A PROGRAMMING FRAMEWORK TO IMPLEMENT RULE-BASED TARGET DETECTION IN IMAGES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

YAVUZ ŞAHİN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

DECEMBER 2008

Approval of the thesis

"A PROGRAMMING FRAMEWORK TO IMPLEMENT RULE-BASED TARGET DETECTION IN IMAGES"

submitted by **Yavuz Şahin** in partial fullfillment of the requirements for the degree of **Master Of Science in Computer Engineering** by,

Prof. Dr. Canan Özgen
Dean, **Graduate School of Natural and Applied Sciences**

Prof. Dr. Müslim Bozyiğit
Head of Department, **Computer Engineering**

Prof. Dr. Volkan Atalay
Supervisor, **Computer Engineering, METU**

Examining Committee Members:

Prof. Dr. Neşe Yalabık
Computer Engineering, METU

Prof. Dr. Volkan Atalay
Computer Engineering, METU

Prof. Dr. Yasemin Yardımcı Çetin
Informatics Institute, METU

Assoc. Prof. Dr. Sibel Tarı
Computer Engineering, METU

Dr. Adem Mülayim
MilSOFT

Date:

ABSTRACT

A PROGRAMMING FRAMEWORK TO IMPLEMENT RULE-BASED TARGET DETECTION IN IMAGES

Şahin, Yavuz
M.Sc., Department of Computer Engineering
Supervisor: Prof. Dr. Volkan Atalay

December 2008, 55 pages

An expert system is useful when conventional programming techniques fall short of capturing human expert knowledge and making decisions using this information. In this study, we describe a framework for capturing expert knowledge under a decision tree form and this framework can be used for making decisions based on captured knowledge. The framework proposed in this study is generic and can be used to create domain specific expert systems for different problems. Features are created or processed by the nodes of decision tree and a final conclusion is reached for each feature. Framework supplies 3 types of nodes to construct a decision tree. First type is the decision node, which guides the search path with its answers. Second type is the operator node, which creates new features using the inputs. Last type of node is the end node, which corresponds to a conclusion about a feature. Once the nodes of the tree are developed, then user can interactively create the decision tree and run the supplied inference engine to collect the result on a specific problem. The framework proposed is experimented with two case studies; *"Airport Runway Detection in High Resolution Satellite Images"* and *"Urban Area Detection in High Resolution Satellite Images"*. In these studies linear features are used for structural decisions and Scale Invariant Feature Transform (SIFT) features are used for testing existence of man made structures.

Keywords: airport runway detection, urban area detection, expert systems, rule-based classification, decision tree, Scale Invariant Feature Transform

ÖZ

GÖRÜNTÜLERDE KURALA DAYALI HEDEF TANIMLAMA UYGULAMALARI İÇİN BİR PROGRAMLAMA ALTYAPISI

Şahin, Yavuz
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Volkan Atalay

Aralık 2008, 55 sayfa

Bir konu ile ilgili uzmanların bilgilerinin değerlendirilmesi ve karar vermede, bu uzmanların yerini alma veya onlara yardımcı olma amaçlı olarak uzman sistemlerin geleneksel yazılımlara göre daha faydalı oldukları görülmektedir. Bu çalışmada uzmanların bilgilerinin karar ağacı yapısında toplanıp saklanması ve bu bilginin kullanılarak kararlara ulaşılması için bir çerçeve sunulmuştur. Burada sunulan çerçeve bir alana özel olmamakla birlikte alana özel karar ağaçlarının oluşturulması için kullanılabilmektedir. Karar ağaçlarının boğumları tarafından öznitelikler oluşturulur ve kullanılırlar. Her öznitelik için bir karara varılır. Bu amaçla çerçevemizde 3 çeşit boğum bulunmaktadır. Birinci çeşit, karar boğumudur. Bu boğum, verdiği yanıtlarla karar ağacındaki aramayı yönlendirir. İkinci çeşit, işlem boğumudur. Bu boğum, girdilerini kullanarak yeni öznitelikler oluşturur. Son çeşit ise sonlandırma boğumudur. Bu boğum, bir öznitelik için karara varıldığını gösterir. Karar ağacının boğumları bir defa oluşturulduktan sonra kullanıcı bu bileşenleri kullanarak karar ağacını interaktif olarak oluşturur ve çıkarsama modülü yardımı ile problemine yanıt arar. Burada oluşturulan çerçeve iki çalışma ile denenmiştir; *"Yüksek Çözünürlüklü Uydu Görüntülerinde Uçak Pisti Bulunması"* ve *"Yüksek Çözünürlüklü Uydu Görüntülerinde Kentsel Yerleşim Alanlarının Bulunması"*. Bu çalışmalarda şekillerin çıkarılması için doğrusal öznitelikler ve insan yapımı nesnelerin tespiti için ölçekten bağımsız nitelik dönüşümü ile elde edilen öznitelikler kullanılmaktadır.

Anahtar Kelimeler: uçak pisti bulma, kentsel alanların bulunması, uzman sistemler, kurala dayalı sınıflandırma, karar ağacı, skala bağımsız nitelik dönüşümü

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor Prof. Dr. Volkan ATALAY, without whose guidance and support this work could not be accomplished.

I deeply thank to Dr. Adem Y. Mülayim for his invaluable suggestions throughout the development of the thesis and his encouragement to finish my study.

I deeply thank my wife for her understanding, support, and for sharing hard times of my work.

To my parents and my beloved wife

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

AI	Artificial Intelligence
DoG	Difference Of Gaussian
IDE	Integrated Development Environment
SIFT	Scale-Invariant Feature Transform

CHAPTER 1

INTRODUCTION

1.1 Motivation and Problem Definition

With the recent advances in the technology, today's satellites are capable of acquiring high resolution images up to a resolution of 41 *cm* [11] or even more than this. This improvement has two opposite effects on the classification of satellite image content and detection of possible targets. The first effect is that those tasks are getting easier and more accurate because of the high resolution which provides more information about the content. On the other hand, the amount of data to be processed grows exponentially which means it takes more time for an expert to analyze image content; therefore number of experts on target detection should be increased or some or all of their tasks should be automated.

For the automation of such tasks a lot of classification and target detection systems have been designed for different problems. Although these systems attacked different problems, many atomic subproblems solved in these systems are common. For example, calculating statistics on linear features can be a part of both urban area detection and road detection. Therefore, it is useful to provide a way to reuse solutions to such subproblems across different problems. Furthermore, if we look at the flow of development activities in a traditional automated system, which is shown in Figure 1.1, we see that human expert and the final software are disconnected from each other, which may result in the dissatisfaction of human expert.

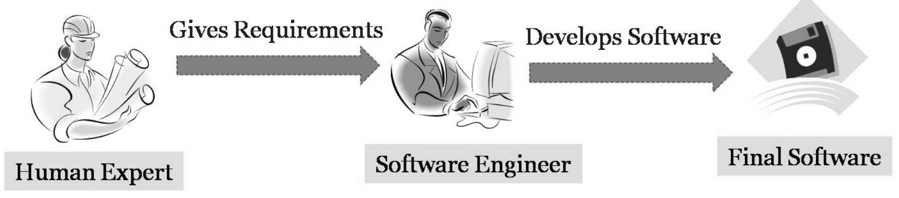


Figure 1.1: Traditional development flow of an automated system.

In this study, we aim to attack the situations described above. First of all, since the need of the automated systems increased, we aimed to provide a framework that will speed up the development process of automated systems. Furthermore, this framework should encourage re-use of solutions to commonly faced problems and let developers concentrate on specific problems of the automated system. Finally, involvement of human expert on the automated system development process should be increased such that this increase in involvement may result in more satisfaction of user.

1.2 Proposed Solution

For the automation of detection and classification problems mentioned in Section 1.1 an expert system can be used. An expert system is a system that attempts to mimic the human behavior during decision making process. It captures the human expert knowledge and attempts to make a decision about a problem based on this knowledge.

In this work, an expert systems framework to implement rule-based target detection is proposed to be employed in development process of automatic classification and target detection problems on satellite images and to provide a way to encourage reuse of solutions to sub-problems. The proposed framework employs a decision tree for knowledge representation and we developed an inference engine to work on this tree. In this decision tree, features are created or processed by nodes, and a decision is made for each feature. A *decision*, an *operator* or an *end* node developed in accordance with the supplied interfaces can be plugged into our system on runtime and can be used immediately for capturing expert knowledge. A *decision* node represents a choice and guides the search path with this choice. An *operator* node produces new features from its inputs and an *end* node corresponds to a decision about a feature. An expert system is domain specific by its nature, but our system provides a generic framework to be used across domains. On the other hand the final decision tree is domain specific.

In order to employ our system, after collecting requirements from the human expert, problem specific nodes are developed. Then the expert knowledge is input throughout the interfaces supplied by framework and nodes. This flow is shown in Figure 1.2. Once the decision tree for the specific problem is constructed, then this tree can be run in order to achieve a result. The inference engine starts with the root node of the decision tree, then each feature generated by an operator node is applied down to this node, path defined by each decision is followed and results are gathered at end nodes.

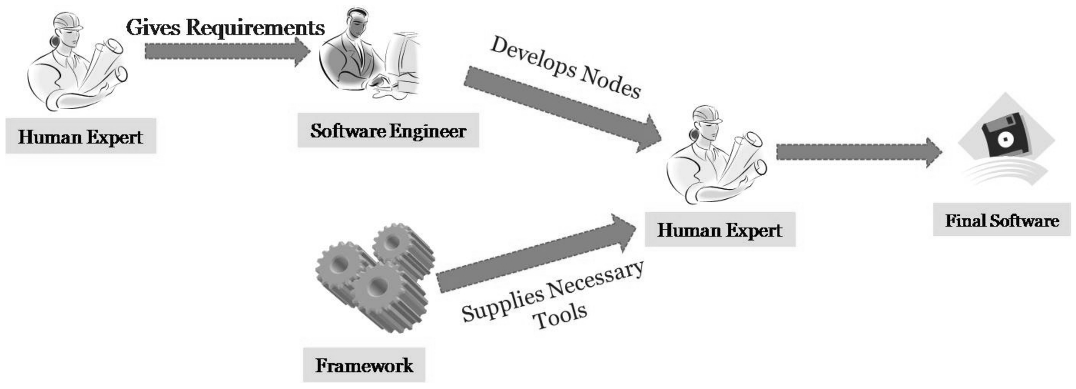


Figure 1.2: Development flow of an automated system using proposed framework.

