2-D Scene Analysis Using Split-Level Relaxation

Tom Henderson and Ashok Samal

Computer Science Department
University of Utah
Salt Lake City, Utah 84112 USA

UUCS-85-113

26 November 1985

Abstract

We present a new method for applying multiple semantic constraints based on discrete relaxation. A separate graph is maintained for each constraint relation and used in parallel to achieve a consistent labeling. This permits both local and global analysis without recourse to complete graphs. Here local means with respect to a paricular constraint graph, and thus actually includes global spatial relations on the features; e.g., parallel edges on an object will be neighbors in the parallel constraint graph even though they are far apart in Euclidean space. Another major result is a technique for handling occlusion by incorporating the use of spatially local feature sets in the relaxation-type updating method.

This work was supported in part by NSF Grants ECS-8307483, MCS-82-21750, DCR-8506393 and DMC-8502115.

1. Introduction

One of the problems in computer vision is to identify the set of objects present in a given image. This is, in essence, the *Scene Labelling Problem.* The problem can be mapped into what has been variously called the *Consistent Labelling Problem* [13], the *Satisfying Assignment Problem* [10], the *Constraint Satisfaction Problem* [17], *Waltz Filtering* [24], etc. We will refer to it as the Consistent Labelling Problem (**CLP**). It has been shown this problem is NP-complete. There have been several approaches to solving the problem, including: backtracking, graph matching, relaxation, etc.

The approach used here is based on discrete relaxation. However, there are some major philosophical differences. As explained later, we distinguish between different types of constraints during the process of relaxation. Many approaches based on relaxation make use of only the local constraints as opposed to global constraints. They tend to ignore the global constraints because occlusion sometimes prevents their use. We argue that they can be useful, and under certain circumstances they may be extremely advantageous. The model proposed in this paper uses both types of constraints. Also, with the advent of parallel computers it is imperative to look at the problem again and see if the approaches are suitable for parallel processing or not. This is especially important in light of the fact that the problem at hand has an exponential growth rate. For an example of the use of multiple semantic constraints with stochastic relaxation, see Faugeras and Proce [9].

We give a formalism to describe the 2-d scene analysis problem as a Consistent Labelling Problem. We also explore the suitability of the approach for parallel processing.

1.1. The Consistent Labelling Problem (CLP)

Although there are several variations, the **CLP** can be formulated as follows. Given,

- A set of items or units, $\mathbf{U} = \{u_1, u_2, \dots, u_n\}$.
- Each unit u_i, has a domain D_i, which is the set of acceptable labels. Often the units all have the same domain. In that case $D_1 = D_2 = \dots = D_n = D$. Also, $\mathbf{D} = \{D_1, D_2, \dots, D_n\}$.

A *labelling* $\mathbf{L} = (L_1, L_2, \dots, L_k)$, where $k \le n$, $L_i = (u_i, l_i)$, $u_i \in \mathbf{U}$ and $l_i \in D_i$. L_i's are called *unit-labels.* Without loss of generality, we can assume that all u_i's are distinct. If $k < n$, then the labelling is a *partial* labelling and if $k = n$, then it is called a *complete* labelling.

A unit can have any label which is in its domain. Usually, however, there are restrictions on the labels a set of units can have simultaneously in order to be consistent. These constraints are expressed by a *Constraint Relation* **R**. Potentially **R** can be an n-ary relation.

A pair of unit-labels $L_i = (u_i, l_i)$, $L_j = (u_j, l_j)$ are consistent if and only if $(u_i, l_i, u_j, l_j) \in \mathbf{R}$. A given labelling $L = (L_1, L_2, \dots, L_k)$ is consistent if unit-labels L_i and L_j, are consistent for all $i, j \le k$.

The goal of the labelling problem is to find a <u>complete</u> <u>consistent</u> labelling, given a set of units **U**, the domains of these units **D**, and the constraint relation **R**.

<CLP> ::= (**U**, **D**, **R**).

Other formulations of the problem ask for all solutions, but it doesn't change the nature of the problem. If a complete consistent labelling can not be obtained, then the largest (or "best") partial labelling should be found.

1.2. Solutions to CLP

The consistent labelling problem can be solved in many ways. The most straightforward method is what is called the *Generate and Test* method. Here we enumerate all the possible configurations and then select those which are consistent. It should be obvious that this method is going to be extremely slow for problems with large **U** and **D**. For example, if |**U**| = 10, and |D| = 10, then the number of possible configurations is 10^{10}. In many cases the labels assigned to the first few units make the whole labelling inconsistent and this can be detected early, during the configuration process. This observation could save a lot computation.

A better method, which takes advantage of the above observation is *standard backtracking*. Here, we start with a single unit and assign a label to it from its domain. Then we select another unit from the rest of the units, and assign a label to it, from its domain such that the partial labelling built so far is consistent. If at any point we can't find such a label, then we back up one step and give the next possible label to the unit which was last assigned a consistent label and continue the process. If we manage to assign labels to all the units consistently then we have found a solution, and if we run out of labels then there is no solution.

Although standard backtracking is more efficient than the generate and test method, it is still not good enough for many practical applications. There have been several approaches to overcome this problem. Gashnig [10] attempted to solve this within the backtracking framework and gave two new backtrack-type algorithms:

- BACKMARK - where all redundant pair-tests are eliminated.
- BACKJUMP - where it is possible to backtrack across multiple levels, instead of just one level.

These two algorithms do indeed show better performance than the standard backtrack algorithm, as shown by Gaschnig. Haralick and Elliot [11] gave another algorithm called *Forward Checking* to improve backtracking, which performs better than these in some cases. However, these results are only known to hold on problems where the labeling of each unit is constrained by all the other units.

Haralick et al. [12] have described two look-ahead operators Φ, Ψ to reduce the computation during the backtracking process. Haralick and Shapiro [13, 14] generalized these operators to have arbitrary look aheads.

Waltz [24] took another approach to solve the problem. The basic idea is to initially assign all possible labels to all the units and remove a label from the label-set of a unit if it is found that the label is not compatible with any of the labels of the others. Removing a label from a unit in turn makes some labels of some other units inconsistent. This process continues until there is no label of any unit can be removed or the label set of one unit becomes empty. In the former case, it is still necessary to search for an unambiguous solution, while in the latter situation, no solution exists. Convergence to a consistent set of labels depends on the nature of the problem and the constraint set **R**. Rosenfeld et al. [23] have described a modified version of Waltz's filtering algorithm which is a parallel, iterative procedure and was also generalized to allow probabilities to be asociated with the labels.

Mackworth [17] and others use yet another approach to reduce the computation during the backtracking process. For binary constraints, the problem can be nicely formulated using graphs, where the nodes correspond to the units and the arcs represent the constraints between the units. Each node also has an associated label-set, which gives the possible labels for the unit. Montanari [19] explores the fundamental properties of such networks and their applications. Mackworth [17] gives three consistency tests, *Node Consistency*, *Arc Consistency*, and *Path Consistency*, which prevent the *thrashing* behavior of the backtrack algorithms. He also gives several algorithms to achieve the above three consistencies in networks. These consistency checks are done first and then the backtracking process is applied. Recently Mohr and Henderson have given an optimal algorithm for arc consistency and an improved algorithm for path consistency [18].

1.3. Complexity of CLP

In this section we analyze the complexity of the CLP. As Knuth [16] points out, it is difficult to analyze the backtracking algorithms without actually running programs. Even then the results are not very convincing, since it is not obvious why and if they should hold in a totally different problem domain. Gaschnig [10] took the experimental approach to study the complexity of different algorithms.

Since it is known that CLP is NP-Complete [19], it is not so important to find the complexity of the various algorithms used to solve the **CLP**, since the worst case complexity is going to be exponential anyway. What is more important is to determine if a particular algorithm is consistently better than the others. Even this is not a straightforward problem. Gaschnig [10] and Haralick and Elliot [11] have experimental results for the efficiency of some of these algorithms.

