image. The set of global parameters for both the model and the image data can be represented as a vector of parameters. Matching proceeds by comparing the image vector with a set of model vectors and selecting the best matching model. The pose of the object in the image can sometimes be calculated from the parameters of the best match. There are several drawbacks with this approach. First, the global properties may depend on the illumination condition, surface reflectance properties of the object, the position, strength and distribution of the light sources , the orientation of the surface patch reflecting the light to the image point (of course, this not a problem for range images and is mentioned here only for completeness). Second, the global matching approach can't deal with occlusion and spurious data and is therfore unable to distinguish between overlapping objects. For further discussion see citation list on page 24 in [?].

An interesting 3D global matching algorithm is the *iterated closest point matching technique.* It tries to match a set of data points against a set of model points by minimizing the distance between them. It iteratively solves the global minimization problem, estimates the pose of the object and moves towards the object until a steady state within some error is reached. One advantage of this method is that there are no restrictions in the shape of the objects to be matched, it is applicable to polyhedra as well as free formed objects. The main disadvantage is the high matching complexity and occlusion. It can't exploit local features in order to confine the search space.

Local feature based matching or simply feature matching is dominant in most real vision systems because it copes with the problem of occlusion and spurious data. One example using feature matching is the interpretation tree search which has already been discussed in the previous section. There are several variations of it. One uses the ordering of data features to minimize the search. One can order the data feature by saliency by taking the largest feature first, such as surface patches with the largest area [Grimson & Lozano-Perez 87, Ayache & Faugeras 86]. One can also start with a data feature and then use proximity or connectivity to select the next one [Ayache & Faugeras 86]. One can also order the model feature

in some way to reduce the tree search. If data features are ordered according to connectivity one can apply the same connectivity ordering to model features [Ayache & Faugeras 86]. Another very effective way to reduce the search effort is the exploitation of the visibility constraint. Each pose has only a limited number of model features which are visible. Model features that are not visible from a certain pose can simple be ignored.

Alignment or hypothesize-and-test techniques are another variation on the interpretation tree approach. The key idea is that only a minimum number of data-model pairings sufficient to calculate a complete transformation are used. The transformation is used to align the model with the data and the alignment in turn is used to predict model features that might be evident in the data and to test for possible matches. Thus additional matches are used to get a better pose estimation. For further discussion of the alignment method.

The local feature focus (LFF) method develoved by Bolles and collaborators utilizes focal features to guide the search. It first tries to match a model focus feature against image data and for each successful match it tries to find nearby features in the image. Then, an association graph is constructed and the maximal clique is computed. The maximal clique represents maximal clusters of mutually consistent assignments of data and model features. From the maximal clique the pose of the object model can be estimated and verified on the image data. Although LFF can reduce the search expense drastically there must be focal features visible in the image for the method to work.

Chapter 3

New Matching Algorithm

3.1 Principles

This section starts with the traditional interpretation tree and gives reason for the introduction of the new algorithm. It then introduces the new interpretation tree, analyses its complexity and gives an informal description of the algorithm. In the next section a detailed description of the new algorithm is given including a short description of the data structures used. In the last section the data structures and the implementation of the algorithm are covered in detail.

3.1.1 Motivation for the New Algorithm

In order to appreciate the underlying ideas of the new algorithm one should first start with the current model matcher and point out the weak points of the algorithm employed.

The current model matcher in IMAGINE II employs the classical interpretation tree search and explores the tree in depth-first order using backtracking. Regarded as a black box the matcher essentially gets a set of pairings of data and model surface patches with the corresponding plausibility values from the invocation module. Thus each pairing is assigned a plausibility measure telling how probable it is that the pairing is part of the hypothesis. The invocation module not only reduces the number of possible data-model pairings drastically but also evaluates the pairings remaining. Thus the invocation process is the first and an important part of the overall model matching process. As it has been already discussed in Chapter 2 the matcher generates a set of possible hypotheses which are verified

in the geometric reasoning module to get a unique interpretation of the data in question.

The current interpretation tree search algorithm in IMAGINE II basically uses three ordering principles in order to reduce the search complexity:

1. viewpoint-centered models

2. data-patch size

3. model-patch ordering according to the invocation plausibilities

The division of the complete body-centered model of the object in a set of viewgroups containing the so called viewpoint-centered models allows matching of each of this groups against the whole set of data surface patches. Obviously the viewgroups contain only a subset of all model surface patches of the body-centered model and by assigning a separate interpretation tree to each viewgroup the entire search space can be significantly reduced compared with the interpretation tree using the body-centered model.

The interpretation tree search needs some ordering (either arbitrary or systematic) for the data patches to be matched. The input to the model-matcher is a sparse matrix of possible data-model pairings and invocation plausibility values. The algorithm clusters the pairings according to the set of data surface patches, where each cluster contains pairings with the same data patch in. The clusters are sorted in descending order according to the size or area of the data-surface patch involved. Each cluster corresponds to a certain depth-level in the interpretation tree and the cluster with the largest data-patch is examined first. The reason for this ordering is the fact that large data-patches are providing more constraints and should thus be explored first.

There is also an ordering inherent in the clusters itself. The data-model pairings in each cluster are sorted in descending order according to the invocation plausibilities. So the most plausible pairing within a cluster is examined first. It should be pointed out here that the ordering of pairings within a cluster is local with respect to invocation plausibility.

The current model matcher doesn't take advantage of the possibility that the most plausible data-model pairing could be explored first. This is a key disadvantage in the current algorithm and one would like to have an algorithm which is able to exploit a global ordering of the pairings. In the next section this idea is refined and embedded in the new algorithm.

3.1.2 Interpretation Tree for the New Algorithm

The key motivation for the new algorithm is to use a best-first strategy rather than a depth-first one in order to reduce search complexity. Global ordering of data-model pairings should also be exploited in the new algorithm. The new interpretation tree presented here and the corresponding algorithm discussed in detail later on in this chapter originated in discussion with R.B Fisher and A. Fitzgibbon.

The new interpretation tree consists of nodes representing model-data pairing with their associated invocation plausibility values. Each node has also an index to the corresponding data-model pairing but this is not relevant for the algorithm and will be neglected here. Each node has children with invocation plausibilities sorted in descending order from left to right. The invocation plausibility of the parent node is greater or equal than the values of the children. Suppose an arbitrary element selected from the ordered list of invocation plausibilities is the current parent node. The children of this node from the left to the right are then the rest of the invocation list starting with the element immediate right to the node in the list representing the parent node till to the end of the list. The level and the position within the level of the interpretation tree determine the position of the parent node in the list. The term level or equivalently depth of a tree is simply the number of branches along the way from the root node to the node in question.

The new interpretation tree is a directed acyclic graph with the special property that no child node has two different parents. In contrast to the traditional interpretation tree, here each node has varying number of branches depending on the location in the tree.

The interpretation tree starts with the root node, which is only a dummy node, and expands to all children $s_1, \cdots, s_n$. The node s_1 itself expands to $s_2, \cdots, s_n$ and

node s_2 expands to $s_3, \cdots, s_n$ and so on. All children of s_1 and s_2, etc, also expand in the same way. This process of expansion the tree continues until the leaf-node s_n, which has the smallest plausibility value, has been reached in which case no further expansion is possible (see Figure 3–1).

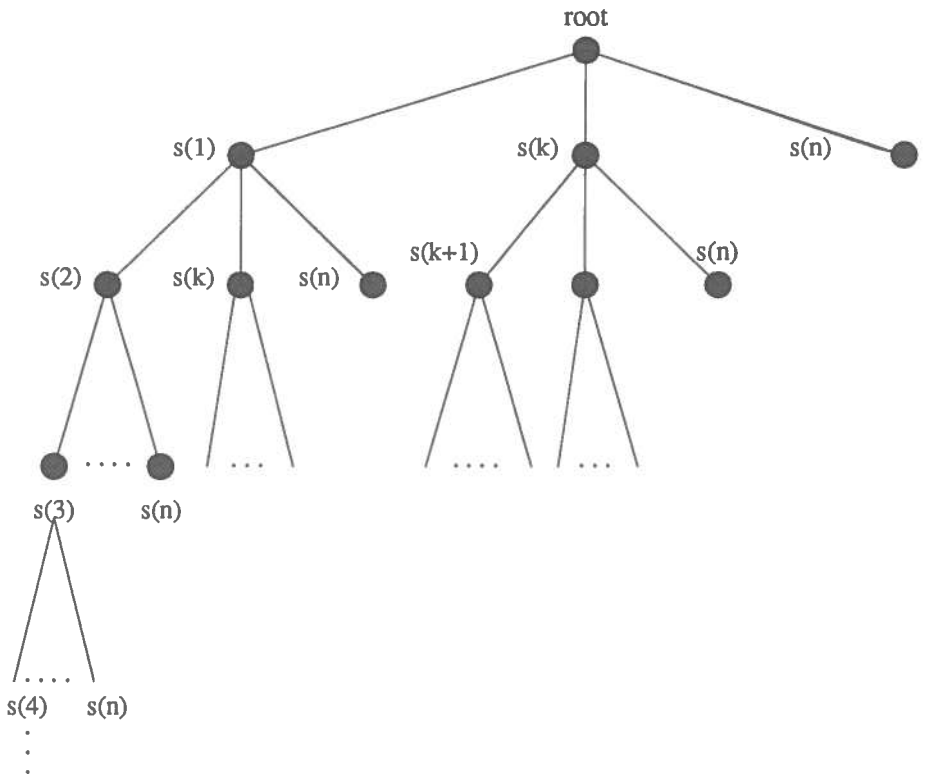

Figure 3–1: *New Interpretation Tree*

The structure of the interpretation tree is a left-hanging tree with the longest path to the left and the shortest path to the right whereas path here is meant to be the way from the root node to the leaf node. An important observation is that paths with the same path-length have invocation plausibility values assigned which are decreasing from left to right. The idea behind this is that paths with higher invocation plausibility values should be favoured. A second important observation is that longer paths are more concentrated to the left of the interpretation tree. Here again longer path are favoured and should be explored first. A third important observation is that each sequence of invocation plausibility values corresponding to the path from the root node to an arbitrary node is monotonically decreasing. A chosen path has a unique sequence of invocation plausibility values

or indices to their list and therefore no other permutation of the sequence need to be considered as can be seen in Figure 3–1. The fourth observation is that the rightmost possible child of a parent is always a leaf node, a fact which is an important consequence of the algorithm as will be seen later on.

The complete interpretation tree can be build up recursively. Suppose one would like to build up an interpretation tree of order n (n is the length of the list of invocation plausibility values or the longest path in the tree or the number of children or branches from the root node). Then the smallest recursive structure is the root node and the rightmost child n of it. This constitutes the smallest unit which is used in the next stage. The unit is copied downwards such that the root is aligned with the $(n-1)$-th child of the root which is the sibling immediate to the left of the n-th node of the root. This constitute the second smallest unit. Then this unit is copied downwards such that the root of this unit is aligned with the $(n-2)$-th child of the root node. After $(n-1)$ copy operation or recursion the complete interpretation tree of order n is built. This structural principle should be kept in mind if one would like to build a complete interpretation tree.

Having discussed this property of the interpretation tree we are able to analyse its complexity. The first complexity measure is the number of nodes in the tree. In order to calculate it one have to consider the number of different sets for level one, for level two, etc, up to level n and simply sum it up. Table 3–1 treats the general case and will be justified by a generalisation of a simple example which follows later on (n is the order of the interpretation tree).

The number of nodes in a complete interpretation tree of order n is:

$$\sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) = 2^n$$

The second complexity measure of interest is the number of leaf nodes in a complete interpretation tree. Using the above formula as the induction hypothesis one can argue that the number of leaf nodes at level n is simply the difference of the number of nodes at level n and the number of nodes at level $n-1$. Thus the following statement is valid:

The number of leaf nodes in an interpretation tree of order n is:

$$2^n - 2^{n-1} = 2^{n-1} \cdot (2-1) = 2^{n-1}$$

level in the tree	number of different sets with level elements
0	$\binom{n}{0}$
1	$\binom{n}{1}$
2	$\binom{n}{2}$
$\vdots$	$\vdots$
i	$\binom{n}{i}$
$\vdots$	$\vdots$
n	$\binom{n}{n}$

Table 3–1: *Number of nodes for each level*

Now we want to justify the above formulae by analysing a simple interpretation tree of order 4 (see Figure 3–2).