The proposed framework is tested with two case studies as explained below;

- ***"Airport Runway Detection in High Resolution Satellite Images"***: In this case study, nodes to be used in decision tree are designed and implemented. Most of these nodes are atomic rules so they can be reused in other target detection tasks, such as road detection. First linear features are extracted from image, then parallel line pairs are formed based on linear features. A runway should have a clean area around it for safe operation; therefore SIFT [9] features are used for deciding if there are man made structures around the parallel lines or not. At last, parallel lines accepted through these steps are classified as non-runway, liaison runway, surveillance runway, light-lift runway, medium-lift runway and tactical runway based on their widths and lengths. This classification can be extended to include for example roads, highways etc. by adding proper nodes to the tree.

- ***"Urban Area Detection in High Resolution Satellite Images"***: In this case study, decision tree nodes for urban area detection are designed and implemented. Input image is divided into sub-regions, linear features and SIFT [9] features are calculated for each sub-region. Finally, each sub-region is classified to be urban area or not according to those features.

1.3 Improvements and Contribution

A pluggable expert systems framework is proposed such that work of a developer is decreased down to just implementing specific problem nodes since inference engine and the user interfaces are already supplied. It can be seen that once a good library of nodes is achieved then need for development of such nodes are minimized. In addition, when a library of re-usable nodes is built then node developer can concentrate on developing only the very specific nodes of the problem. Here emphasis is on the re-usability of nodes which is the art of designer. Furthermore, we demonstrate the execution of the system on two applications.
On the image processing side, although SIFT is mainly a method for object recognition and matching in different images, in this work, SIFT is used to test existence of man-made structures in a specified area, and the results points to its success.

1.4 Organization of the Manuscript

The organization of this manuscript is as follows. In Chapter 2, background information on concepts applied in this thesis is given. In Chapter 3, details of framework software for building a classification tree is presented. A case study of runway detection is given in Chapter 4 and a case study of urban area detection is given in Chapter 5. The manuscript is concluded with Chapter 6 where the presented study is discussed.

CHAPTER 2

BACKGROUND INFORMATION

2.1 Expert Systems and Classification Trees

2.1.1 Rule Based Expert Systems

Expert systems is a subfield of *Artificial Intelligence (AI)*, that is concerned with the emulation of human cognitive skills (such as problem solving, visual perception). Conventional computer programs use well-structured algorithms and data structures in order to solve problems, but for more complex problems it proves to be useful to introduce human expert knowledge. An expert system is a software system that tries to mimic the human expertise in decision making process in order to achieve desired decision on some specific domain. An expert system can fulfill the job of a human expert or just help him in making the decision [8]. Human knowledge is captured throughout a knowledge acquisition process and stored in a knowledge-base with a proper representation. Knowledge can be represented in different forms such as frames, *if-then* rules or networks. The main goal of knowledge representation is to store knowledge in a form that can be processed by a software, *inference engine*. This stored knowledge is processed by the inference engine and then presented to users through a user interface. Architecture of a simple expert system composed of parts recently mentioned is given in Figure 2.1.

A rule based expert system makes reasoning on knowledge-base where rules are used for knowledge representation. A rule is composed of two parts and can be written as, in its simplest form, '*if <condition> then <conclusion>*'. Where the *then* part of the rule is true only if the *if* part is true. Rules are written in a language closed to the speaking language in order to enable people from different backgrounds to define rules. In conventional programming approaches procedure for solving a problem should be given completely but in rule based expert systems an expert gives only what he knows about the problem and inference

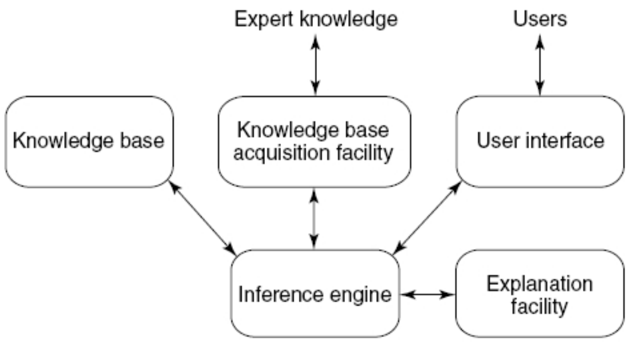


Figure 2.1: Architecture of a simple expert system. *Adapted from [1]*

engine would find out how to apply this knowledge.
Rules can be precise like '*if age < 18 then can have a driving license*'. On the other hand, a rule can also be written as '*if person is young then can run fast*'. In this latter approach measurement of *being young* and *running fast* is not precise, fuzzy rule based expert systems are used for inferring such rules.

2.1.2 Classification Trees

A decision tree is a knowledge representation technique that can be used in a decision support process for achieving a result using known facts[10]. A decision tree represents data with a tree model. This tree model is useful for 1) explaining the structure of data in a human readable form, 2) providing data for the inference engine such that different heuristics, search methods such as breadth-first, depth-first search can be applied. A classification tree is a decision tree with discrete outcome and a regression tree is a decision tree with continuous outcome.[13] A classification tree is built using binary recursive partitioning, which is a divide-and-conquer method that divides the input into pieces and applies further division on these pieces until a solution is reached.

2.1.3 Tree Traversal Methods

A sample tree structure is given in Figure 2.2. Different traversal methods can be applied to such a tree as explained below.

2.1.3.1 Breadth-First Traversal

This method is based on the principle that, a sibling of the current node is processed only if the all subtree of current node is traversed [2]. Preorder and postorder traversals are the types

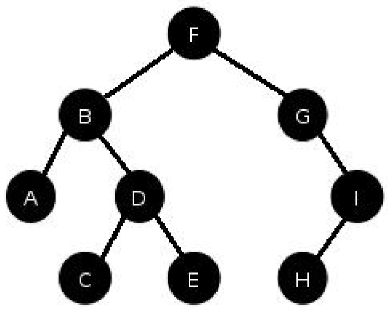


Figure 2.2: A sample tree.

of breadth-first traversal and are explained in Algorithm 1 and Algorithm 2 respectively. The traversal order of nodes for the tree given in Figure 2.2 is as follows.

- **Preorder:** F, B, A, D, C, E, G, I, H
- **Postorder:** A, C, E, D, B, H, I, G, F

Algorithm 1 Preorder Traversal of a Tree

```
1: Visit the node
2: for all children child belonging to node do
3:     Preorder traverse child
4: end for
```

Algorithm 2 Postorder Traversal of a Tree

```
1: for all children child belonging to node do
2:     Postorder traverse child
3: end for
4: Visit the node
```

2.1.3.2 Depth-First traversal

In depth-first traversal levels are visited successively starting from the root level and all nodes of a level are visited before going to next level [2]. Result of depth-first traversal order of the tree in Figure 2.2 is F, B, G, A, D, I, C, E, H. The depth-first traversal algorithm, which is based on the use of queue data structure, is given in Algorithm 3.

Algorithm 3 Depth-First Traversal

```
1: queue = empty queue
2: queue.enqueue( root )
3: while queue is not empty do
4:   node = queue.dequeue
5:   visit node
6:   for all children child in node do
7:     queue.enqueue( child )
8:   end for
9: end while
```

2.2 Image Analysis Methods

2.2.1 Canny Edge Detection Method

In an image, edges are characterized by the high contrast changes between pixels. Applying edge detection to an image may significantly reduce the amount of data to be processed and may therefore filter out information that may be regarded as less relevant, while preserving the important structural properties of an image. In this study edge detection is used as the first step of extracting straight lines in an image and Canny edge detection method [4] is selected.

In Canny edge detection method, first image is convolved with a Gaussian in order to reduce noise, then intensity gradient of the image is calculated with some gradient operator (for example Sobel [4]). Subsequently, non-maximum suppression is applied in order to find edge locations and at last, those edges are traced throughout the image applying hysteresis thresholding.

This method is presented in Algorithm 4 and result of applying this method on the image given in Figure 2.3 is shown in Figure 2.4. Gaussian scale and low/high thresholds of hysteresis thresholding are parameters of this algorithm.

Algorithm 4 Canny Edge Detection (*Adopted from [11]*)

```
1: Convolve Image $I$ with a Gaussian of scale $\sigma$
2: for all pixels $p$ in $I$ do
3:     Find edge magnitude and edge direction using Sobel operator
4: end for
5: for each pixel $p$ with non-zero edge magnitude do
6:     inspect two adjacent pixels along the edge direction
7:     if edge magnitude of one these is greater than edge magnitude of $p$ then
8:         mark edge pixel $p$ for deletion
9:     end if
10: end for
11: Apply hysteresis thresholding with $t_{low}$ and $t_{high}$
```

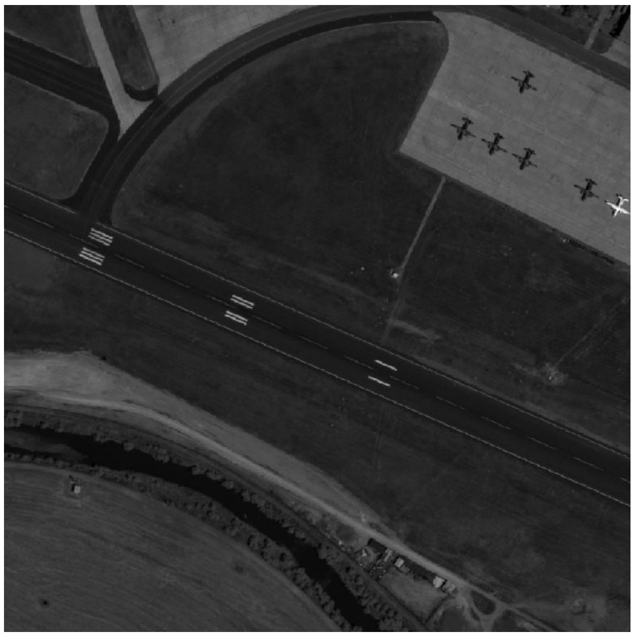


Figure 2.3: Image sample from an airport runway.