1.4. Application of Relaxation to Scene Analysis

Relaxation techniques have been successfully applied to several problems. Gaschnig investigated the application of relaxation to the n-queens problem, criptarithmatic problems, the Soma Cube and Instant Insanity. Henderson and Davis used it for syntactic shape analysis [8, 15]. The application with which we are concerned here is *Scene Analysis*. The problem is to locate and identify all the objects in a given scene. There are several methods to locate the objects. If we locate the boundary of the object then it unambiguously determines the object. Also sometimes a set of features, if located in the right configuration, can determine the location and orientation of the object. So, in order to locate the objects, we must locate the various features (or the boundary edges). Once these features are obtained from the image, they can be assigned a set of possible labels depending on the problem at hand. Then the relaxation procedure can be applied to produce a consistent set of labels and backtracking is used to find a solution.

1.5. Parallel Relaxation

Since the worst case execution time for the problem at hand is exponential, it might be worthwhile to explore if the problem lends itself to a parallel execution model. Also, now that multiprocessors are available we can actually test these models, not just theoretically analyze them. In this paper, we describe a model which exploits the parallelism in the relaxation process.

1.6. Local vs Global Constraints

One of the first steps in locating an object is to locate its features. We can recognize objects on the basis of *global* features, like number of holes, size of various segments, total area of the segments, perimeter, etc. Alternatively we can also use local features to locate objects. Here we use local features like corners, holes, etc. We look for certain structure with respect to these local features in the image, and if we can find such a structure then we can locate the object.

Both methods have their advantages and disadvantages. A system based on global feature matching is prone to mistakes, particularly when the object is occluded or even partially defective. On the other hand, if the object is completely isolated in the scene under consideration then the method is very straightforward and efficient. A system using local features is probably more sophisticated and is more robust in a general scene. But it is also very time consuming.

A method based on local features can use only the local constraints, i.e., constraints between nearby features. This is useful in many circumstances, when the object is partially occluded and chances are that some of the features would be visible and they can be used to identify the object. However, there are instances when it would be very useful to have global constraints also. And in certain instances it is necessary to use global constraints to identify an object. For example, the object in Figure 1(a), as seen in Figure 1(b) can never be identified using local features alone since all the holes are occluded. However, by using global constraints like the parallelism and perpendicularity of sides, we can easily identify the object.

1.7. Scene Analysis Using Local Features

Recently several successful 2-D scene analysis systems have been proposed based on the use of local features [2], [6], [3]. Of course, many of the very first systems proposed were based on global features; for example, Fourier coefficients and moment invariants. In order to give a basis of comparison, we briefly describe a system which uses local features to recognize objects in a scene. It is described in detail in [5]. Most of the terms used in this section are from the above reference. Then we analyse the algorithm for its complexity.

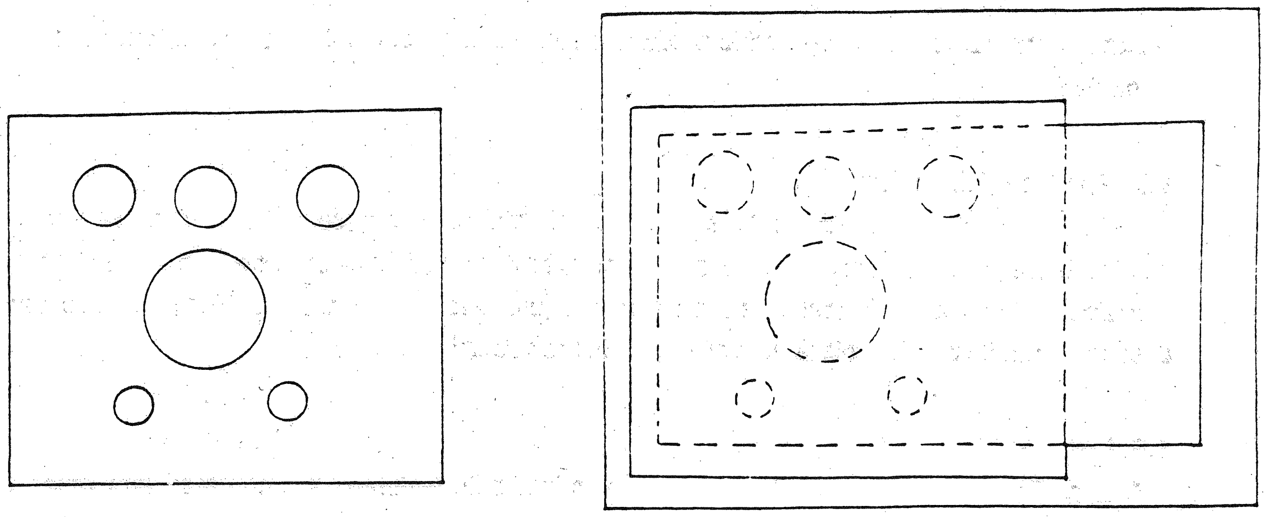

Figure 1. An object and a scene with the object

1.7.1. LFF Method

The Local Feature Focus (LFF) method has two major components: a training system and a run-time system. During the training phase, the object models are fed into the system and the system comes up with a set of *strategies* for recognition of each type of object. For each object it gives a list of features, which are termed focus features along with a set of neighboring features. The idea is that if the focus feature is found in the image along with the neighboring features in the right configuration, with respect to the distance and orientation, then the object's location and orientation can be unambiguously determined.

The steps during the run-time can be summarized as follows:

1. Locate *potentially* all useful local features in the image.
2. Locate a *Focus Feature*. If there are none available in the image, then Quit.
3. Get the set of *near-by* features for the focus feature.
4. Select those features which satisfy certain criteria, i.e., those features which have a corresponding occurrence in the actual image.
5. Make *Object-feature* to *Image-feature* assignments and transform it into a graph.
6. Find all the *maximal cliques* of the graph and hypothesize the object using them.
7. Verify the hypotheses. If the hypothesis succeeds, mark all the visible features of the object in the image as *explained*. Go to 2.

1.7.2. Analysis of the LFF method

Let,

The total number of features in the image be $\mathbf{N_I}$.

The number of line segments defining the boundary of the object be $\mathbf{N_s}$.

The total number of features in the object be $\mathbf{N_f}$.

The number of focus features for the object be $\mathbf{N_F}$.

The average number of nearby features for each focus feature be $\mathbf{N_n}$.

The average number of possible labels for a nearby feature in the image be $\mathbf{N_a}$.

Now we analyze the algorithm defined in the previous section, step by step. We are ignoring the complexity of step 1, since the time it takes is proportional to the size of the image and is (most often) independent of the scheme used.

Step 1: O(image size), since features must be extracted from the entire image.

Step 2: O(constant), since the list of focus features can be built when the local features are being extracted from the image.

Step 3: O(constant), since the nearby features can be put in some kind of *property list* of the focus feature. An indexing scheme could also be used, with the same effect.

Step 4: $O(\mathbf{N_n \cdot N_I})$, since we essentially have to go through the whole list of the image features for each nearby feature, which takes on the average takes $(\mathbf{N_n \cdot N_I}) / 2$ steps.

Step 5: $O(\mathbf{N_n \cdot N_a})^2$, since the number of vertexes in the graph is $(\mathbf{N_n \cdot N_a})$ and for building the graph we have to check for compatibility of each pair of vertexes of the graph.

Step 6: This is a slightly complex step to analyze. The general problem of finding maximal cliques of a graph is known to be NP-complete [1, 4]. However, if the graph is planar then it can be done in linear time with respect to the number vertexes [21]. Using Kuratowski's theorem [7], a planar graph cannot have K_5 or $K_{3,3}$ as a subgraph. This reduces the problem to finding only K_4s, and K_3s in the graph, which can be done in linear time. So, the best case for this step is $O(\mathbf{N_n \cdot N_a})$ and the worst case complexity is $O(k^{f(x)})$, where x is $(\mathbf{N_n \cdot N_a})$ and k is a constant .

Step 7: $O(\mathbf{N_s} + \mathbf{N_f})$, since we have to loop over the list of segments of the object and then we also have to loop over the list of the features for that object.

So, the total complexity for each of iteration through steps 2 to 7 is the sum of all the above complexities. And the complexity of the whole algorithm is $\mathbf{N_F}$ times the above factor, since we have to loop over the list of focus features.

Since *step* 6 in the worst case takes exponential time, the worst case complexity of the algorithm is also exponential.