To start with we consider each level of the tree and calculate the number of combinations with level elements. The table below summarizes the results.

level in the tree	combinations of level elements			
0	$\{s[0]\}$			
1	$\{s[1]\}$	$\{s[2]\}$	$\{s[3]\}$	$\{s[4]\}$
2	$\{s[1], s[2]\}$	$\{s[2], s[3]\}$	$\{s[3], s[4]\}$	
	$\{s[1], s[3]\}$	$\{s[2], s[4]\}$		
	$\{s[1], s[4]\}$			
3	$\{s[1], s[2], s[3]\}$	$\{s[1], s[3], s[4]\}$	$\{s[2], s[3], s[4]\}$	
	$\{s[1], s[2], s[4]\}$			
4	$\{s[1], s[2], s[3], s[4]\}$			

Counting the above elements and summing it up gives the number of nodes in

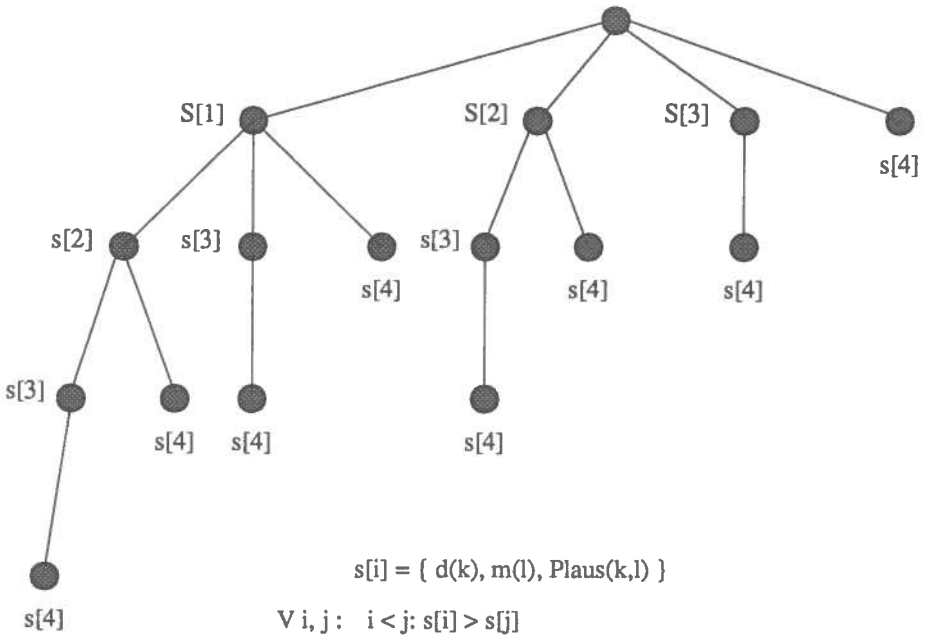

Figure 3–2: *Sample Interpretation Tree of Order 4*

a complete interpretation tree of order 4:

$$\sum_{k=0}^{4} \binom{n}{k} = 2^4$$

The same result can be obtained by considering the expansion of the tree from level 3 to level 4. Each node gets an additional leaf node (in our case $s[4]$) which simply doubles the total amount of nodes in the tree.

The observation made in the above example can be used to conduct an induction proof for the general case. The induction hypothesis for the trivial case of the first level is two. The induction step is a multiplication factor of two that is the total number of nodes in the next level is the double of the current one. Thus applying the induction step to an arbitrary level leads finally to the general result already stated. This argument also implicitly proofs the number of leaf nodes in the complete tree.

3.2 Algorithm and Data Structure

Having discussed the new interpretation tree in detail one may ask how this structure relates to the new algorithm. The structuring principle is only a consequence of the algorithm and doesn't say anything about the dynamic behaviour of the algorithm. Below a detailed discussion of the algorithm is given and the dynamic behaviour of the algorithm is explained on the basis of a simple example followed by a short discussion of the data structure used by the algorithm. To begin with the basic principles of best-first search are represented and explained.

As you already know one would like to avoid depth-first as well as breath-first search because they explore the search space systematic but in a brute force way until a solution or hypothesis is found. In the worst case they explore an exponentially sized search space and this results in an exponentially sized time complexity. By incorporating knowledge about the search space one is able to direct the search towards the correct solution. This knowledge also called heuristic is an important feature of each informed search algorithm.

The approach taken in the new algorithm is the use of a best-first strategy to guide the search. The search is explicitly guided by an estimate which tells the node in question how good it is on some global scale, which can then be used to order the next nodes to be explored. If any other node anywhere in the tree has a better estimate then that node should be expanded next. Thus best-first search employs four important steps.

1. select the best candidate

2. expand the tree

3. estimate the goodness of the expanded nodes

4. order the nodes according to the estimates

The estimation step is crucial for the algorithm to work properly. As mentioned above the heuristic employed is a knowledge about the search space and

is expressed as a single number which can then be used by the algorithm. The heuristic evaluation function, which return a number, consist of two parts namely the actual estimate of the previously combined pairings and the heuristic estimate of the value of the new addition. Expressed mathematically one have:

$$f(n) = g(n) \star h(n)$$

where $f(n)$ is the total heuristic estimate, $g(n)$ is the actual estimate, $h(n)$ is simply the heuristic estimate and the special character "$\star$" is an arbitrary operator combining the two estimates in some specific way.

The selection step and order step are related in a sense that one selects the best candidate from an ordered set of nodes. The ordered set could be implemented as a sorted list or as a priority queue. A sorted list is not absolutely necessary because one only want to know what the best element in the list is. Therefore a priority queue is a satisfactory solution. The best candidate is meant to be a candidate with the best total heuristic estimate expressed as $f(n)$ which is as already mentioned a single number.

The expansion step is a defined strategy to expand the best node either to all other nodes connected to it or to select only few nearby nodes depending on the strategy used. These expanded nodes are estimated by a heuristic evaluation function and then inserted in the priority queue. The priority queue uses only the total heuristic estimates of the nodes to be inserted and the nodes already in the queue to establish its internal order.

Having explained the basic principle of best-first search we are able to discuss the basic ideas of the new algorithm. Each node in the tree has a number indicating which invocation plausibility value is assigned to it. It is essentially an index to an array of invocation plausibility values which can be seen as a sort of cache. Number or index "1" correspond to the highest invocation plausibility value and index "n" to the lowest one. The ascending order of indices thus correspond to the descending order of invocation plausibility values.

Each node in the tree keeps record of the path explored so far, and is implemented as a parent pointer for efficiency reasons, and an index to indicate which child to expand next. But this is only implementation detail and is not significant

for the understanding of the algorithm. What is really important is the fact that each node stores two kinds of information:

1. total heuristic estimate of the current node

2. actual estimate of the parent node

The total heuristic estimate is a measurement of how plausible the path from the root node to the current node is to be the correct hypothesis. As mentioned above the total heuristic estimate consists of an actual estimate and a heuristic term. The actual estimate of the parent node is calculated by a geometric consistency evaluation of the set of model-data pairs matched on the path from the root node to the parent node. It is important to emphasize that this evaluation doesn't work like a function returning a value indicating pass or fail. It rather returns a value saying how plausible the path is to be geometrically consistent. This is a significant improvement compared to the current model matcher which uses a threshold function (avoiding thresholds is a good thing because one gets rid of parameters which the system performance depend on).

In expanding a node one has to consider two important cases:

1. open node (not a leaf node)

2. leaf node (node from which no expansion is possible)

If an open node with index k is to be expanded then the first descendent is its child with index $k+1$ and the second descendent is the child $k+1$ generated from its parent node. Thus the second descendent is the next sibling to the right (see Figure 3–3).

Before node k is expanded it has the total heuristic estimate $f(\{s_1, \ldots, s_k\})$ and the actual estimate $g(\{s_1, \ldots, s_i\})$ of its parent node. After expansion the first descendent is assigned the actual estimate $g(\{s_1, \ldots, \ldots, s_i, s_k\})$ of the node currently being expanded (i.e. the parent of the first descendent) and the total heuristic estimate $f(\{s_1, \ldots, s_k, s_{k+1}\})$. The second descendent is assigned the actual estimate $g(\{s_1, \ldots, s_i\})$ delivered by the node currently being expanded and

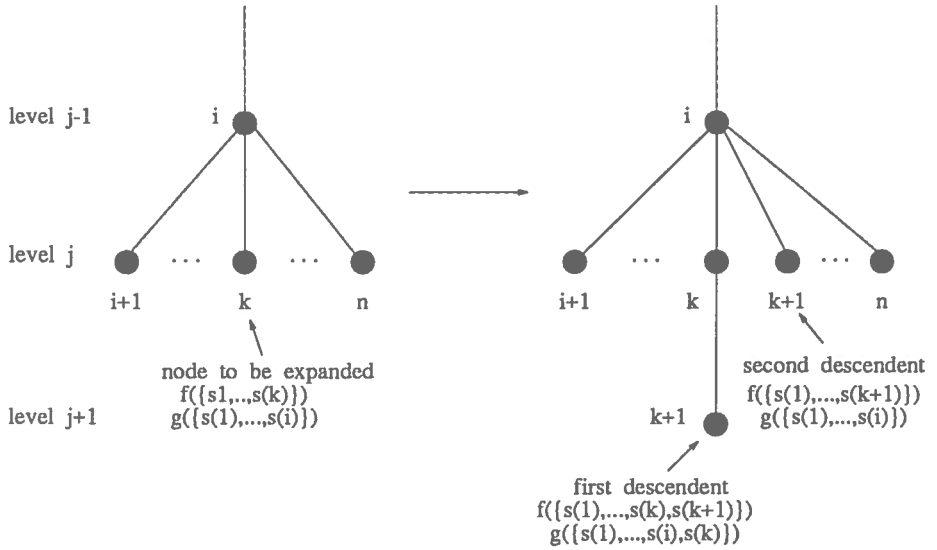

Figure 3–3: *Open node expansion*

the total heuristic estimate $f(\{s_1, \ldots, s_{k+1}\})$. In both cases the actual estimate must be known first in order to calculate the total heuristic estimate.

However, if one has reached a leaf node special treatment is necessary. The leaf node is marked as a special node saying that this node has been evaluated by the geometric consistency function and both the field with the total heuristic estimate and the field with the actual estimate gets the actual estimate of this node assigned. In other words, the node is geometric consistency evaluated. Although no further expansion from the parent node of the leaf node is possible a new element is generated and both fields the total heuristic estimate and the actual estimate are assigned the actual estimate of the parent node. In addition this node is marked as a special node indicating that this node has been evaluated by the geometric consistency function. Thus one node, the leaf node, comes from the priority queue and two nodes with geometric consistency evaluation go in the priority queue.

To summarize: before the leaf node is evaluated it has the total heuristic estimate $f(\{s_1, \ldots, s_n\})$ and the actual estimate $g(\{s_1, \ldots, s_i\})$ of its parent node. After evaluation the leaf node is assigned the actual estimate $g(\{s_1, \ldots, s_n\})$ and the total heuristic estimate is assigned the same value $g(\{s_1, \ldots, s_n\})$. It is denoted as a special node and goes in the priority queue. For the parent node a new element

is created. It is assigned the actual estimate $g(\{s_1, \ldots, s_i\})$ of the parent node and the total heuristic estimate is assigned the same value $g(\{s_1, \ldots, s_i\})$. As above it is denoted as a special node and goes in the priority queue. See Figure 3–4.

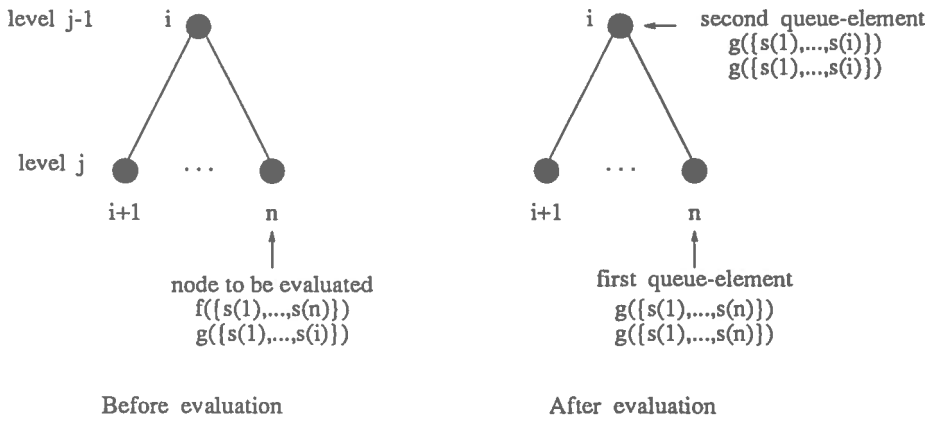

Figure 3–4: *Leaf node evaluation*

It is important to emphasize that the queue contains nodes ranked by both total heuristic estimate and the actual estimate (or the geometric consistency estimate) and both are equally treated in the internal orderings procedure of the priority queue. As is already known the rightmost possible child of each node is the leaf node. Each leaf node in the tree has an index n according to our convention. An important observation is that all children must have been evaluated before the parent node goes in the priority queue with its actual evaluation. Thus the parent node i in Figure 3–4 is geometric consistency evaluated and goes in the priority queue if and only if all $(n - i)$ children of the parent node have been evaluated or equivalently the leaf node of the parent node has been evaluated.

The algorithm uses three types of priority queues:

1. heuristic queue

2. geometric consistency queue

3. raytracing queue

There is a time ordering in the way these priority queues are filled. The algorithm gets the input from the invocation module, processes the data and fills the

heuristic priority queue. In the heuristic queue, a mixed set of elements both with geometric consistency and total heuristic evaluation is kept. Candidate hypotheses to be verified or raytraced are stored in a geometric consistency queue. If the best element of the heuristic queue (which is the top element of that queue) has been geometric consistency evaluated then it is taken off that queue and inserted in the geometric consistency queue. Because the geomertic consistency queue is also a priority queue the raytracer is able to select the best element from a set of possible candidates or hypotheses. After a while the raytracer takes the top element of the geometric consistency queue and starts the verification step by estimating the pose of the object and applying the raytracing function. The output of the raytracer is stored in a raytracing queue and to enable further refined analyses.