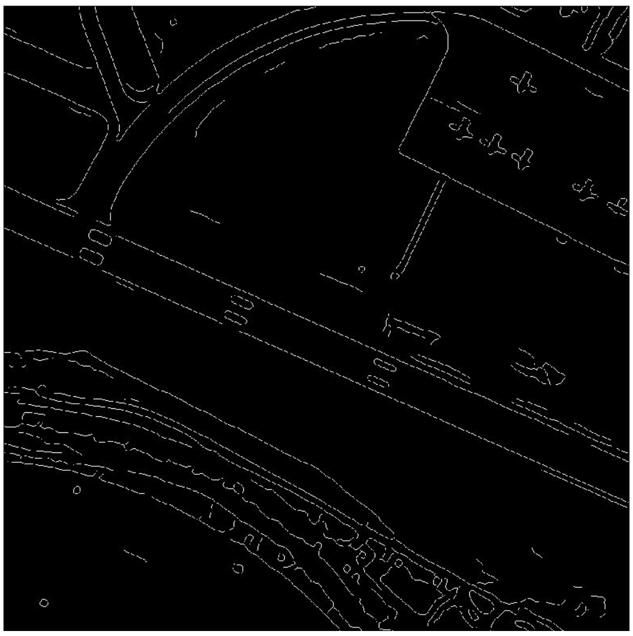


Figure 2.4: Result of applying Canny edge detection on Figure 2.3.

2.2.2 Connected Component Labeling

Once the binary edge image is obtained, connected component labeling algorithm is used to identify the set of pixels that belong to same edge based on the pixel connectivity. Pixel connectivity is described by the needs of the algorithm and the space, and for a two dimensional image it can be 4-connected (Table 2.1) or 8-connected (Table 2.2).

	X	
X	P	X
	X	

Table 2.1: 4-connected

X	X	X
X	P	X
X	X	X

Table 2.2: 8-connected

Connected component labeling algorithm is given in Algorithm 5. The algorithm makes two passes over the input data producing temporary labels and equivalence classes in the first pass and replacing each temporary label with its equivalence class in the second pass.

Algorithm 5 Connected Component Labeling (*Adopted from [11]*)

```
Require: binary edge Image $I$
1: for all pixels $p$ in $I$ do
2:   if $p == 1$ then
3:     if all of neighbor pixels $== 0$ then
4:       assign a new label to $p$
5:     else if only one neighbor $== 1$ then
6:       assign neighbor's label to $p$
7:     else
8:       assign one of the neighbors' label to $p$ and make a note of equivalences
9:     end if
10:  end if
11: end for
12: Assign a unique label for each equivalence class
13: for all pixels $p$ in $I$ do
14:   if $p$ has a label then
15:     replace its label by the label assigned to its equivalence class
16:   end if
17: end for
```

2.2.3 Boundary Curve Segmentation By Polygonal Approximation

After labeling connected pixels in the image a sequence of linked pixels that form a curve is obtained. At this point, our aim is to obtain linear segments from these curves. For this purpose three important points of the curve are detected: A and B are two end points of the curve and third point, C, is the point with maximum perpendicular distance from line connecting the two end points. If the maximum perpendicular distance from C to line connecting the two end points is greater than some specified threshold then curve is split into two parts, from A to C and from C to B as shown in Figure 2.5.a. The same process is repeated recursively with these two parts as shown in Figure 2.5.b. On the other hand if the maximum perpendicular distance from C to line connecting the two end points is smaller than the specified threshold, then this curve is accepted as a single linear segment.

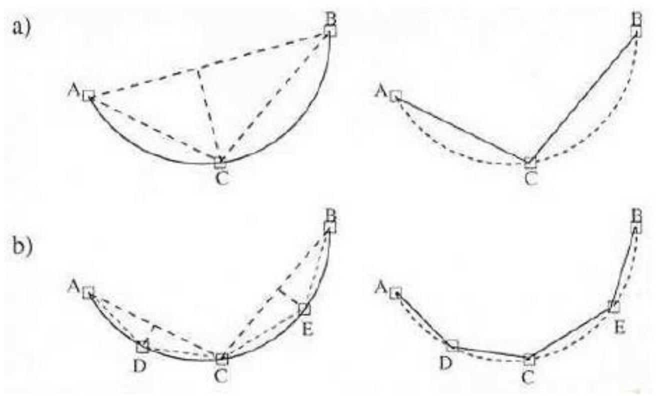


Figure 2.5: a) The algorithm splits the input curve from C and creates two segments AC, CB. b) The algorithm continues with the new line segments created in a). (*Adopted from [3]*)

Result of applying boundary curve segmentation on the edge image of Figure 2.4 is shown in Figure 2.6.

2.2.4 Connecting Collinear Lines

As seen in Figure 2.6, boundary curve segmentation can result in disconnected collinear line segments. The primary reason of such disconnected line segments is the geometry of the shape we are looking for. For example in Figure 2.6, there exists disconnected collinear lines which are in fact part of the runway edge, but divided into parts by an intersecting taxiway or discontinuities in edge. Therefore, this will be a common problem that we have to overcome often. For this purpose a modified version of technique proposed by Jun Wei Han, Lei Guo and YongSheng Bao in [6] is applied. The referenced method compares the length between far ends of lines to the sum of lengths of lines and length between near ends of lines. In our method, each linear segment is compared to all other linear segments, and two lines are accepted to be collinear if their slopes are nearly equal, their perpendicular distance to each other, $d1$ as shown in Figure 2.7, is relatively small and shortest distance between these two lines, $d2$ as shown in Figure 2.7, is not too big. The method is explained in Algorithm 6 and result of applying this method on lines of Figure 2.6 is given in Figure 2.8.

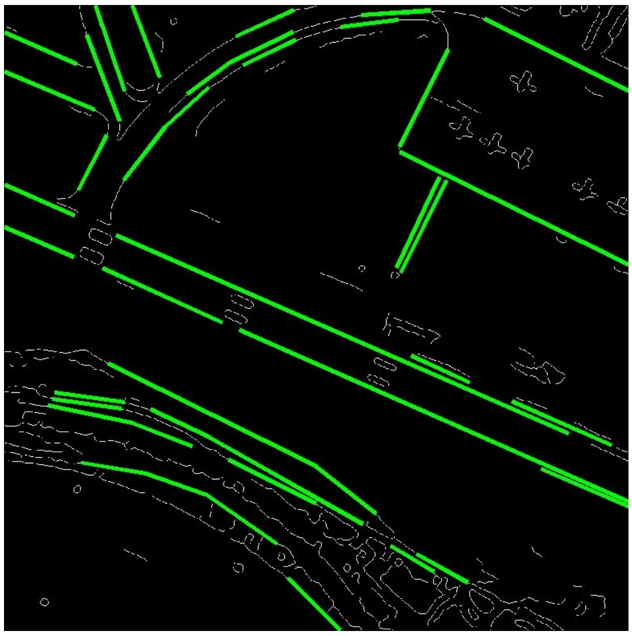


Figure 2.6: Result of applying boundary curve segmentation on Figure 2.4.

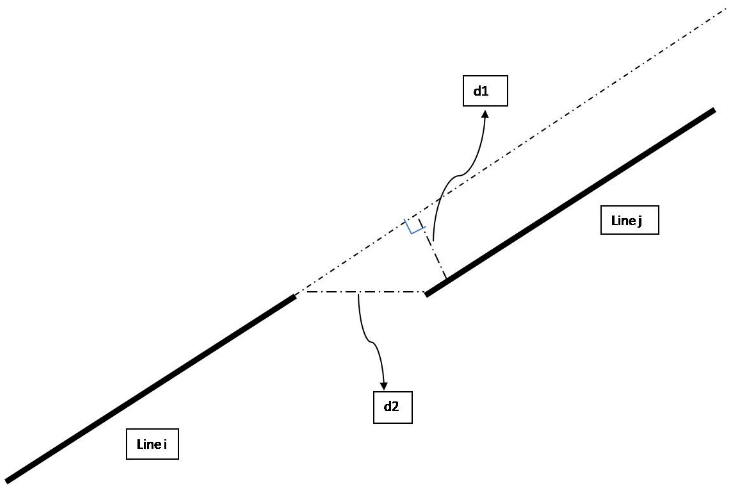


Figure 2.7: Definitions on Disconnected Lines.

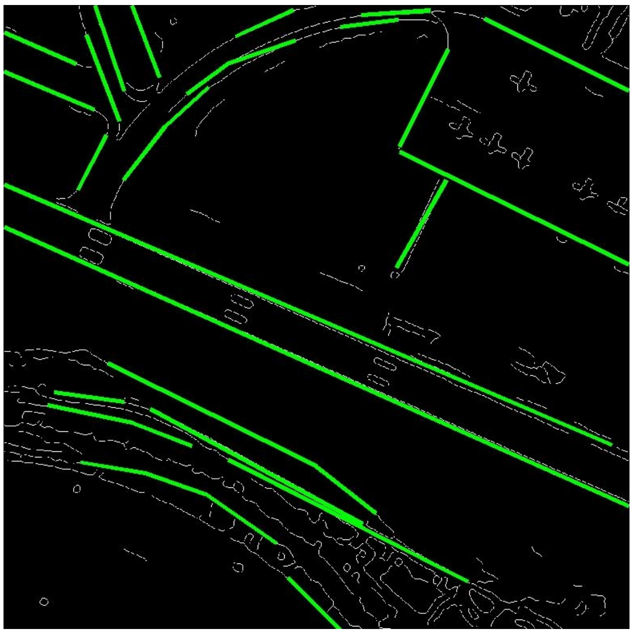


Figure 2.8: Result of connecting collinear lines of Figure 2.6.