2.4. Network Model for Relaxation

In this section we briefly describe a graph/network model which is the underlying basis for the relaxation process explained in section 2.6. This is similar to the network model used by Mackworth [17] and others in the sense that the nodes represent the units to be labelled and the constraints are represented by the arcs in the graph. However, there are several differences. What we have here is conceptually closer to a set of graphs than a single graph.

Instead of one graph, we have a set of graphs, one for each of the relations in the image. We also have a set of graphs, corresponding to the object models. This set of graphs is used to represent the constraints in the model. Let G^m, be the composite graph to represent the set of *model graphs* and G^s, denote the same for the *image graph*.

2.4.1. Model Graphs

$G^m = \{ G^m_1, G^m_2, \ldots\ldots , G^m_n \}$, where n is the number of models being considered. Each of the G^m_i also consists of a set of graphs corresponding to the various relations in the constraints.

$G^m_i = \{ G^m_{i,1}, G^m_{i,2}, \ldots\ldots , G^m_{i,k_i} \}$, where k_i is the number of relations in C^m_i.

Each $G^m_{i,j}$ is a graph and can be represented by: $G^m_{i,j} = < F^m_i , E^m_{i,j} >$, where edge (x,y) is in $E^m_{i,j}$, iff $(x,y) \in C^m_{i,j}$. The nodes of the graph are the features in the image which need to be labelled and the arcs represent a relation.

2.4.2. Image Graphs

Unlike the model graphs, we have only one composite image graph, since we are concerned about only one image at a time. However, the composite image graph, G^s,is again composed of a set of graphs, one for each of the relations in the image. So,

$G^s = \{ G^s_1, G^s_2, \ldots\ldots , G^s_{k_s} \}$, where k_s is the number of relations used in the image graphs.

Each of these graphs corresponds to a relation in the image. So,

$G^s_i = < F^s, E^s_i >$, where edge (x,y) is in E^s_i, iff $(x,y) \in R^s_i$.

2.5. An Example

Before proceeding any further, we present an example to make things clearer. The example is simpler than the scenes one would normally find in actual cases. But it is only used to explain the concepts. Figure 2 shows two simple industrial parts, taken from [20]. We would refer to it as $Part_1$ and $Part_2$, respectively. The set of features for each part consists of only its boundary edges.

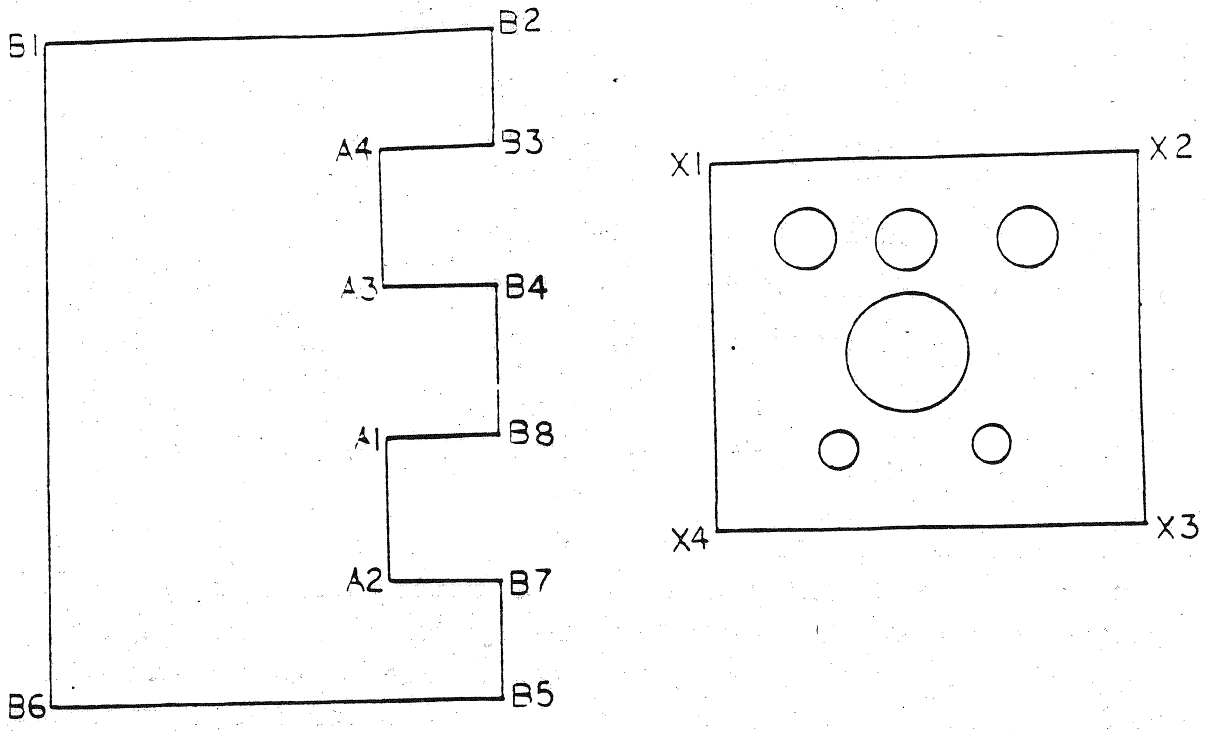

Figure 2. Part$_1$ and Part$_2$

M = { *Part*$_1$, *Part*$_2$ }.

Part$_1$ = (F_1^m, C_1^m), *Part*$_2$ = (F_2^m, C_2^m).

F_1^m = {B1B2, B2B3, A4B3, A3A4, A3B4, ..., A2B7, B5B7, B5B6, B1B6, B1B2}.

C_1^m = { *parallel, perpendicular, longer-than, equal-length* }.

F_2^m = { X1X2, X2X3, X3X4,X4X1 }.

C_2^m = {*parallel, perpendicular, longer-than, equal-length* }.

We have four relations to express the constraints in the model for *Part*$_1$. These four relations are given in Figure 3. The relations are not complete. It just gives an idea of what these relations looks like. *Part*$_2$ has similar relations.

Parallel	*Perpendicular*
(A3B4 A1B8)	(B1B2 B2B3)
(A1B8 A2B7)	(B2B3 A4B3)
(A2B7 B5B6)	(A4B3 A3A4)
(B1B6 A3A4)	(A3A4 A3B4)

Longer-than	*Equal-Length*
(B1B6 B1B2)	(B1B2 B5B6)
(B1B6 A3A4)	(A3A4 A1A2)
(B1B2 B2B3)	(A4B3 A3B4)
(A3A4 B2B3)	(A1B8 A2B7)

Figure 3. Constraint Relations in *Part*$_1$

2.6. The Relaxation Process

In this section we describe how the actual relaxation process is executed and how it fits in a parallel processing framework. The first step is to build all the graphs, i.e., all the model graphs and the image graphs. With each node in the image graph we associate a set of labels, which represents the set of features it could be. It should be pointed out that, although we treated the graphs separately in the previous sections, they need not be really disjoint in actual implementation. Since the node set of all the image graphs is the same, they could be shared. The same applies to the model graphs, too. The topology of all these graphs remains unchanged during the relaxation process. What changes is the label set attached to each of the nodes in the image graph. The model graphs remain completely unchanged.

Once the graphs are constructed, the next step is to enforce the *Node*, *Arc*, and *Path* consistencies in the image graphs. This is where the system lends itself to parallel execution. We have these graphs which are independent in the sense that they represent totally different types of constraints. We may have one graph to incorporate the size constraints, e.g., *Edge*$_1$ is *longer than Edge*$_2$. We may also have another graph to represent the directional constraints, e.g., *Edge*$_1$ is *parallel* to *Edge*$_2$, etc. The size and directional constraints are independent. However, they share the same set of nodes. To exploit parallelism we create a process for each of the constraint types. So, in the above example, we will have four processes, trying to enforce consistencies in the *parallel*, *perpendicular*, *longer-than* and *equal-length* graphs. Hence the relaxation process is now distributed or *split* into sub-processes or *levels* which work on individual constraint types. The sub-processes structure is given in Figure 4. The controlling process is described in Figure 5.