The question when one would like to take off the element from the geometric consistency queue is an important one. A simple minded solution like asking for a certain queue length is problematic because this threshold is most likely to be object dependent. In general, one would like to avoid such thresholds.

A better way to solve the problem is to start the raytracer as a function of the the number of node expansions made and the length of the geometric consistency queue. Although this is a possible solution a far better one exists. By introducing parallel processes one resolves all problems in one stroke. There are at least two processes working in parallel. One process is operating on the heuristic priority queue and the other one on the geometric consistency queue. Because both processes are working in parallel no queue length need to be examined. While there are elements in the geometric consistency queue the raytracer takes the best candidate and analyses it. During the analysis the geometric consistency queue may be filled or not. After analysis, the next candidate from the geometric consistency queue is selected until no one is left.

Having discussed the underlying principle of the algorithm we are able to present the pseudo-code description of the best-first search algorithm:

```
Initialize heuristic queue;
Initialize geometric consistency queue;
assign first element {s(1)} to best;
while heuristic queue not empty
{
   while best not geometric consistency evaluated
```

```
    {
       if best not a leaf node
          generate and evaluate first descendent
          insert in heuristic queue
          generate and evaluate second descendent
          insert in heuristic queue
       else
          update best
          insert in heuristic queue
          create element for parent node of best
          update element
          insert element in heuristic queue
       get new best candidate from top of heuristic queue
    }
    move best to geometric consistency queue
    get new best candidate from top of heuristic queue
}
search failed

//  Raytracer

while verification not satisfactory
{
     remove best element from geometric consistency queue
     apply raytracer to this element and assign plausibility
     insert element in raytracing queue
}
```

Before we move to implementation details a simple example is given to demonstrate the algorithm followed by a short description of the data structure used by the algorithm. More details to the data structure are given in the next section.

Suppose we have to consider four pairings denoted by s_1, s_2, s_3 and s_4 in descending order of invocation plausibility values. As you already know the search space has a complexity of $O(2^4)$ and is depicted in Figure 3–2. Assume the correct hypotheses is the set of pairings $\{s_1, s_2, s_3, s_4\}$. The algorithm starts with the first element s_1 and generates the first descendent s_2 on level 2 and the second descendent s_2 on level 1. Both descendents go in the priority queue and are ordered in such a way that the top element has the highest total heuristic estimate. Figure 3–5 illustrate the current state.

Suppose the top element is s_2 on level 2 which has the total heuristic estimate

of $f(s_1, s_2)$. Then the algorithm proceeds by removing this node from the priority queue and expanding its descendents. The first descendent is node s_3 on level 3 and the second descendent is node s_3 on level 2. Once again both nodes go in the priority queue and the whole set stored in the queue is reordered such that the element on top of the priority queue has the highest total heuristic estimate.

Suppose the top element is s_3 on level 3 which has the total heuristic estimate $f(s_1, s_2, s_3)$. Once again this node is taken off the priority queue and the algorithm generates the first descendent s_4 on level 4 and the second descendent s_4 on level 3. Both elements are inserted in the priority queue and the algorithm maintaining the priority queue ensures that the element on top of the queue has the highest total heuristic estimate.

Suppose the top element is s_4 on level 4 with the total heuristic estimate $f(s_1, s_2, s_3, s_4)$. The algorithm takes this element off the queue and tries to expand it. Because s_4 is a leaf node no further expansion is possible. Here we have the special case that node $\{s_1, s_2, s_3, s_4\}$ need to be geometric consistency evaluated. Thus the total heuristic estimate $f(s_1, s_2, s_3, s_4)$ gets replaced by the actual or geometric consistency estimate $g(s_1, s_2, s_3, s_4)$ and is marked as a special node and is inserted in the priority queue. A second node the parent node $\{s_1, s_2, s_3\}$ of the previous node $\{s_1, s_2, s_3, s_4\}$ with the path $\{s_1, s_2, s_3\}$ need to be considered. As explaind above a new element is created. This element gets the value for the actual estimate from the child and leaf node s_4. Because there is no further descendent to be considered the total heuristic estimate is the same as the actual or geometric consistency estimate of this node. Thus both fields of this element gets the value $g(s_1, s_2, s_3)$. In addition it is marked as a special node or element and is inserted in the priority queue.

Assume that node $s4$ with path $\{s_1, s_2, s_3, s_4\}$ has the highest total heuristic estimate. Thus the value of the total heuristic estimate of this node is $g(s_1, s_2, s_3, s_4)$. The algorithm recognizes that an element with geometric consistency evaluation is on top of the queue by means of a flag set in the data structure of this element during the evaluation process. The algorithm continues by taking this element off the heuristic queue and inserting it in the geometric consistency queue. In our case node $\{s_1, s_2, s_3, s_4\}$ with total heuristic estimate $g(s_1, s_2, s_3, s_4)$ is inserted in that queue. Thus the algorithm is repeating the process of node expansion and

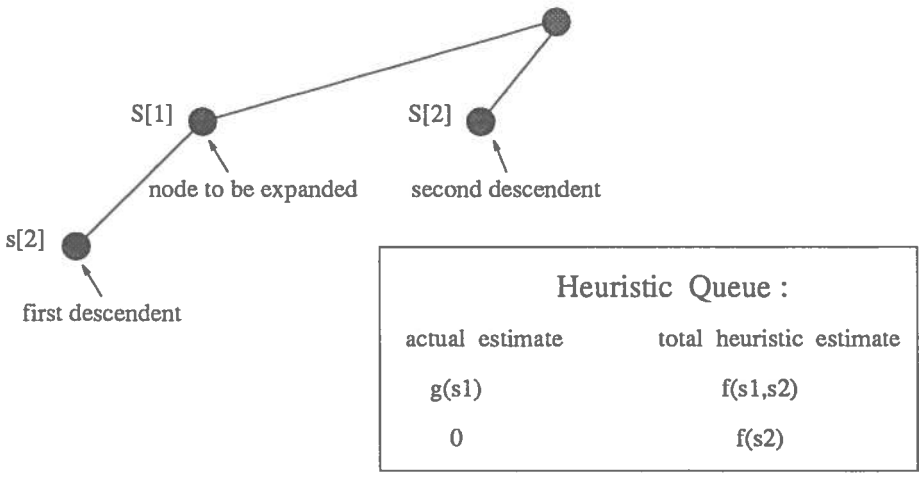

Figure 3–5: *First step in exploring the search space*

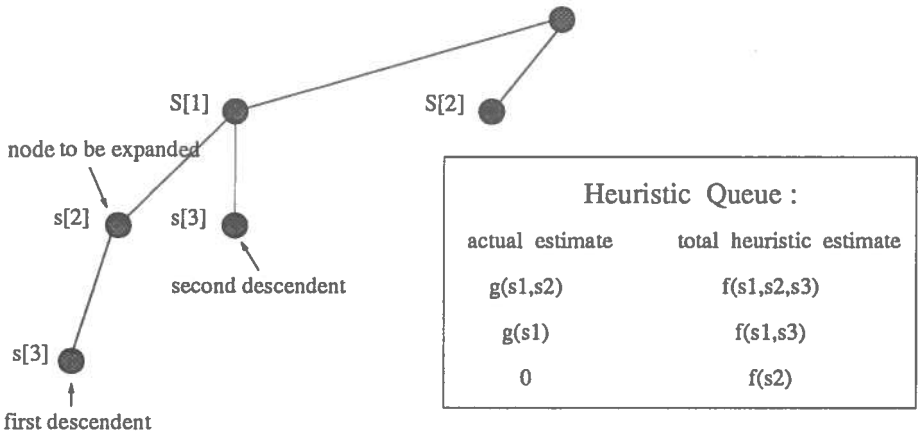

Figure 3–6: *Second step in exploring the search space*

each time a node with geometric consistency evaluation is on top of the queue the element is removed from the heuristic queue and inserted in the geometric consistency queue.

The most important data structure in the algorithm is the priority queue that maintains the best candidate with the highest total heuristic estimate. A simple minded realization of the priority queue as a linear sorted list is not appropriate because the queue might become quite large and thus queue insertion times would be significant.

A more efficient implementation is the use of a heap data structure and will

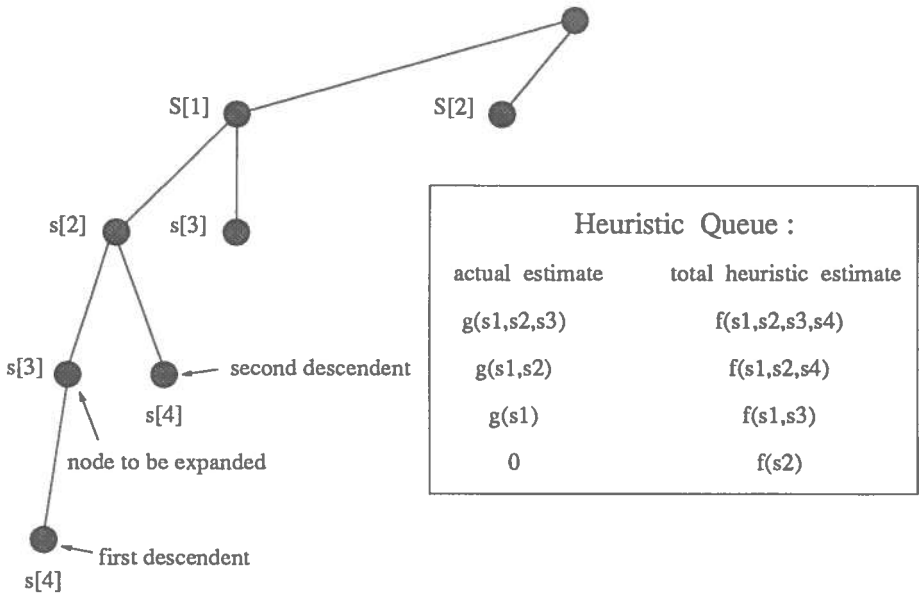

Figure 3–7: *Third step in exploring the search space*

be discussed in detail in the next section. There is a second data structure that maintains the interpretation tree. The two data structures are maintained separately. Only a node pointer in the queue element of the priority queue points to the second data structure. One doesn't want to store the invocation plausibility values in the node data structure because storage is wasted. Instead "Caching" is used. Only an index to the array of invocation plausibility values is employed. The interpretation tree itself is represented as a set of linked list of parent pointer and the queue element in the priority queue has only a pointer to a specific node in the tree. The path is extracted by following the parent pointer. For more detail see the next section.

3.3 Implementation Details

The algorithm for the new model matcher has been implemented in $C++$. $C++$ is a high level language supporting object-oriented programming. It has been chosen as a implementataion language because the current software of IMAGINE II itself is written in $C++$. Apart from this constraint the use of an object oriented language has several advantages. It enables and support

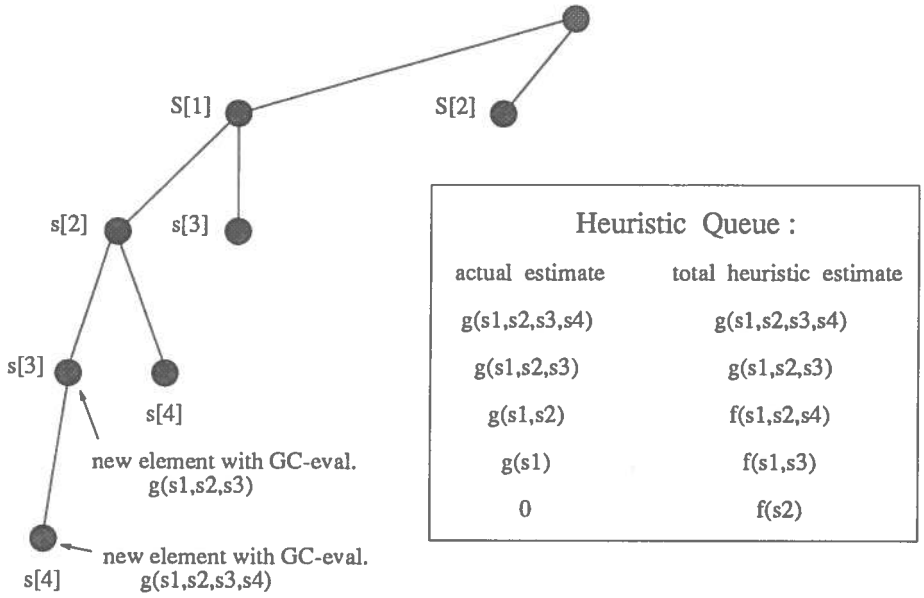

Figure 3–8: *Fourth step in exploring the search space*

1. abstraction
2. type-hierarchy
3. data-encapsulation
4. code-reuseability
5. parallelism

The whole algorithm can be broken down in relatively independent parts. Each part can be thought of as an instantiation of an object of a certain type. A class definition in $C++$ is a type definition. A big advantage of $C++$ is that one can define a inheritance tree very easily. The only difficult part is the decomposition of the problem in the right set of classes which are interrelated.