Algorithm 6 Connecting Collinear Lines. T_p is the threshold for perpendicular distance between lines and T_s is the shortest distance between lines.

```
1:  for all line $i$ in image do
2:    for all line $j$ in image where $i \neq j$ do
3:      if slope of $i \approxeq$ slope of $j$ then
4:        if $(d_1 \leq T_p)$ AND $(d_2 \leq T_s)$ then
5:          connect $i$ and $j$
6:          remove $j$
7:        end if
8:      end if
9:    end for
10: end for
```

2.2.5 Scale-Invariant Feature Transform (SIFT)

Scale-Invariant Feature Transform [9] is a method to extract and describe local features in images. The local features generated by this method are robust to rotation, scale, illumination, noise changes in the image and minor changes in viewpoint. Because of these properties SIFT is very helpful in recognizing known objects in images. In this work, SIFT is used to decide about the lateral clearance of runways, since it generates local features which primarily stem from distinctive shapes on the image. SIFT can be described in 4 main steps:

1. Scale-Space Extrema Detection
2. Keypoint Localization
3. Orientation Assignment
4. Keypoint Descriptor

Details of these steps are given below.

2.2.5.1 Scale-space Extrema Detection

In this step, stable features are searched among possible scale values in order to identify feature points that are stable among different scales. For this purpose input image $I(x, y)$ is successively convolved with a Gaussian $G(x, y, \sigma)$ kernel and scale space function $L(x, y, \sigma)$ is obtained.

$$L(x, y, \sigma) = G(x, y, \sigma) \ast I(x, y) \tag{2.1}$$

where $\ast$ is the convolution operator in x and y and

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{(x^2+y^2)/2\sigma^2} \tag{2.2}$$

Then Difference-of-Gaussian(DoG) $D(x, y, \sigma)$ of consecutive scales are calculated.

$$\begin{aligned} D(x, y, \sigma) &= (G(x, y, k\sigma) - G(x, y, \sigma)) \ast I(x, y) \\ &= L(x, y, k\sigma) - L(x, y, \sigma) \end{aligned} \tag{2.3}$$

Once DoG for different scales are obtained then each pixel in one scale is compared with its 8 neighborhood in the same scale and 9 neighborhoods in each of the neighboring scales. A pixel is accepted as a candidate keypoint if it is the maximum or minimum of this comparison.

2.2.5.2 Keypoint Localization

Scale-space extrema detection step may produce many unstable candidate points. In keypoint localization step those keypoints that have low contrast and that are poorly localized along the edges are rejected.

First, nearby data is interpolated using the shifted Taylor expansion of the $D(x, y, \sigma)$ where sample point is at the origin.

$$D(x) = D + \frac{\partial D^T}{\partial x} x + \frac{1}{2} x^T \frac{\partial^2 D}{\partial x^2} x \tag{2.4}$$

where D is the derivative calculated at the sample point and $x = (x, y, \sigma)^T$ is the offset from this point. Taking derivative of Equation 2.4 with respect to x and setting it to 0 yields the location of the extremum $\hat{x}$. If the location of $\hat{x}$ is greater than 0.5 in any of the directions then this sample point is considered to be closer to another keypoint so it is changed with another point and interpolation goes on. Otherwise, the offset is added to location estimation of candidate point and a new estimation is obtained.

Second, low contrast candidate keypoints are discarded by comparing the result of $D(\hat{x})$ to 0.03. If the value is less than 0.03, this keypoint is discarded.

Finally, edge responses are eliminated in order to reject unstable candidate keypoints among edges, because edge points give strong responses to DoG function but have poorly defined locations.

For poorly defined peaks in DoG function, principle curvature along edges will be small whereas the same curvature is large in perpendicular direction to edges. The principle curvatures are computed with a 2×2 Hessian Matrix H, defined by:

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \tag{2.5}$$

The eigenvalues of H are proportional to the principal curvatures of D. Let α be the eigenvalue with the largest magnitude and β be the one with smallest magnitude. Then the trace of H, $Tr(H) = D_{xx} + D_{yy}$,yields the sum of the eigenvalues and determinant of H, $Det(H) = D_{xx}D_{yy} - D_{xy}^2$, is the product of eigenvalues. Let's say there exists a ratio k between α and β where $\alpha = k\beta$ then one can show that,

$$\frac{Tr(H)^2}{Det(H)} = \frac{(k+1)^2}{k} \tag{2.6}$$

This ratio is a monotonically increasing function of k.

A threshold k_t is selected and Equation 2.7 is checked for each candidate point using this k_t.

If a candidate point does not obey Equation 2.7, then this point is rejected.

$$\frac{Tr(H)^2}{Det(H)} < \frac{(k_t+1)^2}{k_t} \tag{2.7}$$

2.2.5.3 Orientation Assignment

This step assigns orientations to keypoints based on local image gradient directions at each scale and provides invariance to rotation. For a sample image $L(x, y)$ at scale σ, gradient magnitude, $m(x, y)$, and gradient orientation, $\theta(x, y)$, is calculated using Equation 2.8.

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \tan^{-1}\left(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}\right) \tag{2.8}$$

After those values are calculated for each $L(x, y, \sigma)$, an orientation histogram consisting of 36 bins, i.e. one bin for 10 degrees, is constructed. Each sample in the neighboring window added to a histogram bin is weighted by its gradient magnitude and by a Gaussian-weighted circular window with a σ that is 1.5 times that of the scale of the keypoint. The highest peak in the histogram is found, and the peaks that lie within the 80% of this highest peak are also used to create new keypoints in the same scale and location but with different orientations.

2.2.5.4 Keypoint Descriptors

So far keypoints invariant to the scale, translation and rotation are calculated. In this step the purpose is to calculate descriptor vectors for the keypoints such that these descriptors are invariant to illumination, viewpoint etc. The feature descriptor is computed as the orientation histograms of 4x4 pixel neighbors consisting of 8 bins. Contribution of each pixel is calculated in the same manner as in orientation assignment step.

CHAPTER 3

A SOFTWARE TO BUILD CLASSIFICATION TREE

The aim of the software is to provide a framework for developing necessary tools in order to make classification using a classification tree, and providing an interface for the developer to plug these tools into the classification tree on the runtime. For this purpose the developer should develop tools that comply with specific interfaces. Then developer can plug those tools into the classification tree.

3.1 Requirements

Before developing the software, requirements are analyzed and gathered requirements are divided into two main parts. Functional requirements define the behavior of a system, whereas non-functional requirements define the properties of the system. **Shall** implies a must requirement and **should** implies an optional requirement.

3.1.1 Functional Requirements

- Framework shall provide *decision* nodes.
- Framework shall provide *operator* nodes.
- Framework shall provide *end* nodes.
- Framework shall provide *default* nodes.
- Framework shall provide *feature* structures to be passed among nodes.
- Each node should provide its own user interface.

- Each node shall respond to queries about its validity.
- A *decision* node shall accept a *feature* and return the index of child with which the search will go on.
- A *decision* node shall not be the leaf in a valid tree.
- An *operator* node shall accept a *feature* and return a new set of *feature*s.
- An *operator* node shall not be the leaf in a valid tree.
- An *end* node shall only be at the leaves of a valid tree.
- An *end* node shall imply that search is ended for a specific *feature*.
- An *end* node should display its output.
- A *feature* should be just an abstraction over the data to be passed among nodes and shall not put any constraints.
- Each node shall know the type of the input *feature* and cast it to proper type.
- Application shall start with an initial tree containing a *default* node as the root.
- A *default* node shall enable user to change itself with a user defined node on runtime.
- A *decision* node shall be able to proceed an *operator* node or another *decision* node.
- An *operator* node shall be able to proceed a *decision* node or another *operator* node.
- Each node shall be able to serialize its content.
- Each node shall be able to load serialized content.
- Each node shall be able create a copy of itself.
- Application shall run classification by gathering *feature*s from *operator* nodes and querying *decision* nodes with those features.
- Application shall reject incomplete trees, i.e. trees with nodes other than *end* nodes at leaves.
- Application shall reject invalid trees, i.e. trees with invalid nodes, this can happen when all necessary information is not provided for a node.

- Classification shall end when a decision is reached about every feature.
- Application shall be able to serialize a defined tree to disk.
- Application shall be able to load serialized tree from disk.
- Application shall be able to use breadth-first and depth first search heuristics.
- Application should provide tree editing capabilities such as *cut/copy/paste/delete* for easier development of classification tree.

3.1.2 Non-Functional Requirements

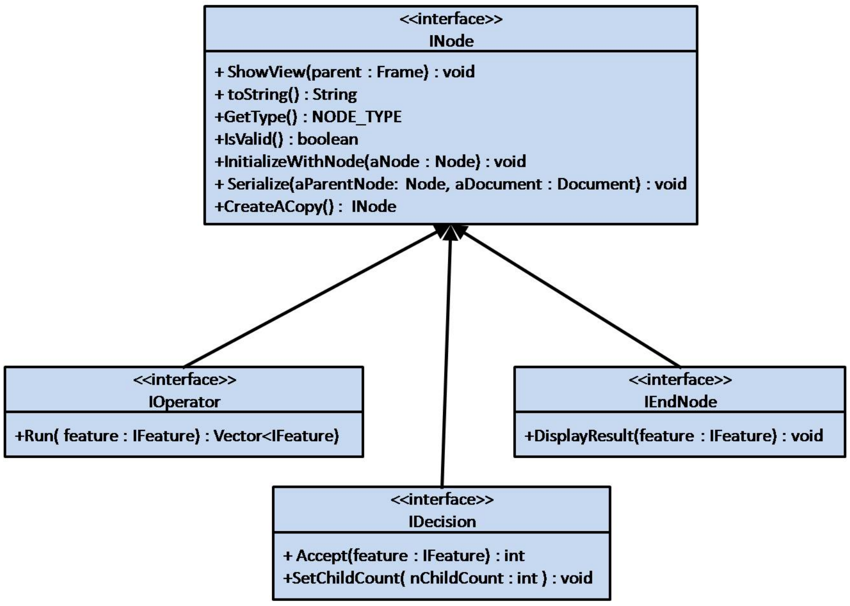
- Software should be open-source.
- Software should be cross-platform.
- Software should be easily installable.