After the consistencies are enforced, we find all the solutions using standard backtracking or one of modified schemes described in section 1.2. Hopefully, the amount of backtracking would be much less now that the networks are arc and path consistent.

```
type
   state = ( changed, unchanged, working );

process relaxG(i : integer);
begin
   repeat
      Status[i] := working;
      Enforce Arc, Node and Path consistencies in the ith graph;
      If there is change in the label-set of any node then
         Status[i] := Changed
      Else
         Status[i] := UnChanged;
   until terminate;
end;
```

Figure 4. Subprocesses Structure

```
var
      m  : integer; (* number of graph types *)
      n  : integer; (* number of models *)
      k  : integer; (* number of features in the image *)
process relaxS(scene, models);
begin
   var i, temp : integer;
         Count  : integer;
         status  : array [1 .. m] of state;
         N          : Set of nodes;
         $L_{all}$       : Set of Features;

   $N := F^s$;   (* Features in image are nodes *)

   $L_{all} := \cup_{i=1}^{n} F_i^m$;

   build the image and model graphs;
   for i := 1 to k do  Label-set of node $N_i := L_{all}$;
   for i := 1 to m do  Status[i] := changed;

   Count := 0;
   for i := 1 to m do  fork-process(relaxG, i);

   while ( Count = m ) do
   begin
      temp := 0;
      for i := 1 to m do
         if ( Status[i] = unchanged ) then
            temp := temp + 1;
      Count := temp;
   end;

   terminate := true;
   find-solutions();
end;
```

Figure 5. Controlling Process

In order to prove the correctness of the of the algorithms in the previous section, we state the following proposition.

Proposition. If a label **L** is removed from the label set of a node **N** in any graph, then the node **N** can't have the label **L** in any complete consistent labelling.

The proof is fairly obvious. For a labelling to be consistent it must be consistent in all the underlying graphs types. Once a label in a node is removed by a process **relaxG** working on graph i, it is not consistent with respect to constraint type i. That means, it cannot be part of a globally consistent labelling. So, we don't lose any solutions by the above procedure.

The correctness of the algorithm is also obvious. After consistency checks are enforced, we use backtracking to find all the solutions and reject all those labels which don't lead to complete consistent solutions.

2.7. Efficiency Considerations

In section 2.6 we described an algorithm to solve the scene analysis problem. In section 2.6 we proved that it is both correct and complete. However, the performance of the algorithm given is very poor. In this section we analyze its efficiency and give some improvements to make it more efficient.

2.7.1. Complexity

The inefficiency of the above algorithm can be seen very clearly at the step where we assign the initial label-set to each of the nodes in the image graph. We assign $\mathbf{L}_{all}$ to label-sets of all the nodes. $\mathbf{L}_{all}$ is the union of all the all the features of all the models which can be in the scene.

$$\mathbf{L}_{all} = \cup_{i=1}^{n} \cup_{j=1}^{m_i} \mathbf{F}_{i,j}^{m}.$$

Clearly, this is a very large set. Even though many of the labels would be removed during the first few iterations of the relaxation process, this is definitely a major source of inefficiency.

2.7.2. Improvements

One simple and obvious improvement to the algorithm can be made by deleting those labels from the label-set of a feature which are not of the same type. For example if the feature in the image is a hole, it makes little sense in assigning a label which is a corner or a boundary edge. Although this is a very simple modification, it should considerably reduce the initial size of the label-set of the nodes.

However, it doesn't solve the problem entirely. One of the reasons the above method is so inefficient is because we are trying to do too many things at the same time. Although we are enforcing the constraints in different graphs in parallel, which is obviously a great help, we are still in essence trying to label all the features of the image at the same time. Since this in general, is going to be a very tough problem to handle (a scene may have a hundred features), it might be worthwhile to break the problem into smaller problems and try to solve them (Divide and Conquer paradigm). We should perhaps note here again that we are dealing with a NP-complete problem. Since in the worst case it is going to take exponential time anyway, all we are trying to do is to gain speed-up wherever possible.

One way to break the problem into smaller (hopefully simpler) ones, is to divide the scene into a set of smaller sub-scenes and try to do the analysis on these. Although it might sound very simple, there are problems with this scheme. The most obvious one is how are we going to do the subdivision?

One simple way is to divide it into an **m** by **n** rectangular grid. Here again the problem is how would one go about choosing optimal values for **m** and **n**. Also, there might be duplication of effort since, an object may be spread over more than one rectangle.

The problem with the rectangular grid scheme is that we have no basis for performing the subdivision. We are essentially using heuristics to choose the values of **m** and **n**. Instead we should be using the image and the models to guide our sub-division process. In the next section we propose a method which is based on the understanding of the models and the image.

2.8. New Model

As mentioned in the previous section, the basic idea is to let the models and the image direct the process of dividing the scene into sub-scenes. There are two key observations which should be noted.

1. A feature instance in the image can belong to only a few objects. For example, if the feature under consideration is a *hole*, then it can only belong to objects which have holes.
2. If a feature instance in the image is hypothesized to belong to a particular object, then the other features of the object should be located within a certain distance of it.

What the first observation means is that, a feature can belong to only certain objects and it should have only the labels corresponding to the features of these objects. The second implies that it is useless to consider the the portion of scene which could not have any features of the object under consideration. The above two observations form the basis of the subdivision of the scene into sub-scenes.

The algorithm which uses these ideas is given in Figures 6 and 7. The process *relaxM* is the top level process which spawns processes to work on sub-scenes. The process *relaxSS* works on the subscene and for each possible object it tries to perform relaxation and find if the hypothesis is true.

```
process relaxM;
begin
  while (there is an unmarked feature in the image) do
  begin
    f := first unmarked feature;
    O := find-objects(f); (* find objects it can be a part of *)
    for (every object O_i in O) do
      fork-process(relaxSS, O_i, f);
  end;
end;
```

Figure 6. Top Level Process

```
process relaxSS(O : object, f : feature);
begin
   S := features-in-bounding-circle(O, f)
   for each feature f in S do mark(f);
   relaxS(S, O);
   if object O is confirmed then
      mark all its features as explained;
   T := all there features in S;
   If T is not empty then
   begin
      unmark every feature in set T;
      restart process relaxM;
   end;
end;
```

Figure 7. Process for working on a sub-scene

Process *relaxM* controls the top level division of the scene into sub-scenes. Function *find-objects* returns all the objects which can have a particular feature. A very simple indexing scheme can be used (See section 2.8.1) to do this. Once a feature is chosen and the objects it can belong to are associated with it, process *relaxSS* works to see if any of the objects actually are in the scene. This could be done in parallel as shown in the processes **relaxS** and **relaxG**. The process finds all the features in the bounding circle of the feature. The radius of the bounding circle is determined beforehand and is available, see section 2.8.2. The process then marks off these features, since they potentially could be *explained*. Then actual relaxation is done on this sub-scene using the object's model. If it turns out that indeed the object is in the sub-scene, then all the features which have been labelled by the relaxation process are marked as *explained* and all the other features are unmarked. This process also restarts the main process if necessary.

2.8.1. A Simple Indexing Scheme

Function *find-objects* in the process **relaxM** takes a feature and returns a set of objects it can be a part of. This can be done by a brute force method, by searching each object for its feature-list and checking if it has the given feature as a member. It is a simple but a very inefficient method. To make it faster, a simple indexing scheme is very effective. The objects are indexed using the features as the keys and when we need the objects for a given feature, it is a one step operation. The computation needed for making the index tables is done before run-time and is a one time step. Also, new objects can be added to the list incrementally, without totally reconstructing the index tables.

Figure 8 shows a hypothetical set of objects and their features. From it we construct the index tables which are given in Figure 9.

$\mathbf{M} = \{ \mathbf{M}_1, \mathbf{M}_2, \mathbf{M}_3, \mathbf{M}_4, \mathbf{M}_5 \}$.

$\mathbf{F}_1$ = { round-hole, square-hole}.
$\mathbf{F}_2$ = { square-hole, triangular-hole}.
$\mathbf{F}_3$ = { round-hole, square-hole, triangular-hole}.
$\mathbf{F}_4$ = { round-hole}.
$\mathbf{F}_5$ = { square-hole}.