The whole algorithm can be represented as a set of classes. Each class is accessable from the outside world through a class member function which is in the public part of the class declaration. The private class declaration allows only access within that class. Similary the protected class declaration allows only access within the class hierarchy. For the algorithm following classes are of interest:

1. class tree

2. class PrioQueue

3. class Queue_element

4. class heap

5. struct PathNode

6. struct Pair

The best way to explain the implementation of the algorithm for the new model matcher is to start with its data structures. The most important data structure in the new algorithm is the priority queue. It is very important that the time complexity of the algorithm maintaining the queue is as low as possible. Sorting a linked list or array of elements is not satisfactory because sorting an arbitrary sequence of numbers needs $O(n \cdot \log n)$ time. Inserting an element in a sorted list takes $O(n)$ time. Because one incrementally inserts elements in the queue a sorted list can always be obtained in $O(n)$ time. This is not very efficient. There are ways to get around this problem but before that we must specify the problem precisely.

What criterions for the implementation of the priority queue need to be met?

1. queue length N

2. insert element

3. remove element with highest estimate

4. remove element with lowest estimate

One would like to remove the element with the lowest estimate in order to insert a new element in a completely filled queue. Thus the best candidate to eliminate is the element with the lowest estimate. It is obvious that one can only maintain a queue of a certain length but the point is to find the minimal length for the vision task to be solved. There is a strong relationship between performance and queue length.

An efficient implementation of the priority queue is possible using the heap data structure. In the standard algorithm found in the literature the remove operation (remove element with highest estimate) takes $O(1)$ and the insert operation takes only $O(\log n)$ time. Although efficient it doesn't fulfill all of the criteria mentioned above. In our case an additional problem need to be solved: the element with the lowest estimate need to be removed in the case when the priority queue is full.

The solution employed in the algorithm uses two communicating heap data structures. One keeps track of the maximum element and the other keeps track of the minimum element. To update the priority queue both data structures need to be maintained in parallel. While a bit more difficult it keeps the advantage of the heap data structure and satisfies all criteria stated above. As a matter of fact, the insert and remove operation takes only $O(\log n)$ time.

```
****************** class tree *********************************

 class tree {
  ChunkingVector<PlacedSurfacePair> S;
  const Assembly * assembly;
  int LAMBDA;
  PrioQueue<Queue_element> heuristic_queue;
  PrioQueue<Queue_element> geometric_consistency_queue;
  Queue_element best_queue_el;
  Queue_element next_queue_el1;
  Queue_element next_queue_el2;

  float geometric_consistency_evaluation(PathNodePtr);
  float heuristic_evaluation(PathNodePtr,float);
  float accumulate_evaluations(PathNodePtr node, GeometricEvaluator *);
  void create_pathnode(PathNodePtr&);
  void print_prio_queue(char *, char *, PrioQueue<Queue_element> &);
  void extract_relevant_placed_surfaces(PairVector& invoke_pairs);
public:
  tree(const Assembly*, PairVector& invoke_pairs);
  ~tree(){};
  void generate_descendents();
  void tree_expansion();
  void print_path(PathNodePtr);
  void print_queue_element(char*, Queue_element&);
  void print_heuristic_queue();
  void print_geometric_consistency_queue();
};
```

```
********************** class PrioQueue ****************************

#define PRIO_QUEUE_SIZE 131072

template<class T : NodeIndex>
class PrioQueue {
  int heapsize_max;
  int heapsize_min;
  T * a;
  T * b;
  heap<T> h;
  void insert_support(T&,int);
public:
  PrioQueue();
  ~PrioQueue() {};
  T remove();
  T find_max();
  void insert(T);
  int queuesize();
  T * queue_max();
  friend void printout(char *, char *, PrioQueue &, int);
};

********************** class Queue_element ****************************

class Queue_element : public NodeIndex{
       void clear();
public:
       int         select;
       float       estimate;
       float       heuristic_estimate;
       float       actual_estimate;
       PathNodePtr path;

       Queue_element(): NodeIndex() { clear(); };
       Queue_element(float est);
};

inline int operator<( const Queue_element x, const Queue_element y )
{
   return x.estimate < y.estimate;
}

inline int operator>( const Queue_element x, const Queue_element y )
{
   return x.estimate > y.estimate;
```

```
}


************************** class heap ********************************

struct BaseHeap {
  int parent(int i) { return i/2; }
  int left(int i) { return 2*i; }
  int right(int i) { return 2*i+1;}
};

struct NodeIndex{
  int partnerIndex;
    NodeIndex();
};

template<class T : NodeIndex>
class heap : public BaseHeap {
   int heapsize;
   int sel;
   T * c;
   T * d;

   void swap( T *, T *);
   void heapify(int);
public:
   heap(){};
   ~heap(){};
   void heapify_sel(int, int, int, T *, T *);
              // parameter: sel, index, heapsize, heap_array_a, heap_arra
   T heap_extract(int, int&,  T *, T *);
              // parameter: sel, heapsize, heap_array_a, heap_array_b
   void heap_insert(int, T, int, T *, T *);
              // parameter: sel, key, index, heap_array_a, heap_array_b
};


********************** struct PathNode ***************************

typedef struct PathNode  *PathNodePtr;
typedef struct Pair  *PairPtr;
struct PlacedSurfacePair;

struct PathNode{
     PathNodePtr  parent;
     PlacedSurfacePair * s;
     int          me;
```

```
    int          child_index;
    PathNode();
  int print() const;
  int length() const;
};

typedef struct SurfacePair * SurfacePair_P;
typedef struct SurfacePair ** SurfacePair_PvP;
typedef struct AssemblyPair * AssemblyPair_P;
typedef struct AssemblyPair ** AssemblyPair_PvP;


********************** struct Pair ************************************

struct Pair {
  enum Type {NONE, SURFACE, ASSEMBLY} type;
  float invoked_plausibility;

  Pair(Type t, float p):type(t),invoked_plausibility(p) {}

  virtual int   print() const;

public:
  static Pair * fread(File&);
};

struct SurfacePair : public Pair {
  int modelindex;
  Surface * model;
  DataSurface * data;

  SurfacePair(Surface * m, int i, DataSurface * d, float plaus):
    Pair(SURFACE, plaus),
    model(m),modelindex(i),data(d) {}
  int   print() const;

static ChunkingVector<Surface*> all_surfaces;
static Surface * surface(int index) { return all_surfaces[index]; }
static int numsurfaces() { return all_surfaces.last(); }
};

struct AssemblyPair : public Pair {
  ViewGroup* model;
  RichContext* data;

  AssemblyPair(ViewGroup * m, RichContext * d, float plaus):
    Pair(ASSEMBLY, plaus),
```

```
    model(m),data(d) {}

  int   print() const;

};

AssemblyPair * isAssemblyPair(Pair * p);
SurfacePair * isSurfacePair(Pair * p);

struct PairVector : ExpandingScalarVec<Pair*> {
  int load(const char *);
  int save(const char *) const;
};

**************************************************************************
```

Chapter 4

Experiments

4.1 Introduction

The new matching algorithm is tested on real data. One test-object is Widget the other one is the so called Renault part. The second test-object is significantly more complex than the first one and is a big challenge for the new algorithm. The goal of the experiment is to determine the influence of parameters to the overall performance of the algorithm. Both parameters and performance have to be defined first. Which parameters are used and how to measure the performance of the algorithm will be explained in this section. An assumption made is that the algorithm can find the correct hypothesis.

Because the algorithm is a best first search algorithm the performance is clearly strongly related to the total heuristic estimate. The definition of the heuristic function is critical and must be set up very carefully. The range of the parameters used in the heuristic function must also be taken in account when considering the performance of the algorithm. The total heuristic function $f(s_n)$ consists of two parts:

1. geometric consistency or actual estimate $g(s_{n-1})$

2. heuristic estimate $h(s_n)$

In the current implementation the total heuristic function uses an additive operator to combine the geometric consistency function and the heuristic function. Thus the total heuristic function is:

$$f(\{s_1, \ldots, s_n\}) = g(\{s_1, \ldots, s_{n-1}\}) + h(s_n)$$

In the experiment two variants of total heuristic functions have been used. In order to describe it the geometric consistency function have to be explained first. The geometric consistency function is a function of three components:

$$g(p_i) = g(g_{ang}(p_i), g_{dist}(p_i), g_{proj}(p_i))$$

where p_i is the path of the current node i. The three components are consistency functions for the binary angle, distance and projection constraints. Each consistency function computes the estimate of each pair of surface pairs and combines it according to the following formula:

$$\prod_i \prod_{j>i} r(s_i, s_j)$$

where s_j is the surface pair of node j and s_i is the surface pair of node i and $r(s_i, s_j)$ is the geometric consistency estimate of the pair of surface pairs. For each pair of surface pairs the consistency function uses the corresponding geometric consistency evaluation function. The angle consistency function uses the angle consistency evaluation function, the distance consistency function the distance consistency evaluation function and the projection consistency function the projection consistency evaluation function. In the following each of these evaluation functions will be considered separately.

Each evaluation function is a binary consistency test applied to planar and biquadratic surfaces. In the current implementation the geometric consistency test for biquadratic surfaces is simply a dummy test. It always returns one. Any pair of surface pairs containing nonplanar data surface patches is regarded to be consistent.

The input to the geometric consistency evaluation function, considered as a black box, is a pair of surface pairs. The function uses the model and data interval table to calculate an estimate of the plausibility of the consistency of the pair of surface pairs. The model interval table is calculated after the model has been built and is available for future recognition tasks. The data interval table is calculated after the range image has been aquired and is delivered by the segmentation modul.

The angle consistency evaluation function compares model and data angle intervals and returns an estimate how good that comparison was by using a gaussian

function. It retrieve the data and model interval from the precalculated data and model interval table and calls the function `eval_gaussian()` (see program code).

```
float eval_gaussian( double dmin, double dmax, double mmin, double mmax )
{
  double data_mean = (dmax + dmin)*0.5;
  double data_width = dmax - dmin;
  double model_mean = (mmax + mmin)*0.5;
  double sigma = 3*data_width;      //  3 is fudge factor
  double x = (data_mean - model_mean)/sigma;
  double f = exp(-x*x);

  return f;
}
```

The angle consistency test employs a gaussian function in order to calculate an estimate of the plausibility of consistent surface pairs. The function simply calculates the mean value of the data and model interval and the difference of the two values is normalized by the standard deviation of the gaussian function. The standard deviation determines the width of the gaussian curve and for constant mean values of data and model angles it determines the probability to accept a pair of surface pairs. The larger sigma the broader the gaussian curve and the higher the probability to accept a pair and vice versa. Low sigma means therefore more discriminate. The standard deviation is determined by the data interval and a fudge factor which must be verified or adapted through experimentation. In our case a fudge factor of 3 has been chosen. The larger the fudge factor or data interval the larger the standard deviation and the broader the gaussian curve and vice versa. A broader gaussian curve means less discriminate and less reluctant to possible matches. This factor is one of many parameters which determine the overall performance of the algorithm and should be kept in mind. There are two other cases where the behaviour of the gaussian function evaluation with respect to varying parameters can be studied. In the first one the mean of the data and model interval as well as the fudge factor are kept constant. Increasing the data interval increases also the standard deviation of the gaussian function and the final estimate and vice versa. In the second case the data interval and the fudge factor are kept constant. Increasing the difference between mean of data interval and mean of model interval decreases the consistency estimate of the surface pair in question.

The distance consistency evaluation function compares maximal and minimal distances of model and data surface patches and uses as criterion either overlapping or insertion to calculate an estimate. The function retrieves the model and data intervals from the interval tables previously calculated and calls the function `eval_interval_overlap` (see program code).

```
float eval_interval_overlap( double dmin, double dmax, double mmin, doubl
{
  double x;
  double f;
  double sigma = 3;      // fudge-factor; have to be determined experiment

  if (dmax > mmin && dmin < mmax)
      f = 1;
  else if (dmax < mmin)
     { x = (mmin-dmax)/sigma;
       f = exp(-x*x);
     }
  else if (dmin > mmax)
     { x = (dmin-mmax)/sigma;
       f = exp(-x*x);
     };

  return f;
}
```

It should be emphasized that an important design decision must be made here. When do we take a function based on overlapping and when a function based on insertion? For overlapping three cases need to be considered.

1. data interval is left of the model interval

2. data interval is overlapping (classical case)

3. data interval is right of the model interval

All three cases can occur due to occlusion, noise and segmentation error. For an example see Figure 4–1.