3.2 Software Design and Implementation

The software is developed using *Java* as the programming language. There are a couple of reasons for this selection:

- *Java* is an object-oriented language [7].
- *Java* is a cross-platform language [7] that enables your programs to be written once and deployed on different operating systems. In addition, another cross-platform tool, *Eclipse*, is used as the integrated development environment (IDE). With this configuration, the software can be implemented, run and maintained in different platforms supporting Java, that is nearly all non real-time operating systems and a couple of real-time ones.
- *Java* provides *reflection* mechanism [7], which enables developers to examine or modify the structure of applications on the fly. Using this mechanism one can create a new instance of a class only by knowing its name. This feature is used when plugging in user defined nodes to our tree.
- *Java* applications are easily installable, such that most of the times a *.jar* pack of your program can be run in other machines having *Java Virtual Machine* by just double clicking.

Three interfaces, *IDecision, IOperator, IEndNode*, which are shown in Figure 3.1, are provided. A developer who wants to create a plug-in to our system should **extend** one of these interfaces properly. After developing a node complying these interfaces, it can be used as a *decision, operator* or *end* node in our system.



<<interface>>
INode
+ ShowView(parent : Frame) : void
+ toString() : String
+GetType() : NODE_TYPE
+IsValid() : boolean
+InitializeWithNode(aNode : Node) : void
+ Serialize(aParentNode: Node, aDocument : Document) : void
+CreateACopy() : INode
<<interface>>
IOperator
+Run(feature : IFeature) : Vector<IFeature>
<<interface>>
IEndNode
+DisplayResult(feature : IFeature) : void
<<interface>>
IDecision
+ Accept(feature : IFeature) : int
+SetChildCount(nChildCount : int) : void

Figure 3.1: A plug-in developer should extend one of the three interfaces, *IDecision, IOperator, IEndNode* .

The tree constructed throughout the system operation is serialized to XML files. XML is an industry standard data representation used in many types of applications and its structure is very convenient to store tree data. For this purpose methods are added to interfaces described above and node implementor is expected to fill these methods properly.
The software is composed of two main packages:

1. **mainframe:** This package is the main controller of the program and acts as the inference engine and knowledge-acquisition facility, shown in Figure 2.1.

2. **tree:** This package is the knowledge-base of our system. Knowledge gathered in knowledge-acquisition facility is stored here in the form of a tree and also presented to the user in a tree diagram.

Although root node can be any type of node, in practice it is meaningful to chose an *operator* node as the root node. Such an *operator* node can gather user input and produce

initial features. A feature is any information produced about the image being processed. The inference engine starts with the root node and an empty feature, and proceeds through its path defined by the operators and decisions. If the current node is a *decision* node, then inference engine asks for the decision of this node and *decision* node directs the search by returning one of its children as the answer, search continues with this child and the same feature. If the current node is an *operator* node, then inference engine runs the *operator* with the given feature. The *operator* node produces output features, and inference engine runs the child node of *operator* node with each of these features separately. If the current node is an *end* node, then this node and input feature is added to results list. When the search is finished, a list of end nodes is obtained corresponding to each generated feature. Then system goes over these nodes and let them display their results. Details of this algorithm is given in Algorithm 7.
The application provides two different search strategies selected from the tree traversal methods described in Section 2.1.3. The first method is the preorder breadth-first traversal. Although there are other breadth-first traversal strategies such as inorder and postorder, our application has to use the preorder strategy since we have to visit the *decision* node and then continue with the subtree pointed by this node, or visit the *operator* node and continue with the feature(s) provided by this node. Therefore, we should always visit the root first and then continue with the children. Application of this strategy on our decision tree is given in Algorithm 7. The second method is the depth-first traversal. Here all the nodes of the tree is traversed level by level starting from the root node. A node-feature pair constructed with root node is put into a queue, the top of the queue is visited and each node-feature pair constructed by *decision* or *operator* nodes are added to the end of the queue. This search continues until the queue is empty and *end* node-feature pairs are gathered like the preorder search. Application of this strategy on our decision tree is given in Algorithm 8.

3.3 User Interface and Program Flow

When the application starts, the main view is displayed with an empty tree, Figure 3.2. The operations available in this screen are:

- **Breadth First :** Indicates that the breadth first search given in Algorithm 7 will be used when searching for the decisions.
- **Depth First :** Indicates that the depth first search given in Algorithm 8 will be used

Algorithm 7 Preorder breadth-first search algorithm for the inference engine to search decision tree. Note that a *default* node can never be a part of complete tree so it is not considered in the algorithm

```
1: procedure breadth-first-search(node, feature)
2: resultlist = empty list
3: if node.type == decision then
4:    nexnode = node.decide(feature)
5:    resultlist.append( breadth-first-search(nextnode, feature) )
6: else if node.type == operator then
7:    nextnode =node.NextNode
8:    featurelist = node.run(feature)
9:    for all feature f in featurelist do
10:       resultlist.append( breadth-first-search(nextnode, f) )
11:   end for
12: else if node.type == end then
13:    resultlist.append( node, feature )
14: end if
15: return resultlist
```

Algorithm 8 Depth-first search algorithm for the inference engine to search decision tree. Note that a *default* node can never be a part of complete tree so it is not considered in the algorithm

```
procedure depth-first-search(node, feature)
resultlist = empty list
queue = empty node-feature pair queue
queue.add(node)
while queue is not empty do
    pair =queue.remove
    node = pair.node
    feature = pair.feature
    if node.type == decision then
        nextnode = node.accept(feature)
        queue.add(nextnode, feature)
    else if node.type == operator then
        nextnode =node.NextNode
        featurelist = node.run(feature)
        for all feature f in featurelist do
            queue.add( nextnode, f )
        end for
    else if node.type == end then
        resultlist.append( node, feature )
    end if
end while
return resultlist
```

when searching for the decisions.

- **Load :** Opens a file selection dialog. If the user selects a valid project document, the tree is filled with the tree in the selected document.
- **Save :** Opens a file save dialog. This option saves the current tree with the selected file name.
- **Run :** Runs the inference engine on the classification tree. Incomplete or invalid trees are rejected.
- **Cancel :** Quits the application.

User creates his own tree using context menu over nodes. This menu allows user to access the node specific view defined by the plug-in developer. For the initial case the only node in the tree is a *default* node. Using the view supplied by *default* node, user specifies the name of the class that extends one of the interfaces and current *default* node will be replaced with this class. A utility is supplied to user in order to help selection of node class. Given a package name, this utility supplies the list of available classes that can be plugged into our system. Also in through the user interface of *default* node, user selects the type of node, this selection is important because the tree will be formed according to this selection. If the selected type is *decision* node then user is supposed to input the number of children that this *decision* node will have. If the type is *operator* node then a single *default* subnode is added and at last if the type is *end* node then a leaf is reached and we are done with this branch. This operation is repeated until the expert knowledge is completely transferred to our knowledge-base. Once the tree is completed then classification can be run and the knowledge-base can be saved to disk for further use.

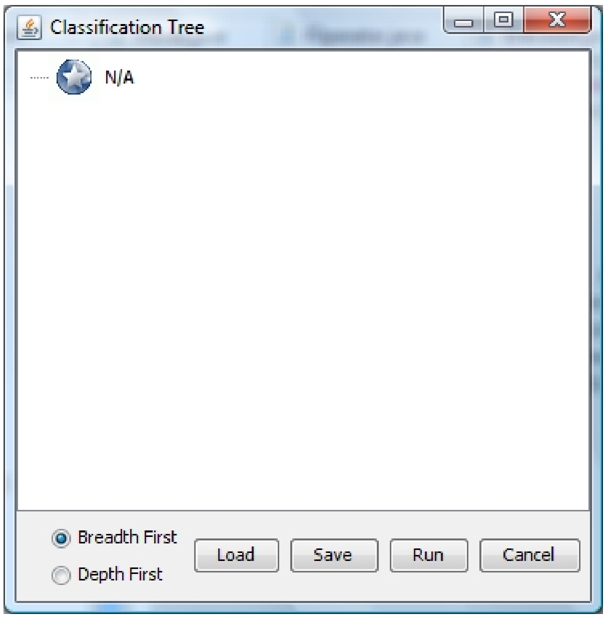


Figure 3.2: Opening Screen.

CHAPTER 4

A CASE STUDY: AIRPORT RUNWAY DETECTION IN HIGH RESOLUTION SATELLITE IMAGES

In Chapter 3, the software and its use is deeply explained. This proposed framework is experimented with a case study. The aim of this case study is to prove the usefulness of proposed system and to detect and classify the runway/runways of an airport. For this purpose special nodes which extend the interfaces explained in Section 3.2 are designed. Using these nodes, we start with extracting the straight lines from the input image, then continue with finding parallel line pairs among these straight lines. Once parallel lines are detected, each parallel line is checked against length difference, SIFT feature ratio, width and length for classification as shown in Algorithm 9. Classification table based on width and length of runways is given in Table Table 4.1 and results of steps applied by these nodes are given in Section 4.3.

Algorithm 9 Algorithmic description of steps used in extracting and classifying airport runways.

```
1: extract straight lines
2: for all line extracted do
3:     find a parallel line pair
4: end for
5: for all parallel line pair do
6:     if passes length difference elimination then
7:         if passes SIFT ratio elimination then
8:             classify according to width and length
9:         end if
10:    end if
11: end for
```

Table 4.1: Runway Classification Measures.*Adopted from manual [12]*

Support Area Runway Type	Runway Length (ft)	Runway Width (ft)	Runway Shoulder (ft)
Liaison	1000	50	N/A
Surveillance	3000	60	10
Light Lift	1500	60	10
Medium Lift	3500	60	10
Heavy Lift	6000	100	10
Tactical	5000	60	4

4.1 Specified Nodes for Airport Detection

4.1.1 FailureNode

A FailureNode is an *end* node, and it indicates that the search is unsuccessful.