Figure 8. A set of objects

triangular-hole	:: { $\mathbf{M}_2, \mathbf{M}_3$ }.
square-hole	:: { $\mathbf{M}_1, \mathbf{M}_2, \mathbf{M}_4, \mathbf{M}_5$ }.
round-hole	:: { $\mathbf{M}_1, \mathbf{M}_3, \mathbf{M}_4$ }.

Figure 9. Index tables for the objects in Figure 8

This indexing can be extended further on the basis of certain measures of the features. For example, if $\mathbf{M}_1$ has round-holes of only a certain size, which is different from those of $\mathbf{M}_3$ and $\mathbf{M}_4$ then the round-holes can be further indexed with the size as the key. However, not all features may have such a measure.

2.8.2. Bounding Circle

Here we describe a simple method to determine the bounding circle for a feature of an object. This helps in breaking up a scene into sub-scenes on which the relaxation is done. Like the index tables, this is done before run-time and is a one time computation. We associate with each feature of the object, a position in two dimensions. So each pair of features f_i and f_j has a distance measure $d_{i,j}$ associated with it. We choose the Euclidean distance as the measure of distance. The radius of the bounding circle, r_i for the feature f_i is the maximum of the distances associated with it.

$$r_i = \mathbf{Max} (d_{i,j}) \; \forall \, j : f_j \in \mathbf{F}.$$

Figure 10 describes the computation.

```
function BCircle(f, F) : real;
begin
   var max, dist : real;

   max := 0.0;
   for all features f_i in F do
   begin
      dist := distance(f, f_i);
      if ( dist > max ) then max := dist;
   end;

   BCircle := max;
end;
```

Figure 10. Procedure for computing the radius of the Bounding Circle

Function *distance* computes the Euclidean distance between the positions of the two features.

2.8.3. Comments

The approach we have proposed in this section works better in certain situations than others. Here some of those situations are described. Since our subdivision is based on the scene itself and for the bounding circle we use the worst case estimate, if the objects are very close together, we would use features of both objects for the relaxation process and it would slow down the process. On the other hand if the two objects are close together, then the chances are that many features of one (or some of both) would not be visible. So, the number of features we have to deal with may not be as bad as it may seem.

If however, the objects are spread out without much overlap, the relaxation process would reach a solution really fast. Also, even if the objects are overlapping in the physical space, but fairly disjoint in the feature space, then also, the convergence to the solution is fast, since those features which can't belong to the object under consideration are ignored, thereby effectively reducing the number of initial labels the node in the graph can have. For example, if one object has only square holes and another has only round holes, then even if the square and the round holes may lie close to each other in the image, we would not include the round holes in the list of features in the bounding circle for a square hole.

3. Extending Relaxation for Occluded Scenes

So far we have not given the details of how the relaxation process actually works for occluded scenes. In this section, it is described in detail, since it has been transformed into a very different form. Although the relaxation process works fine for non-occluded scenes, it doesn't work well if the scene is occluded. The basic reason is that if a constraint is missing in the image, it doesn't mean that the constraint actually doesn't hold. It is possible that a constraint is not satisfied because some or all of the associated units are occluded. So the constraints don't have the same discriminating power in occluded scenes. In unoccluded scenes, we use the boundary edges directly to describe the constraints, but this can't be done in occluded scenes.

Also, some of the constraints have to be used or interpreted differently. For example, if a boundary edge X is longer than another edge Y in the model, it doesn't follow that the corresponding edge for X in the image will be longer than the corresponding edge for Y. In fact, if a boundary edge W is longer than another boundary edge Z in an image, no definite conclusion can be made about them from this information alone.

However some of these constraints can be used under certain circumstances. For example, if there are only a few boundary edges which are longer than some absolute amount and we find an edge in the image which is longer than this threshold, then it has to be one of the above edges. But this doesn't constrain the problem enough to reduce the solution set of the other labels.

One way to get around this problem, is to use only local features like holes, corners, etc. and constraints between them. Instead of using constraints between the sides (or boundary edges), we use constraints between the lines between the locations of features. We refer to these vectors as *inter-feature vectors* (or iv's). The advantage of using these iv's instead of the sides is that, unlike the sides, they are either present or absent; i.e., they cannot be partially missing or broken up into different parts because of occlusion. If both the features constituting an iv are present in the image, then the iv is defined, otherwise it is not. We can use the same constraints as before, but now they are between these iv's instead of the sides. Before they were binary constraints, but now they are essentially 4-ary constraints.

However, the constraints still don't have the discriminating power to drive the relaxation process, since the constraint set in the image is incomplete. Initially each unit has all the labels of the model in the label-set. We delete a label at a node if a certain constraint is not satisfied. But now we are not sure if the constraint is unsatisfiable, or is just not satisfied because of occlusion. So, in order to drive the relaxation process we need to seed the label-set of some units and then let relaxation take over. The goal is to get some positive information from the scene and then propagate it. The next two subsections describe how the iv's are created and how the initial seeding is done.

3.1. Creation of Inter-Feature Vectors

An inter-feature vector is defined to be the vector between two feature locations. The magnitude of the vector is the distance between the two feature locations. If there are N features under consideration, then there could be as many as N * (N-1) iv's. This could be too large to be manageable, particularly in an image where there could be a large number of features. To avoid this problem, we connect only the features which are within a certain distance of each other. This distance is not arbitrarily chosen, rather it is computed for each model separately so as to keep the number of iv's small. For example, for the object in Figure 9, there are 28 iv's for a distance threshold of 2.1.

3.2. Initial Seeding of Label-Sets

The goal of the seeding process is to reduce the label-sets of some units, drastically if possible so that it will the drive the relaxation engine. This means we need to find the *peculiar* properties of some features, their locations or their relationships. Since we are using the iv's as the basis for the constraints, we look for using the iv's as the basis for our constraints, we decided to look for peculiarities among these iv's. Two such quantities are used here and we found them very useful for the seeding process. They are:

- The **magnitude** of the inter-feature vector (or the distance between the corresponding two feature locations).
- The **angle** between two inter-feature vectors. The two iv's are chosen such that they have exactly one common end point, i.e., they share one feature.

We group these quantities and try to find quantities which are very unique. In our case we say a quantity is unique, if it doesn't occur more than a certain number times in the above groups. We determine this by histogramming. Figure 11 gives a brief outline of how this process works. It should be noted that all these things (threshold-distance, inter-feature vectors, histogramming, etc.) are done for the models beforehand. This computation is performed only once.

```
procedure ComputeSpecialDistances;
begin
      Compute the threshold-distance;
      for i:= 1 to No-of-Features do
          for j:=i+1 to No-of-Features do
              If distance(feature[i], feature[j]) <= threshold-distance then
                  iv = Make-inter-feature-vector(feature[i], feature[j]);
                  Append(IVs, iv)
              endif;
      size := GetSize(IVs, error)
      H := Make-histogram(IVs,size)
      for i:=1 to size do
          if H[i] <= Nunique then
                  Append(Special-Length-List, details(i));
          endif;
      Store the Special-Length-List in Model;
end;
```

Figure 11. Computation of Special Distances

Function *details* collects some more information about the inter-feature vectors which correspond to the i'th histogram entry (for example, the associated features, etc.). Function *GetSize* computes the size of the histogram. It is computed such that two consecutive entries in the histogram are at least *error* apart. All the special distances are collected in a list and become a part of the model. Special angles are also computed by a similar procedure and are stored in the model. A special-distance is actually not a single quantity, but it has a range associated with it. The idea is that if the distance between two units in the image is in this range, then the two units should correspond to the two features in the model.

In the case of the objects given in section 4.1, we found that there are a few special lengths and angles for each of the models, that are fairly unique in the sense that, there is practically no other quantity which is close to these special quantities in terms of their magnitude. This information can be used to seed the label-set of units in the image.

3.3. Labelling Process

The process of labelling starts with *type-checking*. Initially each unit (primitive) in the image is given all the labels of the desired model. Then, if the types of the unit and any of its labels are not consistent then those labels are removed from the label-set (see section 2.8.1). The next step is the seeding process. Before the actual seeding can be done, the inter-feature vectors and the angles are constructed for the image. The distance used is the threshold-distance used by the model. After the lengths and the angles are constructed, they are searched for the special angles and lengths. If any of the distances (or angles) match a special distance (or angle) for the model, then the label-sets of the associated units are updated.