A overlap-function which only accepts the case where the two intervals are strictly overlapping rejects the other two cases. Thus the threshold function isn't

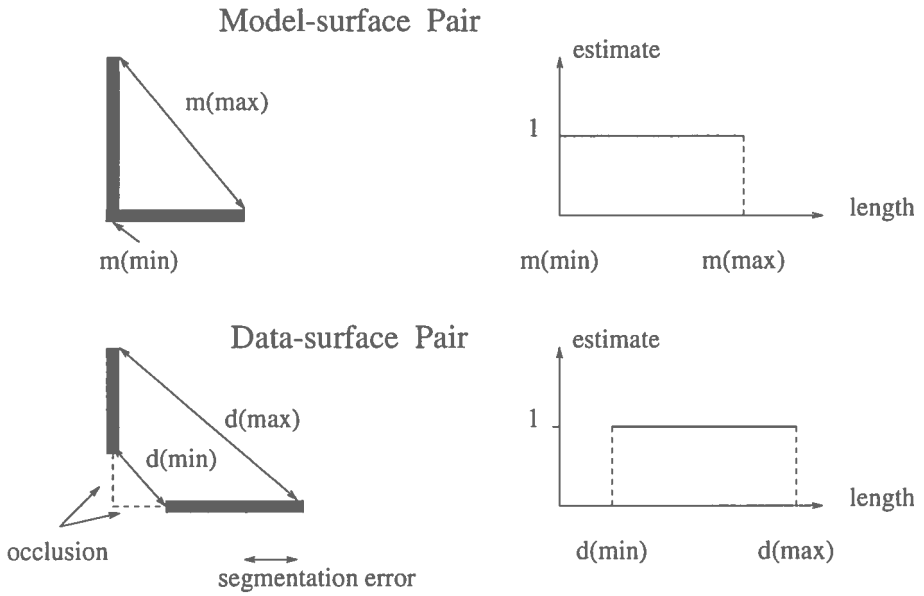

Figure 4–1: *Overlap-function*

appropriate. Smoothing the left and right edges of the estimation function (see Figure 4–1 right) avoids the threshold effect and includes the other two cases to some extent with decreasing plausibility values.

For insertion two main cases need to be considered:

1. data interval within model interval

2. data interval enclosing model interval

As above this two cases can occur due to occlusion and segmentation error. For an example see Figure 4–2. The two parallel planes are larger after segmentation and therefore the data interval encloses the model interval.

If one tries to smooth the right and left edge of the threshold estimation function then one could include cases where the data interval can overlap either the right or left end of the model interval with decreasing plausibility values. This is reasonable to counteract the effects of occlusion and segmentation error.

For both cases the overlap and insertion function there is a tradeoff between discrimination and avoiding occclusion and segmentation error. In the experiment the overlap-function has been chosen for both the distance and the projection

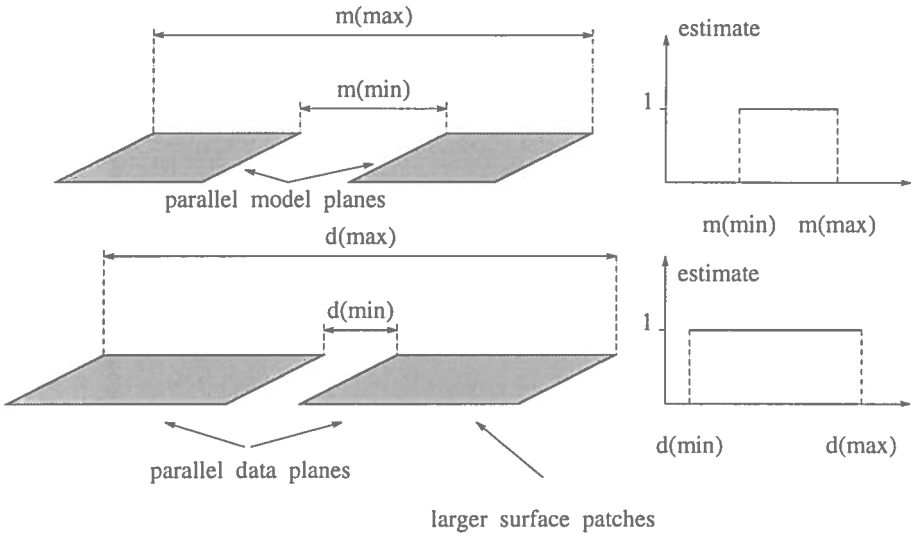

Figure 4–2: *Inclusion-function*

consistency evaluation function. The experience with the current model matcher has been taken to justify the decision.

Now we are in position to state the total heuristic functions. As mentioned above there are two functions. The first one is:

$$\begin{aligned} f(s_n) &= \{\alpha_1 \cdot \log(g_{ang}(s_{i-1})) + \alpha_2 \cdot \log(g_{dist}(s_{i-1})) + \alpha_3 \cdot \log(g_{proj}(s_{i-1})) + \\ &\quad \alpha_4 \cdot g(s_{i-1}) + \frac{c}{h_{max} - h_{min}} \cdot (\gamma \cdot \log(s_n) + \beta_1 \cdot n^{\beta_2} - h_{max})\} \cdot \frac{1}{2} \end{aligned} \quad (4.1)$$

and the second one is:

$$\begin{aligned} f(s_n) &= \{g_{ang}(s_{i-1})^{\alpha_1} \cdot g_{dist}(s_{i-1})^{\alpha_2} \cdot g_{proj}(s_{i-1})^{\alpha_3} \cdot g(s_{i-1})^{\alpha_4} + \\ &\quad \frac{1}{h_{max} - h_{min}} \cdot (\gamma \cdot \log(s_n) + \beta_1 \cdot n^{\beta_2} - h_{min})\} \cdot \frac{1}{2} \end{aligned} \quad (4.2)$$

with the following constants:

$$h_{max} = \gamma \cdot \log(s_{max}) + \beta_1 \cdot \lambda^{\beta_2} \quad (4.3)$$

$$h_{min} = \gamma \cdot \log(s_{min}) + \beta_1 \quad (4.4)$$

In the formulae 4.1 and 4.2 the logarithmic function is limited to a negative constant c to avoid shooting off to minus infinity for very small argument values.

Formula 4.1 is normalized to a range of negative constant c to zero. Formula 4.2 is normalized to a range of zero to one. It is important that both the geometric consistency and the heuristic term have the same range in order to avoid that either geometric consistency or heuristic evaluated nodes are favourized in the priority queue. The normalization parameters h_{max} and h_{min} is the maximum respective minimum value of the unnormalixed heuristic term whereas s_{max} and s_{min} is the maximal respective minimal value of the invocation plausibility values. The longest path or equally the number of invocation plausibility values is denoted with λ. Notice the actual or geometric consistency estimate $g(s_{i-1})$ of the parent node in both formulae. Notice also that the start value in formula 4.1 is $\log(g(s_1))$ and in formula 4.2 $g(s_1)$.

The parameters of the total heuristic estimate are:

$$\alpha_1 \quad \alpha_2 \quad \alpha_3 \quad \alpha_4 \quad \beta_1 \quad \beta_2 \quad \gamma$$

Parameters which measure the performance of the overall algorithm are:

1. number of node expansion
2. number of geometric consistency evaluation
3. number of raytrace operation
4. number of elements inserted in the priority queue
5. minimal, maximal and average throughput of the priority queue
6. maximal, minimal and average priority queue length
7. maximal and average path length
8. path length distribution
9. priority queue length distribution
10. differential path length distribution

11. differential queue length distribution

12. lowest and highest position of correct hypotheses in priority queue

13. position distribution of the correct hypotheses in priority queue

14. differential of position distribution of the correct hypotheses in priority queue

For the experiment the first three have been chosen because they are related to the time and space complexity of the algorithm. The fourth point is a value which is not independent and is calculated by adding the values of the first two points.

4.2 Method

The criterion to terminate the algorithm is that the node on top of the priority queue is geometric consistency evaluated and is the correct hypotheses. Who determines the correct hypotheses? It is the responsibility of the raytracer to find acceptable hypotheses. It does this by first estimating the pose and then comparing data and model matches after projection of the oriented model on the range image. To summarize the raytracer gets as input a geometric consistency evaluated node and calculates an estimate of the probability that the current hypothesis is the correct one.

In the experiment a dummy raytracer has been used instead of a real one. This is not a disadvantage with respect to analysing the behaviour of the algorithm. All the parameters stated above can be analysed independently. The only necessity is that a human determine the correct data model pairings which are then used in the program as a knowledge to filter out the correct hypotheses.

The range of parameters in formulae 4.1 and 4.2 are:

parameters	ranges
α_1	$0 \ldots 1$
α_2	$0 \ldots 1$
α_3	$0 \ldots 1$
α_4	$0 \ldots 1$
β_1	$0.1 \ldots 5$
β_2	$0.1 \ldots 5$
γ	$0.1 \ldots 5$

with the assumption for formula 4.1:

$$\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 = 1$$

4.3 Results

The first experiment was done with the Widget object. The model of the Widget object can be seen in Figure 4–3 and the data can be seen from the range image in Figure 4–4 and 4–7. The input to the model matcher are the invocation plausibility values supplied by the invocation modul. Fourteen data model surface pairs are the input to the algorithm. Thus the maximal pathlength of the search is fourteen. The two heuristic function 4.1 and 4.2 have been used to test the performance.

Notation:

NE . . . Number of node expansion

GV . . . Number of geometric consistency evaluation

RT . . . Number of raytracing operation

L_Q . . . Length of priority queue after solution found

The results for the total heuristic function 4.1 are:

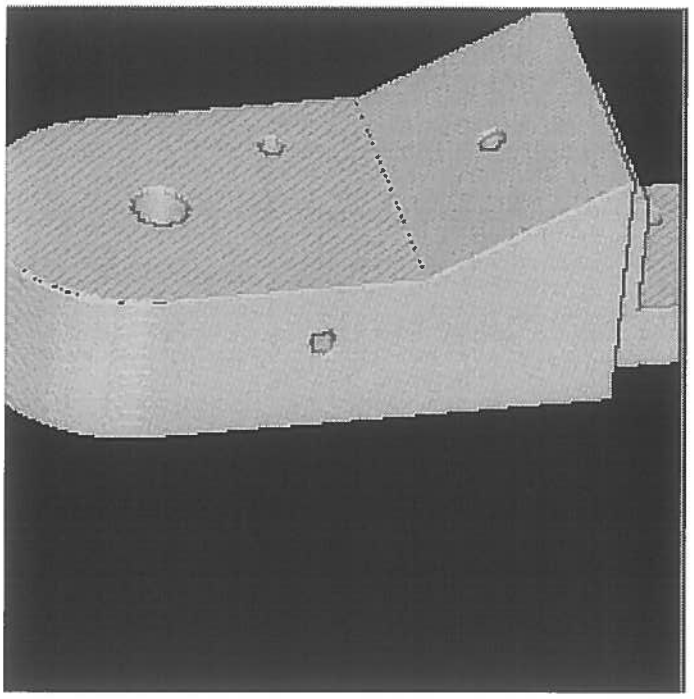

Figure 4–3: *Model of the Widget Object*

α_1	α_2	α_3	α_4	β_1	β_2	γ	NE	GV	RT	L_Q
$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	1	1	1	83	11	7	40
				1	1	2	111	13	8	54
				1	1	0.5	111	13	8	54
				1	2	1	111	13	8	54
				1	0.5	1	397	45	30	191
				2	2	1	83	11	7	40
				0.5	2	1	111	13	8	54

The results for the total heuristic function 4.2 are:

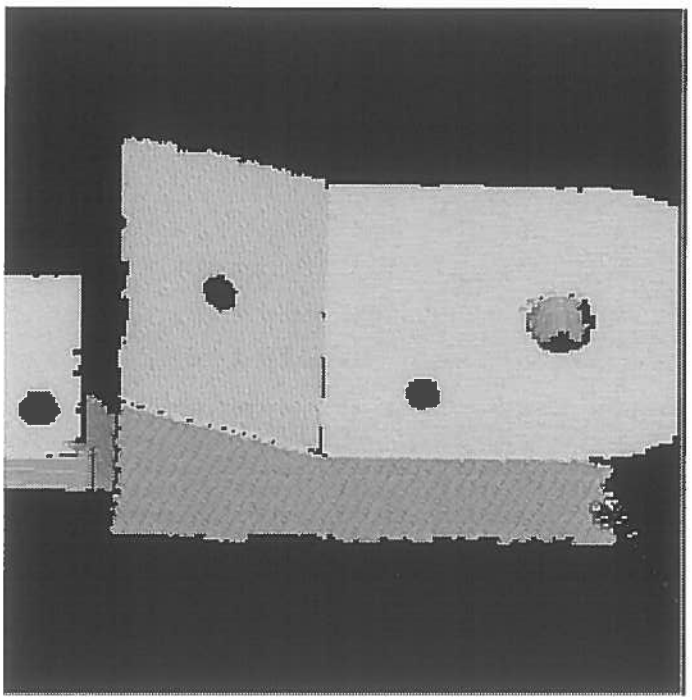

Figure 4–4: *Range Image of the Widget Object*

α_1	α_2	α_3	α_4	β_1	β_2	γ	NE	GV	RT	L_Q
1	1	1	1	1	1	1	47	4	1	25
				1	1	2	83	8	3	43
				1	1	0.5	47	4	1	25
				1	2	1	51	4	1	27
				1	0.5	1	273	16	7	138
				2	1	1	47	4	1	25
				0.5	1	1	83	8	3	43

The second experiment was done with the Renault part. The model of the Renault part can be seen in Figure 4–5 and the object can be seen in the range image in Figure 4–6. The model has been aquired from the range data of the object. The Renault Part is considerable more complex than the Widget object and the invocation hypotheses list comprises hundred data model surface pairings. In the actual run of the program the algorithm couldn't find a solution. The problem has been simplified by excluding biquadratic model surfaces from the invocation list. The invocation list has been reduced from hundred to 42. Despite the changes

made the algorithm failed to find the correct hypotheses. The reason for the failure is founded in bad heuristic estimates and not the algorithm itself.