4.1.2 SuccessNode

A SuccessNode is an *end* node, and it indicates that the search is successful. In addition, when the classification task is finished, a SuccessNode draws the runway found and outputs the name of classified result which is input by user in the preparation phase.

4.1.3 InputNode

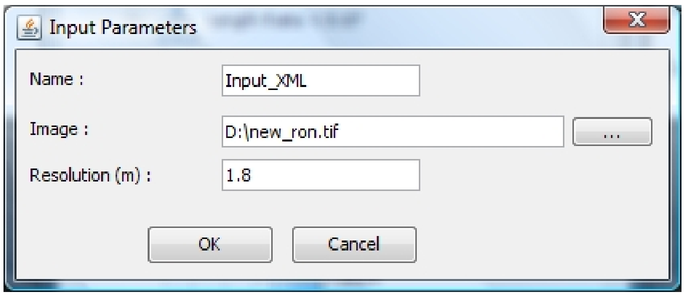
An InputNode is an *operator* node. This node is the root of the tree. Accepts the input image and resolution from the user using the interface given in Figure 4.1, and uses the techniques described in Section 2.2 to produce linear features and SIFT features to be used in further classification.

In order to produce linear features, image is split into sub-regions. Then, in each sub-region we apply edge detection and line detection to produce linear segments. After the linear segments are extracted they are connected based on their collinearity and proximity. At the end, linear segments found in each sub-region is connected across sub-regions based on collinearity and proximity again. The steps of this method is given in Algorithm 10. Result of applying Algorithm 10 is given in Figure 4.8 and Figure 4.14.

Once the straight lines are extracted, another important feature about detecting runways is the SIFT output. InputNode also runs the SIFT algorithm given in Section 2.2.5, and passes results of this algorithm to future nodes.

Algorithm 10 Algorithmic description of steps used in extracting straight lines.

```
1: all-lines = empty list of lines
2: divide image into sub-regions
3: for all sub-region in image do
4:     edge-image = detect edges in sub-region
5:     subregion-lines = detect lines in edge-image
6:     subregion-lines = connect collinear lines in subregion-lines
7:     all-lines = all-lines + subregion-lines
8: end for
9: all-lines = connect collinear lines in all-lines
10: return all-lines
```



Name : Input_XML

Image : D:\new_ron.tif

Resolution (m) : 1.8

Figure 4.1: InputNode user interface.

Here are the steps used for constructing straight lines in image.

4.1.3.1 Divide Image into Sub Regions

The input image is a high resolution image so an image that contains an airport can be very big in size which makes processing this image very slow. Running time of some algorithms grow exponentially with the image size, so input image is divided into sub-regions. Currently a size of 500x500 pixels is used, and a sample region is given in Figure 4.2.

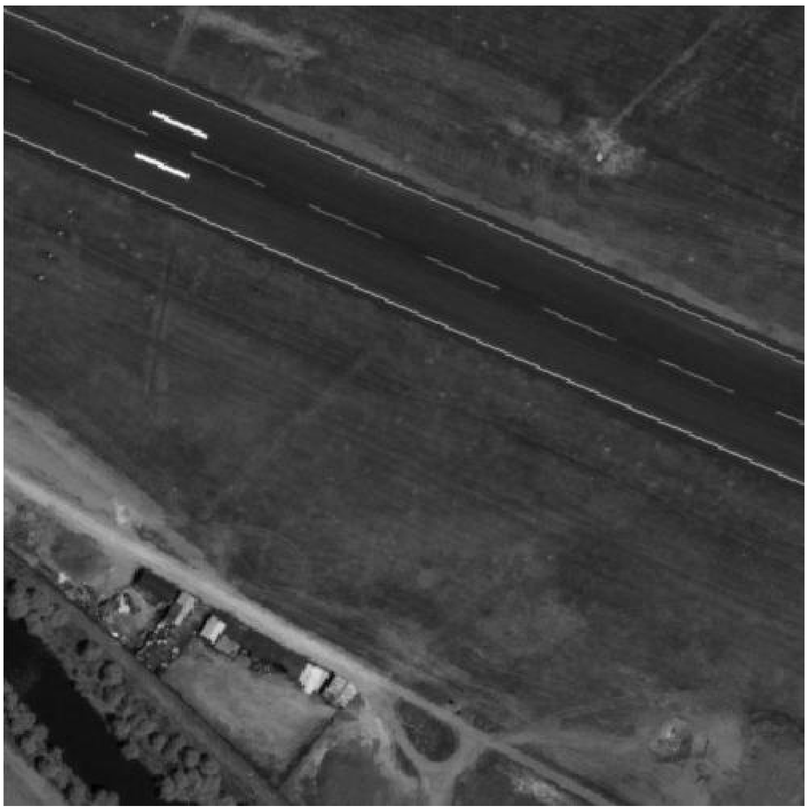


Figure 4.2: A sample subregion.

4.1.3.2 Detect Edges in Subregions

Shape of a runway is an important clue about the existence and classification of the runway, and edges provide important clues about the shape. Methods described in Section 2.2.1 are used to detect edges in sub regions. Result of applying edge detection on a sample region is given in Figure 4.3.

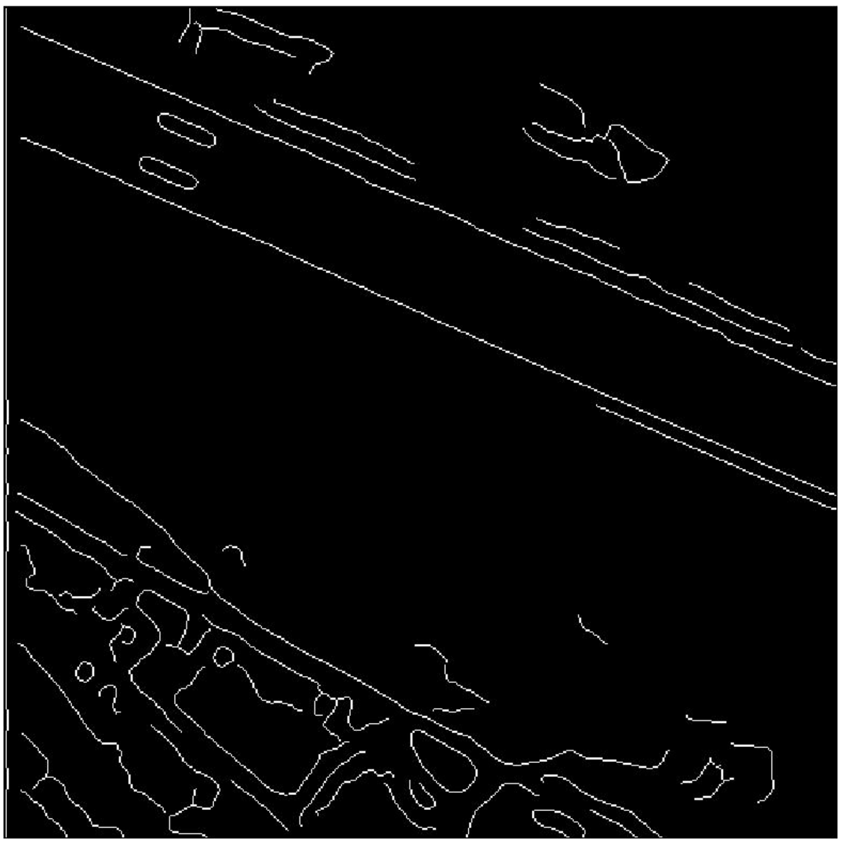


Figure 4.3: Result of applying edge detection to Figure 4.2.

4.1.3.3 Extract Straight Lines From Sub Regions

Each sub region may include a part of runway and this straight line should be extracted from the sub image. Once the edge image is obtained, then connected component labeling, Section 2.2.2, and boundary curve segmentation, Section 2.2.3, are used to extract straight lines. During this process, straight lines that are shorter than some threshold number are eliminated in order to reduce the amount of data to be processed. A sample result is given in Figure 4.4.

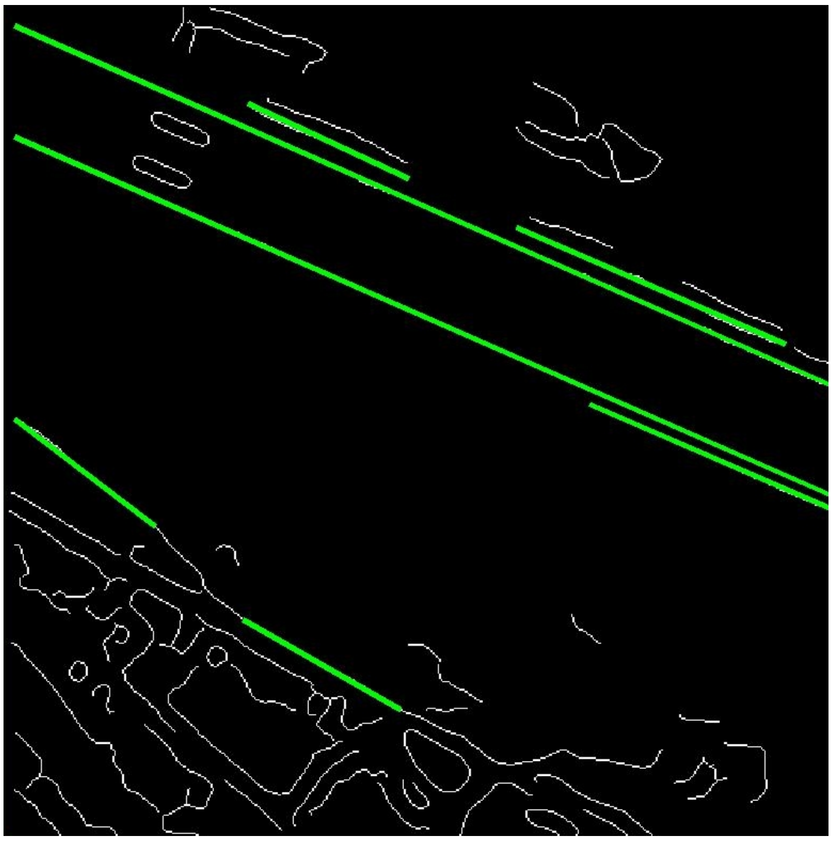


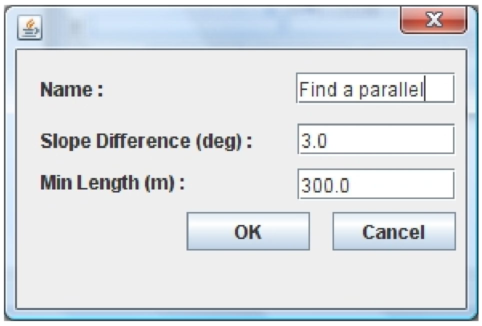
Figure 4.4: Result of line detection on Figure 4.3.