Then the control is passed on to the relaxation process, and finally to the backtracking operator. Due the changed nature of the problem, the structure of the relaxation process has been changed considerably, too. In the next section, we give the motivation and the structure of the new version of the relaxation process.

3.4. Split-Level Relaxation Process

The main reason to change the structure of the relaxation process is that there is no way to use positive information in relaxation. For example, if during the seeding process we infer that the primitive X can be either A or B, then we really can't use this information in any way more than if the label-set of X had A and B without seeding. Relaxation treats all the nodes equally and is not able to use the information from seeding, which is very powerful and positive information. As a matter of fact, it is possible for node X to lose all its labels, during relaxation just because it lacks support at some other primitive which may not even belong to the model.

We divide the nodes (primitives) into two groups. The nodes whose labels are fixed during the seeding process, are called the *strong* nodes and others are called *weak* nodes. The *strong* nodes signify positive information. The strong nodes, always remain strong, while the weak nodes may be elevated to strong status. During relaxation, a strong node can affect the label-set of another (either strong or weak) node. This prevents the nodes which are not really a part of the model from affecting the label-sets of other nodes. Those units which survive the first iteration, i.e., whose label-set is not empty at the end of the iteration, are then changed to *strong* nodes. Those nodes which do not survive the first iteration, are not considered further and are not considered to be not part of the model. After that, the relaxation process works as usual.

3.4.1. A Second Approach

Another way to solve the problem of *weak* nodes deleting the labels of *strong* nodes is to allow a label at a node to remain, if there is at least one support for it from any one other node. (In standard discrete relaxation, a label remains iff it has support from all neighboring nodes.) This is motivated by the fact that some of the constraints may be unsatisfied due to occlusion. It should be noted however that, there has to be support for the label in each of the possible graph types. This is fairly effective, but takes more time, since the labels are deleted after all the constraints are processed. The effect of change can only be perceived during the next iteration. In split-level relaxation, we delete a label once it is determined that there is no support, and the change can be used by the next constraint during the same iteration.

3.5. Discussion

In this section we point out some of the advantages and disadvantages of this approach. Clearly the success of the method depends on the special-lengths and angles. If they are occluded, then the method may take a longer time than usual. During the process, it might lose all the labels in which case it would not recognize the object. Or it may reduce the label-sets of some nodes and leave the rest of the work to the backtracking procedure, which can be expensive.

However, if there are enough *special* distances and angles, then even if some are missing, the method will work very well. Our experience is that this will work fine, if we can reduce the label-sets of two or three nodes to about 2 or 3. Also, it is not unusual to find about 4 or 5 special distances and angles. If they are not enough, we could increase the threshold distance (for connection) to include more inter-feature vectors, thereby increasing the possibility of finding more special distances and angles. Of course, this increases the cost of computation, both for the model and the image. The positive side of this is the fact that, these computations can be done ahead of time, more than once if necessary to arrive at an optimal set of special distances and angles. Once this is done, it is a part of the model of the object and need not be recomputed.

Also, this takes care of a certain amount of error in both the model and the image. As pointed out before, the special distances have a range associated with them. So, if there is any error the initial seeding is not affected. This makes the system more robust.

4. Implementation and Results

Most of the above ideas were implemented in PSL (Portable Standard Lisp) [22]. The compiled code was run on a VAX-8600. The actual runtimes can be vastly improved if implemented on a lisp machine like the Symbolics 3600.

The system works in two phases. In the first phase, all the models are input and the threshold distance, special lengths and angles are determined and stored in the model structure. This is only a one-time cost and is similar to the training phase of Local Feature Focus [5]. The input is at a somewhat intermediate level of description. What is input to the program is a list of local features, their locations and the constraints between them. The different types of constraints used are *parallel*, *perpendicular*, *adjacent*, *longer-than*, etc.

In the second phase, an image description is given along with the set of constraints associated with the local features in it. We also give a model to be searched for in the image. We only check for node and arc consistency. The AC-3 algorithm as defined by Mackworth [17] is used for enforcing the consistency. Although, we look for a single object in an image, it would be be a straight-forward extension to look for multiple objects in the same image. We essentially follow the model given in section . The image description is similar to that for the models.

At the end of relaxation process, control is passed on to a backtracking procedure which does further consistency checks and lists out all the solutions. The output is a listing of local features in the image and the corresponding features in the object. If the label-set of a primitive is empty, it is labelled as *unknown*.

4.1. Unoccluded Object Recognition

We now present some results obtained using the algorithms described. First, we give the run-times for recognizing unoccluded objects. Next we give the same for some occluded scenes.

Figure 12 gives the time taken to recognize three parts shown in Figures 13, 14, 15 in unoccluded scenes. It also, gives the number of different types of constraints and total number of constraints used. It should be noted here, that the number of constraints and the constraint types are not optimized (See section 5 for further discussion). Here only the boundary edges are used for recognition.

Object No	RunTime (msec)	Constraint types	Total Number of constraints
1	357	5	43
2	476	6	33
3	85	4	15