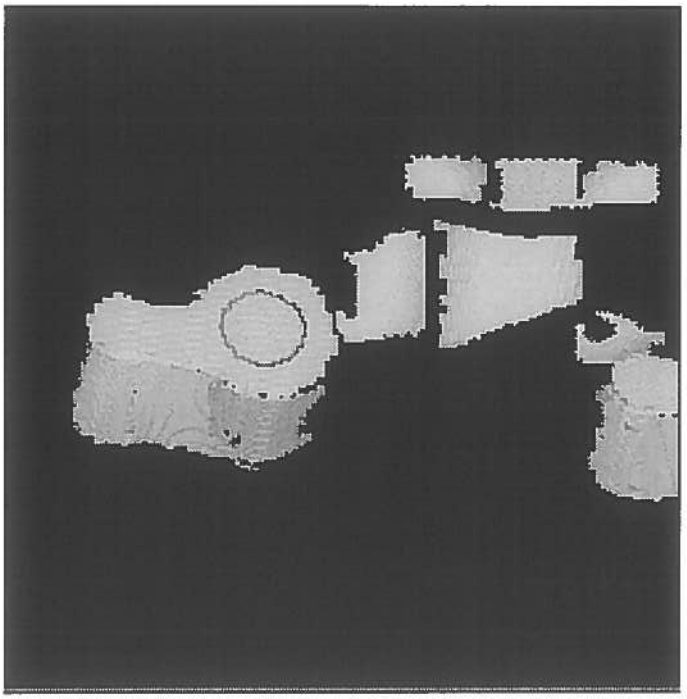

Figure 4–5: *Model of Renault Part*

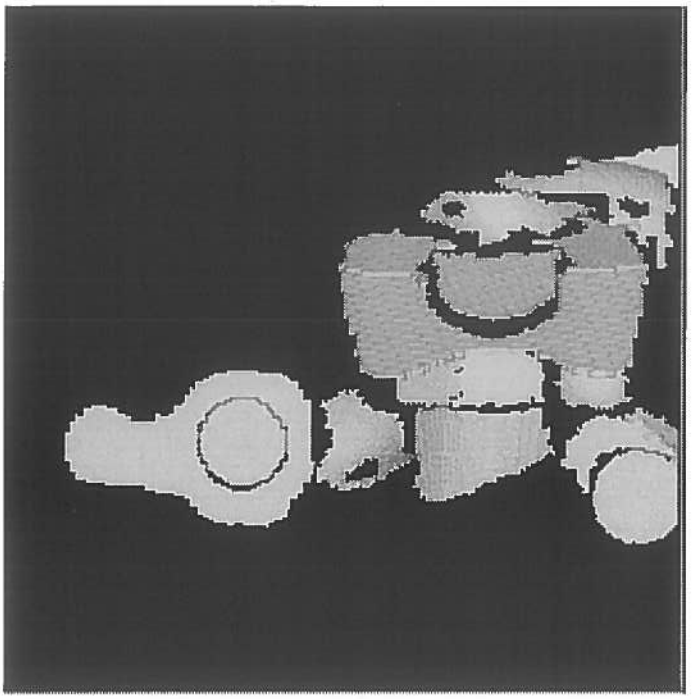

Figure 4–6: *Data of Renault Part*

Descriptions File: ../ren315/data.des

	1	2	3	5	6	11	12
2	58						
3	57	1					
5	0	58	57				
6	58	1	1	58			
11	6	63	63	6	64		
12	5	63	62	5	64	2	
14	58	0	1	58	1	63	63

Plane D=467.741	1	2968	[-0.01 0.85 -0.53]
Plane D=936.998	2	1564	[0.00 0.01 -1.00]
Plane D=938.655	3	559	[-0.00 0.01 -1.00]
Biquad(24,84304)	4	973	[0.54 -0.19 0.08]
Plane D=462.414	5	1084	[-0.01 0.85 -0.53]
Plane D=937.013	6	565	[-0.01 -0.01 -1.00]
Biquad(-27,-12)	7	400	[0.03 0.05 0.91]
Biquad(-57,0)	8	326	[-0.45 0.65 -1.43]
Biquad(-27,-12)	9	526	[-0.17 -0.43 -0.33]
Biquad(634,-40)	10	304	[-0.01 0.01 -0.20]
Plane D=388.29	11	641	[-0.01 0.90 -0.44]
Plane D=395.647	12	607	[-0.04 0.89 -0.45]
Biquad(105,-69)	13	383	[0.01 -0.00 1.49]
Plane D=959.394	14	206	[-0.00 0.00 -1.00]
Biquad(28,-303)	15	245	[0.47 0.38 -1.17]
Biquad(62,-75)	16	195	[-0.50 0.78 -0.42]
Biquad(13,19)	17	215	[0.42 -0.07 0.63]

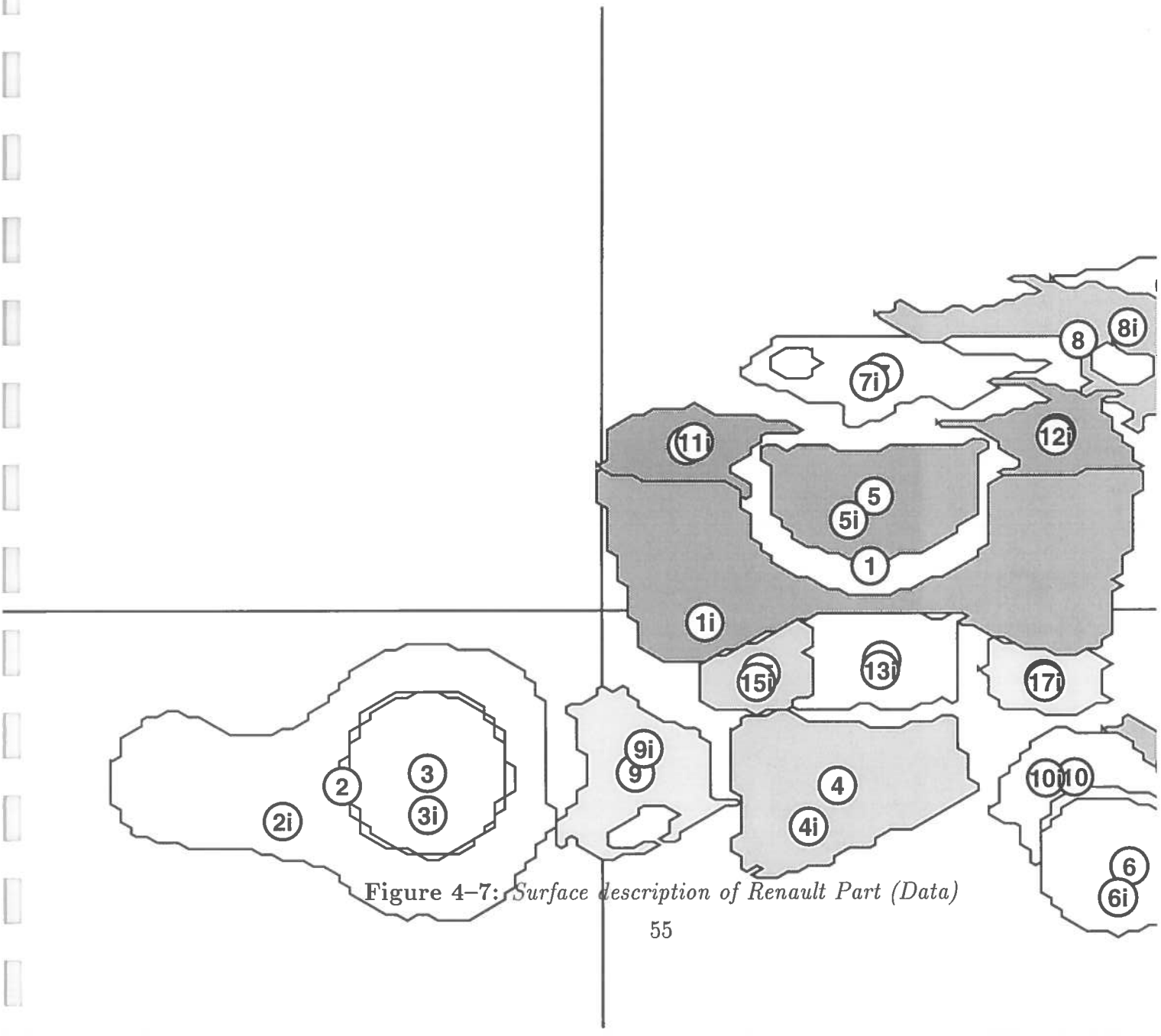

Figure 4–7: *Surface description of Renault Part (Data)*

Chapter 5

Conclusion

5.1 Summary

The new algorithm has been successfully implemented and incorporated into the IMAGINE II environment. The Widget object has been used to analyse the performance of the algorithm. One assumption made in this experiment was that the correct solution can be found. Two heuristic functions have been designed and used to compare the performance of the algorithm. One key result is that the heuristic function which uses parameters in the exponential domain works better that the one in the logarithmic domain. It only needs one raytrace operation to find the solution. It is also interesting to observe that the candidate with the minimal number of raytraces is also the one which has the minimal number of node expansion, geometric consistency evaluations and the minimal priority queue length. Further investigation of the influence of the alpha parameters $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ with respect to the performance of the algorithm was not done because the solution was already found in the first raytrace operation. The next natural step was to use a more complex object to investigate the performance of the algorithm.

In the second experiment the Renault part, known as one of the most complex parts in the vision literature, has been used. There were hundred data surface pairs that the model matcher has had to deal with. As a result the new model matcher hasn't found the solution at all. In order to reduce the complexity of the matching task all biquadratic model surface pairings have been removed from the input file. This reduced the complexity to 42 data model pairs. As a result the algorithm couldn't find the solution either. Analyses of the explored search tree showed that the algorithm explored first the tree in breadth-first order and then

dove deep in the search tree before coming back to do some search on this level and dove again deep in the search tree. During that time he hasn't generated the node in the tree with the correct solution. The algorithm explored the tree like a fork. It also reaches the maximal possible depth of the search tree.

5.2 Future Work

As stated several times the critical point in best-first search is the heuristic function to direct the search. As the Renault part has nicely demonstrated an inadequate heuristic function can lead to the problem of combinatorial explosion. An important point is what strategy ought to be chosen to cope with the search complexity. Additional constraints could help to prune the interpretation tree more efficiently. One possibility is to eliminate duplicates of data and model pairings along the path of the node in question. The idea is e.g. that two data patches which matches the same model patch must have about the same curvature and area as the model patch to get an invocation plausibility. It is therefore wasted search effort to investigate both patches at the same path. Imagine two data planes separated a certain distance from each other lying about in the model plane. The first data plane supply a constraint and the addition of the second one doesn't give an additional constraint and can simply be ignored. If the first plane seems to be inconsistent then there is still a way to match the second one.

In designing the heuristic function it has been observed that a statistical balance of f and g values in the priority queue is significant. If g values are dominant than they are located on top of the queue and therefore blocking the algorithm from exploring further open nodes. If on the other hand the f values are dominant than they are located on top of the queue and are explored as long as there are no open nodes left therefore ignoring the leaf nodes of the tree completely. The normalization step in designing the heuristic function prevents the dominance of either of the terms in the priority queue.

A simple measurement can be used to measure the frequency of f and g values in the priority queue. For a given queue length one counts from the top the number of g or f values as a function of the the relative distance from the top.

Differentiating the distribution of g or f values with respect to the relative queue position supplies the frequency of g or f nodes in the priority queue.

In the discussion of the experiments a large list of parameters measuring the performance of the algorithm has been given. The different ways performance can be analysed can help to extract the important parameters and ranges.

The new model matcher hasn't incorporated the pose estimation constraint. Future implementation of the algorithm should incorporate it, with an expected improvement in performance.

Including biquadratic surfaces in the constraint evaluation process is an important step towards a general model matcher.

To justify the new algorithm a comparison with the current standard interpretation tree search is necessary. Comparison of run time with different objects is one possibility, but depends on both implementations being optimally coded.

Bibliography

[Ayache & Faugeras 86] N. Ayache and O.D. Faugeras. Hyper: A new approach for the recognition and positioning of two-dimensional objects. In *IEEE Trans. Patt. Anal. and Mach. Intell.*, pages 44–54, 1986.

[Fisher 89] R. B. Fisher. *From Surfaces to Objects: Computer Vision and Three Dimensional Scene Analysis.* John Wiley and Sons, Chichester, 1989.

[Fisher *et al.* 93] R. B. Fisher, A. W. Fitzgibbon, M. Waite, E. Trucco, and M. J. L. Orr. *Recognition of Complex 3-D Objects from Range Data.* DAI Research Paper, Dept Artificial Intelligence, University of Edinburgh, September 1993.

[Grimson & Lozano-Perez 87] Grimson and Lozano-Perez. Localizing overlapping parts by searching the interpretation tree. In *IEEE Trans. Patt. Anal. and Mach. Intell.*, pages 469–482, 1987.

[Grimson 90] W. Eric L. Grimson. *Object Recognition by Computer: The Role of Constraints.* MIT Press, Cambridge, Massachusetts, 1990.