4.1.3.4 Connect Collinear Lines in a Sub-region

Edges of a runway may not necessarily produce a single long straight line, instead due to the properties of these edges a straight edge can be divided into some smaller collinear lines. Connecting these collinear lines will lead to a better line segment. Method explained in 2.2.4 is used. Maximum slope difference is taken to be 0.1 radians, since we are looking for shapes that are made up of nearly perfect lines. Besides, threshold values T_s and T_p used in Algorithm 6 are 200 and 4 pixels respectively.

4.1.3.5 Connect Collinear Lines Across Sub-regions

Previous step created straight lines that begin and end in the same sub-region but a runway can pass across multiple sub-regions so collinear line segments are connected across sub-regions. The method used here is the same as the previous step.



Name :	Find a parallel
Slope Difference (deg) :	3.0
Min Length (m) :	300.0

Figure 4.5: User interface of FindParallelsNode.

4.1.4 FindParallelsNode

A FindParallelsNode is an *operator* node. The shape of a runway implies that we should have parallel lines of sufficient length. This node accepts a line feature produced by InputNode, searches through the other lines and finds possible parallel line pairs for this line. Two lines are accepted as parallel if the difference of their slopes are less than a threshold provided by user. Currently 3 degrees is used as threshold. Also a parallel pair is accepted if the length of each line is greater than a value specified by user, 300 meters for a runway. The interface and parameters of node is given in Figure 4.5 and parallel lines features extracted by this node are shown in Figure 4.9 and Figure 4.15.

4.1.5 ParLineLengthDiffNode

A ParLineLengthDiffNode is a *decision* node. In order to label a parallel line pair as a runway, the lines making up the pair should not have great length difference since we expect them to start and end in near positions to each other. This node accepts a parallel lines feature produced by FindParallelsNode and makes its decision based on the ratio of the length difference between lines to length of one of the lines. A ratio of 0.2 is used for a runway, because we are looking for shapes that are nearly rectangle. Parallel lines that are not eliminated by this node are shown in Figure 4.10 and Figure 4.16.

4.1.6 ParLineAroundCleanNode

A ParLineAroundCleanNode is a *decision* node. There can not be buildings, trees etc. at a specified lateral area of a runway; in other words lateral area of a runway should be clean for a safe operation. SIFT algorithm described in 2.2.5 is used for this decision. ParLineAroundCleanNode accepts a parallel line feature, counts the number of SIFT features around these lines in an area defined by user and makes its decision based on the *SIFT feature* *count / length of line* ratio. A ratio of 0.1 is used and sideway width is taken as 10 meters. Parallel lines that are not eliminated by this node are shown in Figure 4.11 and Figure 4.17.

4.1.7 ParallelLineWidthNode

A ParallelLineWidthNode is a *decision* node. Width of a parallel line is an indicator of whether this parallel line can be a runway or not. Also width is an important clue for classifying a runway (Tactical, medium lift, light lift etc.), Table 4.1. This node accepts a parallel line feature and makes its decision based on the normal distance of one line to another.

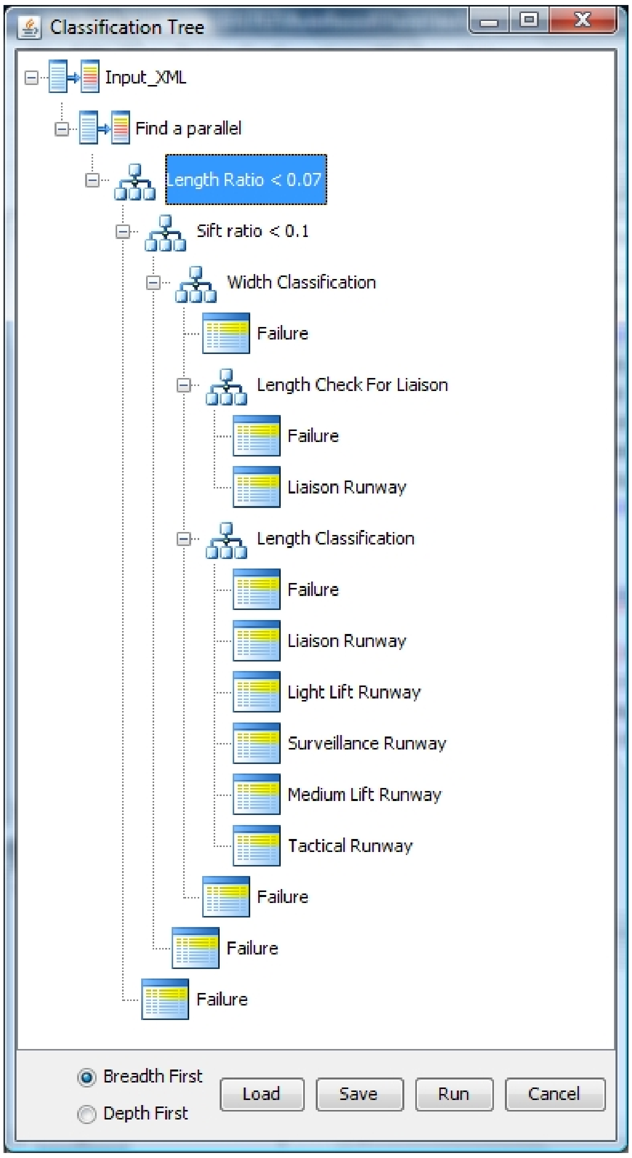
4.1.8 ParallelLineLengthNode

A ParallelLineLengthNode is a *decision* node. Just like the width, length of a parallel line is also an indicator of whether this parallel line can be a runway or not, and used for classifying the runway in the same manner, Table 4.1. This node accepts a parallel line feature and makes its decision based on the mean of length of lines.

4.2 Putting Nodes Together

Once these nodes are developed they must be placed in the correct places for accurate classification of runways. In our attempt to build a classification tree using these nodes, we first placed the InputNode and FindParallelsNode, *operator* nodes, one by one in order to produce the features to be used by the *decision* nodes. *Decision* nodes start with ParLineLengthDiffNode, ParLineAroundCleanNode. These two nodes eliminates most of the deceptive runways and only possible candidates are left. This elimination step is followed by ParallelLineWidthNode's and ParallelLineLengthNode's placed according to Table 4.1 for classification of runway candidates. And at last proper SuccessNode's and FailureNode's are placed to indicate the classification results. The classification tree constructed is shown in Figure 4.6.

Results of running this tree on Figure 4.7 and Figure 4.13 are shown in Figure 4.12 and Figure 4.18 respectively.


Classification Tree

- Input_XML
  - Find a parallel
    - Length Ratio < 0.07
      - Sift ratio < 0.1
        - Width Classification
          - Failure
          - Length Check For Liaison
            - Failure
            - Liaison Runway
          - Length Classification
            - Failure
            - Liaison Runway
            - Light Lift Runway
            - Surveillance Runway
            - Medium Lift Runway
            - Tactical Runway
          - Failure
        - Failure
      - Failure

[✓ Breadth First] Depth First

Figure 4.6: Runway classification tree based on Table 4.1.

4.3 Results of Classification

Two sample satellite images, Figure 4.7 and Figure 4.13, containing airport runways will be used in this chapter in order to describe these nodes and their internals.

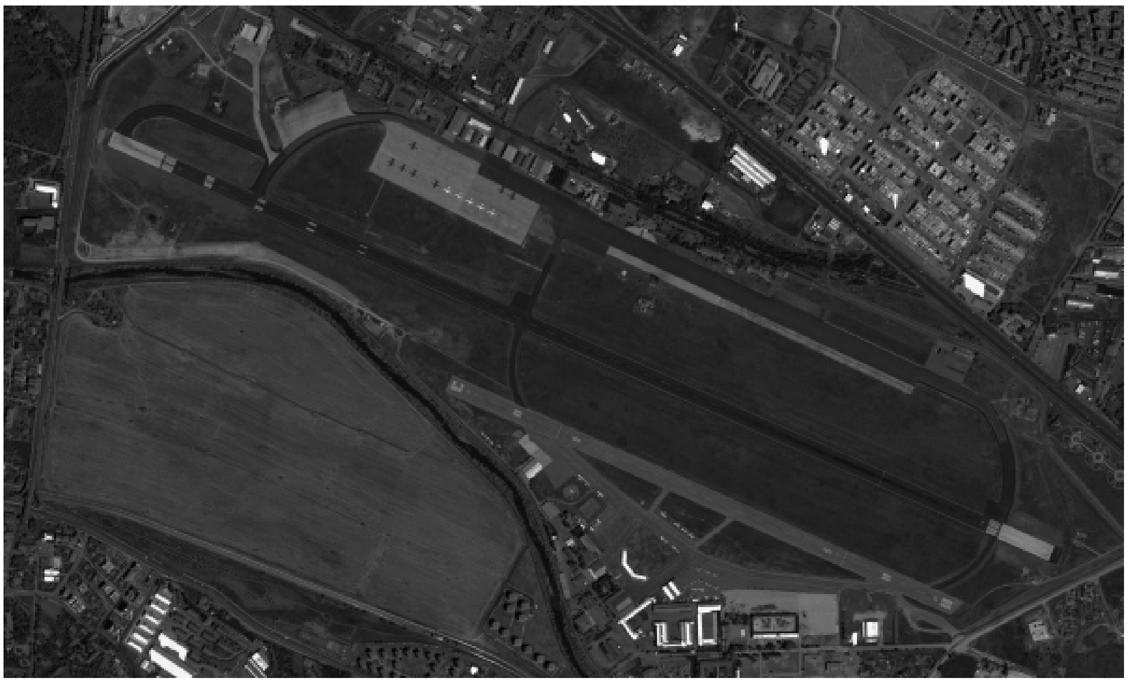


Figure 4.7: A Quickbird image of Etimesgut Military Airport. Image Size is 5000x3000 pixels and resolution is 0.6 meters per pixel.