Figure 12. Table showing results for unoccluded scenes

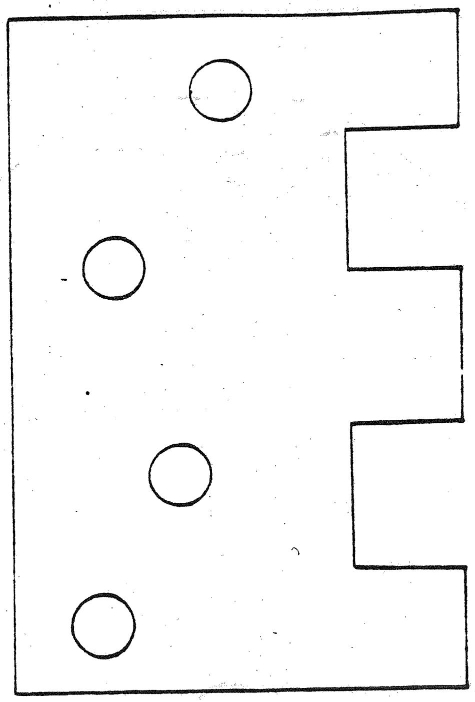

Figure 13. Object No 1

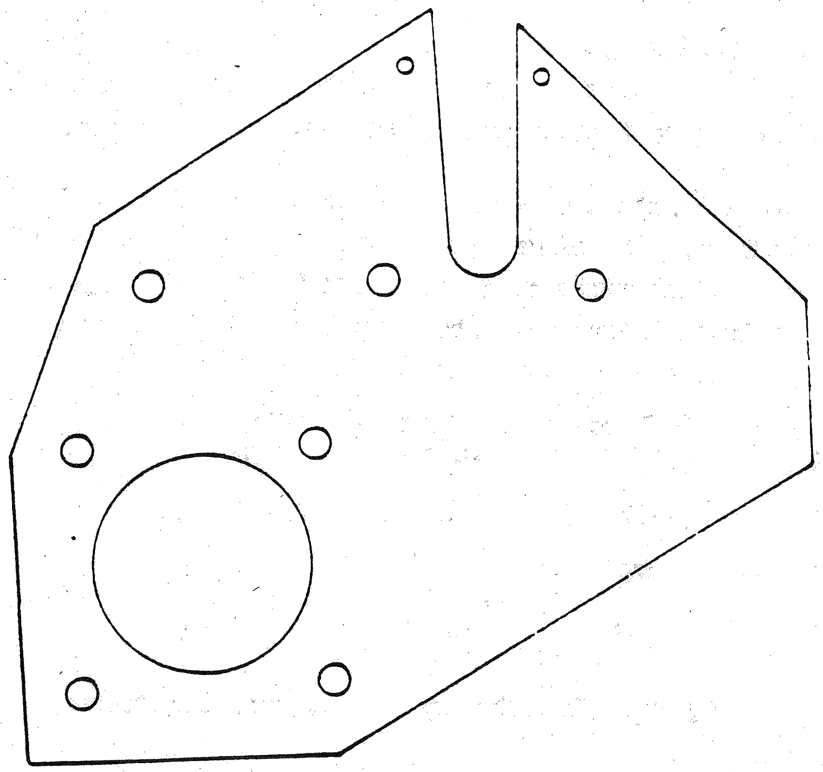
Figure 14. Object No 2

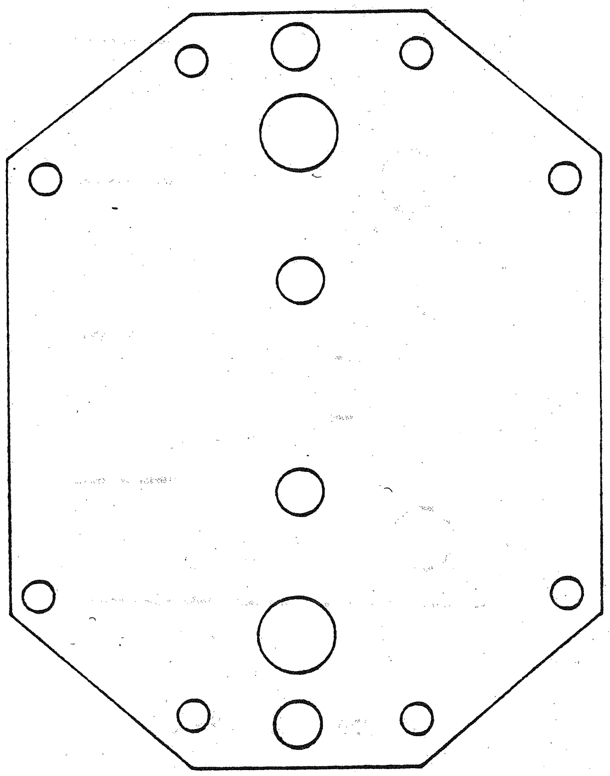
Figure 15. Object No 3

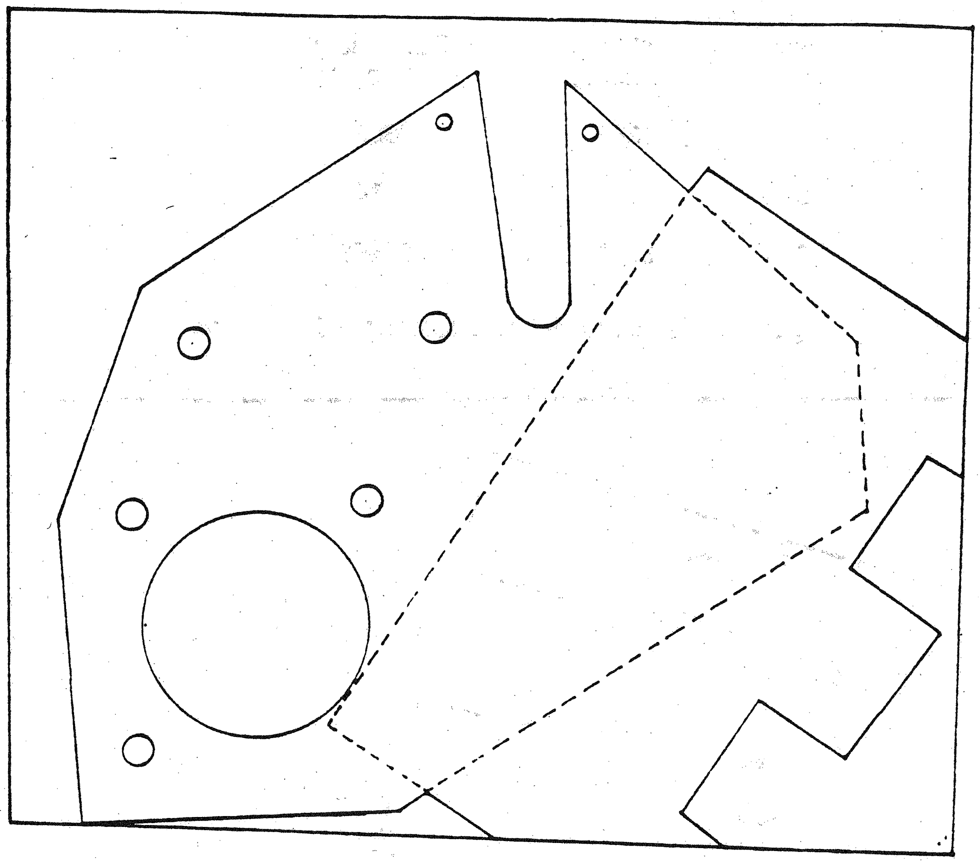

Figure 18. Occluded Scene No 2

4.2. Occluded Object Recognition

Next we present some results for occluded scenes. Figures 17, 18, 19 show three scenes where some parts are occluded. These scenes were searched for occurrence of objects 1, 2 and 3 respectively. Figure 16 gives the time taken to recognize these parts in the corresponding scene. There are two run times in the table, corresponding to the two schemes used (see section 3.4).

Scene No	RunTime(1) (msec)	RunTime(2) (msec)
1	1360	6307
2	1581	2074
3	3791	26452