Appendix A

Program code for New Model Matcher

```
% File:
% Author:      HEBENSTREIT
% Updated:     SEPTEMBER 12 1994
% Purpose:     NEW MODEL MATCHER


#include "libgr/geometry.hxx"
#include "libu2/ChunkingVector.h"
#include "PlacedSurfacePair.h"
#include "queue_element.h"
#include "prio_queue.h"

//     Here's the class to build up the Interpretation Tree

struct PairVector;
struct GeometricEvaluator;

class tree {
  ChunkingVector<PlacedSurfacePair> S;
  const Assembly * assembly;
  int LAMBDA;
  PrioQueue<Queue_element> heuristic_queue;
  PrioQueue<Queue_element> geometric_consistency_queue;
  Queue_element best_queue_el;
  Queue_element next_queue_el1;
  Queue_element next_queue_el2;
  int count1;
  int count2;
  int count3;
  float h_max;
  float h_min;
  float beta1;
  float beta2;
  float gamma;
```

```
  void init_par_heurist_eval();
  float geometric_consistency_evaluation(PathNodePtr);
  float heuristic_evaluation(PathNodePtr,float);
  float accumulate_evaluations(PathNodePtr node, GeometricEvaluator *);
  int  ray_trac_sim(Queue_element&);
  void create_pathnode(PathNodePtr&);
  void print_prio_queue(char *, char *, PrioQueue<Queue_element> &);
  void extract_relevant_placed_surfaces(PairVector& invoke_pairs);
public:
  tree(const Assembly*, PairVector& invoke_pairs);
  ~tree(){};
  int num_node_exp() { count1+=2; };
  int num_gv() { count2++; };
  int num_raytrace() { count3++; };
  int read_num_node_exp() { return count1; };
  int read_num_gv() { return count2; };
  int read_num_raytrace() { return count3; };

  void generate_descendents();
  void tree_expansion();
  void print_path(PathNodePtr);
  void print_queue_element(char*, Queue_element&);
  void print_heuristic_queue();
  void print_geometric_consistency_queue();
};
#pragma ident "%Z%%M% %I% %E%"

//
//        Robot Vision Group
//  Dept. of Artificial Intelligence
//      University of Edinburgh
//
// Author: Josef Hebenstreit
// Date: Summer 94
// Description:
//

#include <stream.h>
#include <iomanip.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>

#include "libu/messages.hxx"
#include "tree.h"
#include "PathNode.h"

//=============================================================================


tree::tree(const Assembly * a, PairVector& invoke_pairs)
{
```

```
        count1 = 1;    // Only for statistics !
                       // counter: number of node expansion

        count2 = 0;    // Only for statistics !
                       // counter: number of geometric evaluation

        count3 = 0;    // Only for statistics !
                       // counter: number of ray-trace operations


        assembly = a;
        extract_relevant_placed_surfaces(invoke_pairs);

        init_par_heurist_eval();  // initialize parameter for heuristic evaluation


        ::info("LAMBDA =  %d \n",LAMBDA);  // test


              // make seed for random number generation dependent from time
        srand(time(0));
                 // create root node
        PathNodePtr root = new PathNode;
        root->child_index = 1;
                 // create first descendents
        PathNodePtr first = new PathNode;
                 // initialize first descendents
        first->parent = root;
        first->me = 1;
        first->s = &S[1];
        first->child_index = 2;
                 // set child index of root node
        root->child_index += 1;

        Queue_element key;
                 // fill queue-element for first descendents
        key.select = 0;     //  indicate that estimate = heuristic_estimate
        key.estimate = S[first->me].invoked_plausibility;
        key.heuristic_estimate = key.estimate;
        key.path = first;
                 // insert first descendents in queue
        heuristic_queue.insert(key);

        print_queue_element("first element:",key);
};

extern float clog(float);

void tree::init_par_heurist_eval()
{
        beta1 = 1;
        beta2 = 1;
```

```
     gamma = 1;

     h_max = beta1 * pow(LAMBDA,beta2)\
             + gamma*clog(S[1].invoked_plausibility);
     h_min = beta1 + gamma*clog(S[LAMBDA].invoked_plausibility);
};


float tree::heuristic_evaluation(PathNodePtr path, float actual_estimate)
{
     float h;

     h = ( actual_estimate + (beta1*pow(path.length(),beta2) \
                     + gamma*clog(S[path->me].invoked_plausibility) - h_min)

     return h;
};

void tree::tree_expansion()
{
 int correct_hyp;
 do {
//
    while (heuristic_queue.find_max().select == 0)
      {
        generate_descendents();

 ray_trac_sim(heuristic_queue.find_max());

        if ( heuristic_queue.queuesize() == 32000)      // test
   {
              printf("generate descendents stop \n");
              printf("node-exp =  %d \n",read_num_node_exp());
              printf("geometric-consist-eval =  %d \n",read_num_gv());
              printf("heuristic-queue-length =  %d \n",heuristic_queue.queuesi
       // print_heuristic_queue();
       //print_geometric_consistency_queue();
             exit(1);
          };

      };

    correct_hyp = ray_trac_sim(heuristic_queue.find_max());
    if (correct_hyp == 0)
      { num_raytrace();
        geometric_consistency_queue.insert(heuristic_queue.remove());
      };

 } while ((heuristic_queue.queuesize() != 0) && (correct_hyp == 0));

 ::info("\n Solution Path:  \n");
   // heuristic_queue.find_max().path->print();
```

```
 exit(1);

};

                        // dummy raytracer widget

int tree::ray_trac_sim(Queue_element& gv)
{
   PathNodePtr current_node = gv.path;
   if ( (current_node->length() == 3) && (current_node->me == 5) )
      { current_node = current_node->parent;
        if (current_node->me == 2)
           {  current_node = current_node->parent;
             if ( (current_node->me == 1) && (current_node->parent->parent == 0
                {
         ::info(" ray-trac stop \n");
                 ::info("node-exp =  %d \n",read_num_node_exp());
                 ::info("geometric-consist-eval =  %d \n",read_num_gv());
                 ::info("raytrace =  %d \n",read_num_raytrace());
                 ::info("heuristic-queue-length =  %d \n",heuristic_queue.queue
                 ::info("geometric_consistency_queue-length =  %d \n",geometric
       // print_heuristic_queue();
       //       print_geometric_consistency_queue();
                 exit(1);
                 return 1;
                };
   };
      };

   return 0;
};


               // dummy raytracer renault part
/*
int tree::ray_trac_sim(Queue_element& gv)
{
   PathNodePtr current_node = gv.path;
   if ( (current_node->length() == 4) && (current_node->me == 14) )
      { current_node = current_node->parent;
        if (current_node->me == 8)
           { current_node = current_node->parent;
             if (current_node->me == 7)
                { current_node = current_node->parent;
                  if ( (current_node->me == 1) && \
                                  (current_node->parent->parent == 0) )
                     {
              ::info(" ray-trac stop \n");
                      ::info("node-exp =  %d \n",read_num_node_exp());
                      ::info("geometric-consist-eval =  %d \n",read_num_gv());
                      ::info("raytrace =  %d \n",read_num_raytrace());
```

```
                    ::info("heuristic-queue-length =  %d \n",heuristic_queue.
                    ::info("geometric_consistency_queue-length =  %d \n",geom
              //   print_heuristic_queue();
              //       print_geometric_consistency_queue();
                    exit(1);
                    return 1;
     };
          };
   };
      };
   return 0;
};
*/

void tree::generate_descendents()
{
                  // remove path with best heuristic estimate
           best_queue_el = heuristic_queue.remove();
           //print_queue_element("remove element:",best_queue_el);

   // ********************************************************************

                    // insert first queue-element in queue
                   // with heuristic estimate
           PathNodePtr path = best_queue_el.path;

                  // Important: Geometric Consistency Evaluation here !!
           if (path->parent->parent != 0)
                 next_queue_el1.actual_estimate = geometric_consistency_evalua
           else
                 next_queue_el1.actual_estimate = best_queue_el.estimate;
           if ( path->child_index <= LAMBDA )
              {
            // statistic: count node expansion ( notice:
           // per expansion 2 nodes )
              num_node_exp();
              next_queue_el1.select = 0;
              create_pathnode(path);
              next_queue_el1.path = path;
                    // heuristic evaluation of the path
              next_queue_el1.estimate = heuristic_evaluation(path,next_queue
              }
           else
              {            // leaf-node reached;
              next_queue_el1.select = 1;
              next_queue_el1.estimate = next_queue_el1.actual_estimate;
              next_queue_el1.path = path;
            // statistic: count nodes with geometric
           // consistency evaluation
              num_gv();
              };
                       // Insert element in Heuristic-Queue
```

```
            heuristic_queue.insert(next_queue_el1);

     print_queue_element("first descendent:",next_queue_el1);

    // ************************************************************************
                    // insert second queue-element in queue
                      // with heuristic estimate
            path = best_queue_el.path->parent;
            if ( path->child_index <= LAMBDA )
               {
                 next_queue_el2.select = 0;
                 create_pathnode(path);
                 next_queue_el2.path = path;
                 next_queue_el2.actual_estimate = best_queue_el.actual_estimate
                 if ( path->parent->parent == 0 )
                      next_queue_el2.estimate = S[path->me].invoked_plausibilit
                 else
                           // heuristic evaluation of the path
                     next_queue_el2.estimate = heuristic_evaluation(path,next_q
                         // Insert element in Heuristic-Queue
                 heuristic_queue.insert(next_queue_el2);

 print_queue_element("second descendent:",next_queue_el2);

               }
            else if ( path->parent != 0 )
       {                 // parent of best-queue-element has no
              // further children to be explored !!
                 next_queue_el2.select = 1;
                 next_queue_el2.path = path;
                 next_queue_el2.actual_estimate = best_queue_el.actual_estimate
                 next_queue_el2.estimate = next_queue_el2.actual_estimate;
                         // Insert element in Heuristic-Queue
                 heuristic_queue.insert(next_queue_el2);

 print_queue_element("second descendent:",next_queue_el2);

             // statistic: count nodes with geometric
            // consistency evaluation
                 num_gv();

               };
            // print_queue_element("second descendent:",next_queue_el2);
            //print_heuristic_queue();
};


/*
void tree::print_path(PathNodePtr p)
{
  cout << "path:\n";
  while (p != 0)
```

```
    {
      cout << "S[" << p->me << "],  " << "child_index = " << p->child_index <<
      p = p->parent;
    }
  cout << "\n";
};
*/

void tree::print_path(PathNodePtr p)
{
  int s[LAMBDA+1];
  int n = 1;
  int count = 0;
  if (p==0) return;
  while (p->parent != 0)
    {
      s[n++] = p->me;
      p = p->parent;
      count++;
    };
  for (int i=count; i>0; i--)
    if (i>1)
      cout << "S[" << s[i] << "]" << ":";
    else
      cout << "S[" << s[i] << "]\n";
};


void tree::print_queue_element(char* s, Queue_element& q)
{
     cout << s << "\n";
     cout << "estimate = " << q.estimate << ",  "
          << "actual_estimate = " << q.actual_estimate << ", "
          << "select = " << q.select << "\n";
     print_path(q.path);
};



void tree::print_heuristic_queue()
{
  print_prio_queue("Heuristic-queue:  ", "H", heuristic_queue);
};



void tree::print_geometric_consistency_queue()
{
  print_prio_queue("Geometric-Consistency-Queue:  ", "GC", geometric_consistenc
};

void tree::create_pathnode( PathNodePtr& path )
```

```
{
                    // create path-node
            PathNodePtr new_path = new PathNode;
            new_path->parent = path;
            new_path->me = path->child_index;
    new_path->s = &S[new_path->me];
            new_path->child_index = new_path->me + 1;
            path->child_index += 1;
            path = new_path;
};


void tree::print_prio_queue(char * str1, char * str2, PrioQueue<Queue_element>
{
   Queue_element * c = q.queue_max();      //  import Max-Queue
   int queuesize = q.queuesize();         //   import Queuesize
   PathNodePtr p;

   cout << str1 << "\n";                        //  print header
   cout.setf(ios::left,ios::adjustfield);    //  adjust output
   for ( int i=1; i <= queuesize; i++)
   {
      cout << str2 << "[" << i << "] :  path: ";
      p = c[i].path;
      while (p != 0)
         {
           cout << "S[" << p->me << "]";
           p = p->parent;
           if (p != 0)
              cout << " --> ";
          };
      cout << "\n";

      cout << "\t"
           << "estimate = " << setw(10) << setprecision(5)
           << c[i].estimate << ",  "
           << "actual_estimate = " << setw(10) << c[i].actual_estimate << ",  "
           << "select = " << c[i].select << "\n";
      cout << "\n";
   };
};
//                prio_queue.h

#ifndef __prio_queue_h_
#define __prio_queue_h_

#include "heap.h"

#pragma interface

#define PRIO_QUEUE_SIZE 131072
```

```
template<class T : NodeIndex>
class PrioQueue {
  int heapsize_max;
  int heapsize_min;
  T * a;
  T * b;
  heap<T> h;
  void insert_support(T&,int);
public:
  PrioQueue();
  ~PrioQueue() {};
  T remove();
  T find_max();
  void insert(T);
  int queuesize();
  T * queue_max();
  friend void printout(char *, char *, PrioQueue &, int);
};

#include "prio_queue.impl.h"

#endif
//       prio_queue.impl.h

#ifndef __prio_queue_impl_h_
#define __prio_queue_impl_h_

#pragma interface

// Member-function implementation for class Prio-queue

template<class T : NodeIndex>
void PrioQueue<T>::insert_support(T& key,int index)
{
        heapsize_min++;
        key.partnerIndex = heapsize_min;
        h.heap_insert(1,key,index,a,b);
        key.partnerIndex = b[heapsize_min].partnerIndex;
        h.heap_insert(0,key,heapsize_min,b,a);
};


template<class T : NodeIndex>
PrioQueue<T>::PrioQueue()
{
  heapsize_max = 0;
  heapsize_min = 0;

  a = new T[PRIO_QUEUE_SIZE+1];
  b = new T[PRIO_QUEUE_SIZE+1];
};
```

```
template<class T : NodeIndex>
T PrioQueue<T>::remove()
{
   T max = h.heap_extract(1,heapsize_max,a,b);
   int index = max.partnerIndex;
   heapsize_min--;
   if ( index <= heapsize_min )
       h.heap_insert(0,b[heapsize_min+1],index,b,a);
   return max;
};


template<class T : NodeIndex>
T PrioQueue<T>::find_max()
{
   return a[1];
};


template<class T : NodeIndex>
void PrioQueue<T>::insert(T key)
{  if (heapsize_max == PRIO_QUEUE_SIZE)
      {
        T min = h.heap_extract(0,heapsize_min,b,a);
        int index = min.partnerIndex;
        insert_support(key,index);
      }
   else
     {
        heapsize_max++;
        insert_support(key,heapsize_max);
     };
};



template<class T : NodeIndex>
int PrioQueue<T>::queuesize()
{
    return heapsize_max;
};



template<class T : NodeIndex>
T* PrioQueue<T>::queue_max()
{
    return a;
};

#endif
```

```
#ifndef __queue_element__h_
#define __queue_element__h_
//                  queue_element.h

#pragma interface

#include "heap.h"
//#include "PathNode.h"

typedef struct PathNode  *PathNodePtr;

class Queue_element : public NodeIndex{
       void clear();
public:
       int         select;
       float       estimate;
       float       heuristic_estimate;
       float       actual_estimate;
       PathNodePtr path;

       Queue_element(): NodeIndex() { clear(); };
       Queue_element(float est);
};

inline int operator<( const Queue_element x, const Queue_element y )
{
   return x.estimate < y.estimate;
}

inline int operator>( const Queue_element x, const Queue_element y )
{
   return x.estimate > y.estimate;
}
#endif
//                      queue_element.cc

#include <iostream.h>
#include <iomanip.h>

// Stuff for PrioQueue OF Queue_elements.

#include "prio_queue.h"
#include "local.h"

// Constructor implementation for class Queue_element

void Queue_element::clear()
{
  select = 0;
  estimate = 0;
  heuristic_estimate  = 0;
  actual_estimate = 0;
```

```
  path = 0;
}


Queue_element::Queue_element(float e) : NodeIndex()
{
  clear();
  estimate = e;
}

instantiate_PrioQueue(Queue_element);

void printout(char * str, char * s, PrioQueue<Queue_element> & q, int sel)
{                    // sel=1 -> max-queue;    sel=0 ->min-queue
   Queue_element * c;
   int heapsize;
   if (sel == 1)
     {  c = q.a;
        heapsize = q.heapsize_max;
     }
   else
     {  c = q.b;
        heapsize = q.heapsize_min;
     };
   cout << str << "\n";
   cout.setf(ios::left,ios::adjustfield);
   for ( int i=1; i <= heapsize; i++)
      cout << s << "[" << i << "] :  "
           << "estimate = " << setw(10) << setprecision(5)
           << c[i].estimate << ",  "
           << "actual_estimate = " << setw(10) << c[i].actual_estimate << ",  "
           << "select = " << c[i].select << "\n";
   cout << "\n";
};

//                          heap.h

#ifndef __heap_h
#define __heap_h
#pragma interface

struct BaseHeap {
  int parent(int i) { return i/2; }
  int left(int i) { return 2*i; }
  int right(int i) { return 2*i+1;}
};

struct NodeIndex{
  int partnerIndex;
    NodeIndex();
};
```

```
template<class T : NodeIndex>
class heap : public BaseHeap {
   int heapsize;
   int sel;
   T * c;
   T * d;

   void swap( T *, T *);
   void heapify(int);
public:
   heap(){};
   ~heap(){};
   void heapify_sel(int, int, int, T *, T *);
              // parameter: sel, index, heapsize, heap_array_a, heap_array_b
   T heap_extract(int, int&,  T *, T *);
              // parameter: sel, heapsize, heap_array_a, heap_array_b
   void heap_insert(int, T, int, T *, T *);
              // parameter: sel, key, index, heap_array_a, heap_array_b
};

#include "heap.impl.h"
#endif __heap_h
//                    heap.impl.h

#pragma interface

// Member-function implementation for class heap
template<class T>
void heap<T>::swap(T *px, T *py)
{
   T temp;
   temp = *px;
   *px  = *py;
   *py  = temp;
};


template<class T>
void heap<T>::heapify(int i)
{
   int extremum;
   int comp;
   int l = left(i);
   int r = right(i);

   if (sel == 1)
       comp = c[l] > c[i];          // comparsion for max-prioqueue
   else
       comp = c[l] < c[i];          // comparsion for min-prioqueue

   if ((l <= heapsize) && comp)
       extremum = l;
```

```
    else
        extremum = i;

    if (sel == 1)
        comp = c[r] > c[extremum];    // comparsion for max-prioqueue
    else
        comp = c[r] < c[extremum];    // comparsion for min-prioqueue

    if ((r <= heapsize) && comp)
        extremum = r;
    if (extremum != i)
      {
        swap( &c[i], &c[extremum]);

        d[c[i].partnerIndex].partnerIndex = i;   // let node of other array poin
        d[c[extremum].partnerIndex].partnerIndex = extremum;
                                          // let node of other array point to thi

        heapify(extremum);
      };
};


template<class T>
void heap<T>::heapify_sel(int choose, int index, int heapsize1, T * heap_array_
{
  if ( choose == 1)
     { heapsize = heapsize1;      // parameter for max-queue
       sel = 1;
     }
  else
     { heapsize = heapsize1;      // parameter for min-queue
       sel = 0;
     };
  c = heap_array_a;
  d = heap_array_b;
  heapify(index);
};


template<class T>
T heap<T>::heap_extract(int choose, int& heapsize_ext, T * heap_array_a, T * he
{
    c = heap_array_a;
    d = heap_array_b;

    T extremum = c[1];
    c[1] = c[heapsize_ext];

    d[c[1].partnerIndex].partnerIndex = 1;
                    // let partnernode of other array point to this node
```

```
    heapsize_ext--;
    heapsize = heapsize_ext;
    sel = choose;
    heapify(1);
    return extremum;
};


template<class T>
void heap<T>::heap_insert(int sel, T key, int leafindex, T * heap_array_a, T *
{
    int comp;
    c = heap_array_a;
    d = heap_array_b;
    int i = leafindex;

    if (sel == 1)
        comp = c[parent(i)] < key;    // comparsion for max-prioqueue
    else
        comp = c[parent(i)] > key;    // comparsion for min-prioqueue

    while ((i > 1) && comp)
        {
          c[i] = c[parent(i)];

          d[c[i].partnerIndex].partnerIndex = i;
                          // let partnernode of other array point to this node

          i = parent(i);
          if (sel == 1)
              comp = c[parent(i)] < key;    // comparsion for max-prioqueue
          else
              comp = c[parent(i)] > key;    // comparsion for min-prioqueue

        };
    c[i] = key;

    d[c[i].partnerIndex].partnerIndex = i;
                         // let partnernode of other array point to this node

};


//                      PathNode.h

//#ifndef PathNode_h_
//#define PathNode_h_

typedef struct PathNode  *PathNodePtr;
typedef struct Pair  *PairPtr;
struct PlacedSurfacePair;
```

```
struct PathNode{
     PathNodePtr  parent;
     PlacedSurfacePair * s;
     int          me;
     int          child_index;
     PathNode();
  int print() const;
  int length() const;
};



//#endif PathNode_h_
//                      PathNode.cc

#include "libu/messages.hxx"
#include "PathNode.h"

PathNode::PathNode()
{
  parent = 0;
  child_index = 0;
  me = 0;
  s = 0;
}

int PathNode::print() const
{
  const PathNode * node = this;
  while (node->me)
    {
      ::info("%d", node->me);
      node = node->parent;
      if (node->me)
  ::info(":");
    }
}

int PathNode::length() const
{
  int l = 0;
  const PathNode * path = this;
  while(path->me) {
    path = path->parent;
    l++;
  }
  return l;
}


struct ViewGroup;
```

```
struct DataSurface;
struct Surface;
struct File;
struct Feature;
struct RichContext;
struct Context;

#include "libu/Type.h"
#include "libu2/ExpandingScalarVec.h"
#include "libu2/ChunkingVector.h"


typedef struct SurfacePair * SurfacePair_P;
typedef struct SurfacePair ** SurfacePair_PvP;
typedef struct AssemblyPair * AssemblyPair_P;
typedef struct AssemblyPair ** AssemblyPair_PvP;


struct Pair {
  enum Type {NONE, SURFACE, ASSEMBLY} type;
  float invoked_plausibility;

  Pair(Type t, float p):type(t),invoked_plausibility(p) {}

  virtual int   print() const;

public:
  static Pair * fread(File&);
};

struct SurfacePair : public Pair {
  int modelindex;
  Surface * model;
  DataSurface * data;

  SurfacePair(Surface * m, int i, DataSurface * d, float plaus):
    Pair(SURFACE, plaus),
    model(m),modelindex(i),data(d) {}
  int   print() const;

static ChunkingVector<Surface*> all_surfaces;
static Surface * surface(int index) { return all_surfaces[index]; }
static int numsurfaces() { return all_surfaces.last(); }
};

struct AssemblyPair : public Pair {
  ViewGroup* model;
  RichContext* data;

  AssemblyPair(ViewGroup * m, RichContext * d, float plaus):
    Pair(ASSEMBLY, plaus),
    model(m),data(d) {}
```

```
  int   print() const;

};

AssemblyPair * isAssemblyPair(Pair * p);
SurfacePair * isSurfacePair(Pair * p);

struct PairVector : ExpandingScalarVec<Pair*> {
  int load(const char *);
  int save(const char *) const;
};
#pragma ident "%Z%%M% %I% %E%"

//
//        Robot Vision Group
//  Dept. of Artificial Intelligence
//      University of Edinburgh
//
// Author: Andrew Fitzgibbon
// Date:
// Description:
//

#include <stdio.h>
#include <string.h>

#include "libu/messages.hxx"
#include "libu/genutils.hxx"
#include "libu2/File.h"
#include "libsms/model.hxx"
#include "libsms/viewgroup.hxx"
#include "libdata/Surface.h"
#include "libdata/ContextImage.h"
#include "Pairs.h"

static char * assembly_token = "AsmHyp";
static char * surface_token  = "Surfhyp";

extern ContextImage data;

int Pair::print() const
{
  return info("Uninitialized pair\n");
}

int SurfacePair::print() const
{
  return info("[Surface %d, %s, %7.4f]\n", data->get_id(), model->get_name(), i
}

int AssemblyPair::print() const
```

```
{
  return info("[Assembly %d, %s, %7.4f]\n", data->get_id(), model->get_name(),
}

ChunkingVector<Surface*> SurfacePair::all_surfaces;

Pair * Pair::fread(File& fp)
{
  char buffer[200];
  if (fscanf(fp," %s", buffer ) != 1 )
    return 0;

  int asmhyp = strcmp(buffer, assembly_token ) == 0;
  if (!asmhyp && (strcmp(buffer, surface_token ) != 0)) {
    fp.error("Unknown MiniHypothesis type %s\n", buffer );
    return 0;
  }

  double plaus = 0;
  int dataid = 0;

  if (fscanf(fp, " plaus %lf context %d is%s", &plaus, &dataid,buffer ) != 3) {
    fp.error("Expected plaus N context N is STR\n");
    return 0;
  }


  Object * model = mb->objects.find(buffer);
  if (!model) {
    fp.error("Unknown object [%s]\n", buffer);
    return 0;
  }

  Context * c = &data.all_contexts[dataid];

  if (asmhyp)
    return new AssemblyPair((ViewGroup*)model, (RichContext*)c, plaus);
  else {
    Surface * surface = (Surface*)model;
    // check if this surface has been seen before...
    int index = -1;
    ScanVector(SurfacePair::all_surfaces, s)
      if (*s == surface) {
index = s_index;
break;
      }
    if (index == -1)
      index = SurfacePair::all_surfaces.add(surface);

    info(2,"Surface %s, index %d\n", surface->get_name(), index);
```

```
    return new SurfacePair(surface,
   index,
   ((SimpleContext*)c)->get_surface(),
   plaus);
  }
}

int PairVector::load(const char * stem)
{
  File fp(local_strcat(stem,".hyps"), "r");

  info("Loading invoke pairs ... ");
  resize(1000);
  int i = 0;
  while ((*this)[i++] = Pair::fread(fp))
    if (i >= n)
      resize(n+1000);
  resize(i-1);

  info("got %d.\n", i-1);
  return 1;
}
```