Figure 16. Table showing results for occluded scenes

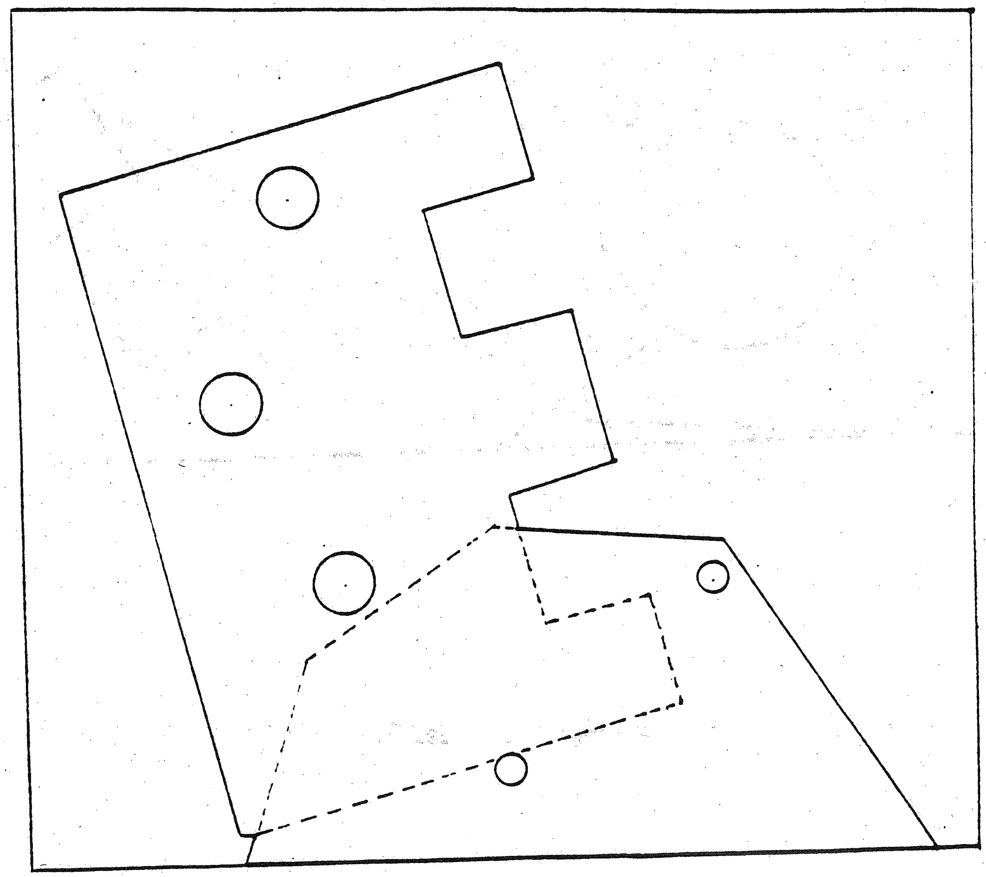

Figure 17. Occluded Scene No 1

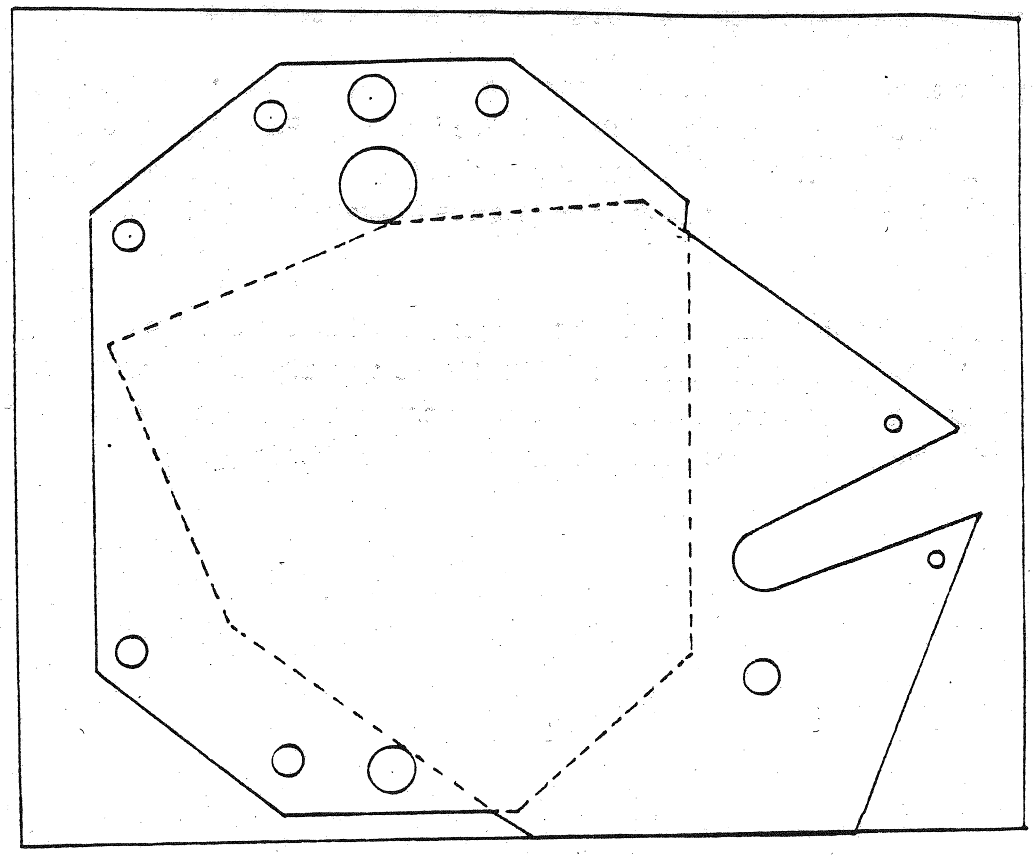

Figure 19. Occluded Scene No 3

5. Conclusions and Extensions

In this paper we have formulated the scene analysis (SA) problem as a consistent labelling problem (CLP). We also gave a solution to the problem within the framework of discrete relaxation methods. However, the approach is based on a different perspective in order to exploit the inherent parallelism in the problem and to account for occlusion. We have also recommended several variations to improve the efficiency of the computation. Also, unlike many other works we have used constraints of arity greater than 2. Infact, we effectively handle unary, binary, ternary and 4-ary constraints.

Although the approach sounds attractive, particularly if a multi-processor is accessible, it by no means solves all the problems. First, we are still dealing with 2-D models. Extending it to 3-D has its obvious challenges. Although we do take into account a certain amount of error, both in the models and during the low level processing, the error measurements are not based on any sound theory to analytically determine them.

We have also shown that global constraints can be useful and we can effectively use them with local constraints. Another problem we have not considered is the optimality of the constraint set. We have used the constraints which were obvious in the models and the image. However, some of them are redundant and there should be a systematic way to eleminate the redundancy.

References

[1] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman.
The Design and Analysis of Computer Algorithms.
Addision-Wesley Publishing Company, 1974.

[2] N. Ayeche and O. D. Faugeras.
A new method for the recognition and positioning of 2 D objects.
In *ICPR*, pages 1274-1280. August, 1984.

[3] Bir Bhanu, O. D. Faugeras.
Shape Matching of Two-Dimensional Objects.
IEEE Transactions On Pattern Analysis And Machine Intelligence PAMI-6(2):137-156, march, 1984.

[4] Robert C. Bolles.
Robust feature matching through maximal cliques.
Proc. SPIE Tech. Symp. Imaging Appl. Automated Industrial Inspect. Assembly, Belingham, Wash , 1979.

[5] Robert C. Bolles and Ronald A. Cain.
Recognizing and Locating Partially Visible Objects : The Local-Feature-Focus Method.
The International Journal of Robotics Research 1(3):57-82, Fall, 1982.

[6] Robert C. Bolles and Ronald A. Cain.
Recognizing and Locating Partially Visible Workpieces.
PRIP :498-503, June, 1982.

[7] Robert G. Busacker and Thomas L. Satty.
International Series in Pure and Applied Mathematics: Finite Graphs and Networks.
McGraw-Hill Book Company, 1965.

[8] Larry S. Davis and Thomas C. Henderson.
Hierarchical Constraint Processes for Shape Analysis.
IEEE Transactions On Pattern Analysis And Machine Intelligence PAMI-3(3):265-277, May, 1981.

[9] Faugeras, Olivier and Keith Price.
Semantic Description of Aerial Images Using Stochastic labeling.
IEEE Transactions on Pattern Analysis and Machine Intelligence :633-642, November, 1981.

[10] John Gaschnig.
Performance Measurement and Analysis of Certain Search Algorithms.
PhD thesis, Carnegie-Mellon University, May, 1979.

[11] Robert M. Haralick and Gordon Elliot.
Increasing Tree Search Efficiency for Constraint Satisfaction Problems.
Technical Report , Virginia Polytechnic Institute and State University, Department of Electrical Engineering, VPISU, Blacksburg, Virginia 24061, March, 1979.

[12] Robert M. Haralick, Larry S. Davis, Azriel Rosenfeld, and David Milgram.
Reduction Operations for Constraint Satisfaction.
Information Sciences 14():199-219, , 1978.

[13] Robert M. Haralick and Linda G. Shapiro.
The Consistent Labelling Problem: Part I.
IEEE Transactions On Pattern Analysis And Machine Intelligence PAMI-1(2):173-184, April, 1979.

[14] Robert M. Haralick and Linda G. Shapiro.
The Consistent Labelling Problem: Part II.
IEEE Transactions On Pattern Analysis And Machine Intelligence PAMI-2(3):193-203, May, 1980.

[15] Thomas C. Henderson and Larry S. Davis.
Hierarchical Models and Analysis of Shape.
Pattern Recognition 14(1-6):197-204, 1981.

[16] Donald E. Knuth.
Estimating the Efficiency of Backtrack Programs.
Mathematics of Computation 29(129):121-136, January, 1975.

[17] Alan K. Mackworth.
Consistency in Network of Relations.
Artificial Intelligence 8:99-118, 1977.

[18] Roger Mohr and Thomas C. Henderson.
Arc and Path Consistency Revisited.
Technical Report UUCS-85-101, The University of Utah, August, 1985.

[19] Ugo Montanari.
Networks of Constraints: Fundamental Properties and Applications to Picture Processing.
Information Sciences 7:95-132, 1974.

[20] D. Nitzan et al.
Machine Intelligence Research Applied to Industrial Autmation.
Technical Report, SRI International, January, 1982.

[21] Christos H. Papadimitriou and Mihalis Yannakakis.
The Clique Problem for Planer Graphs.
Information Processing Letters 13(4,5):131-33, End, 1981.

[22] The Utah Symbolic Computation Group.
The Portable Standard LISP users Manual
Department of Computer Science, 1983.

[23] Azriel Rosenfeld, Robert A. Hummel and Steven W. Zucker.
Scene Labelling by Relaxation Operations.
IEEE Transactions On Systems, Man, And Cybernetics SMC-6(6):420-433, June, 1976.

[24] David Waltz.
Understanding Line Drawings of Scenes with Shadows.
In Patrick Henry Winston (editor), *The Psychology of Computer Vision*, pages 19-92. McGraw-Hill Book Company, 1975.