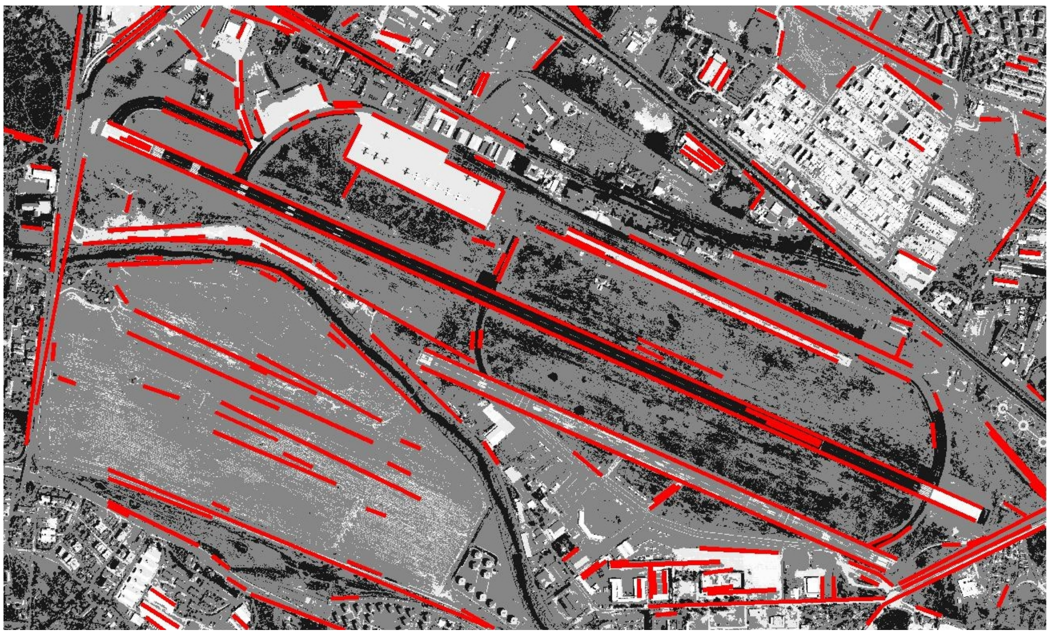


Figure 4.8: Line features created by InputNode (4.1.3) on Figure 4.7.

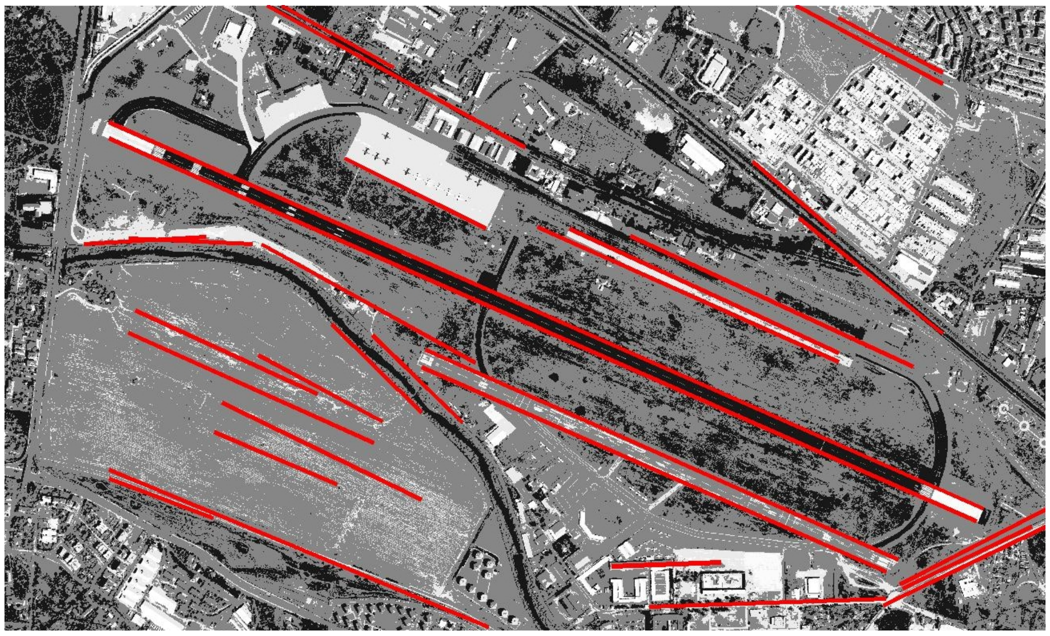


Figure 4.9: Parallel line features created by FindParallelsNode (4.1.4) on Figure 4.8.

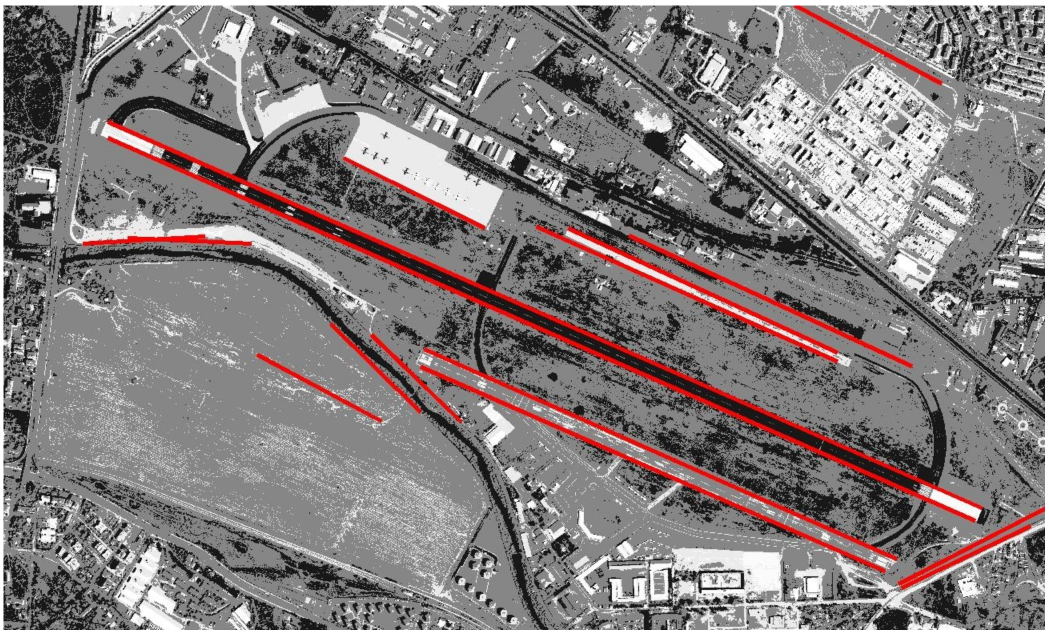


Figure 4.10: Parallel lines of Figure 4.9 eliminated with length difference.

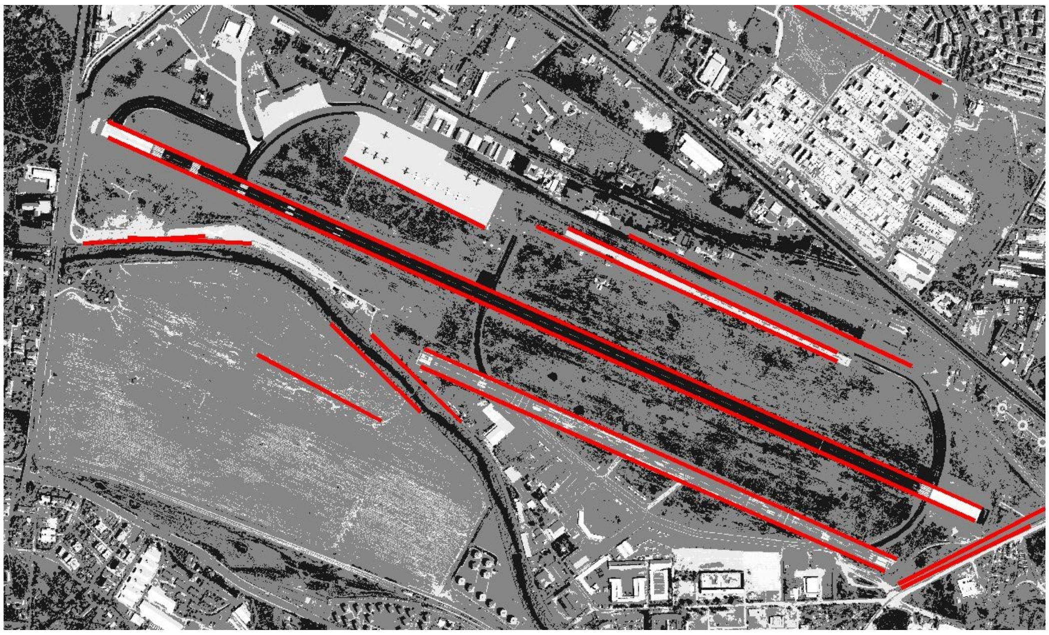


Figure 4.11: Parallel lines of Figure 4.10 eliminated with SIFT feature ratio around the lines.

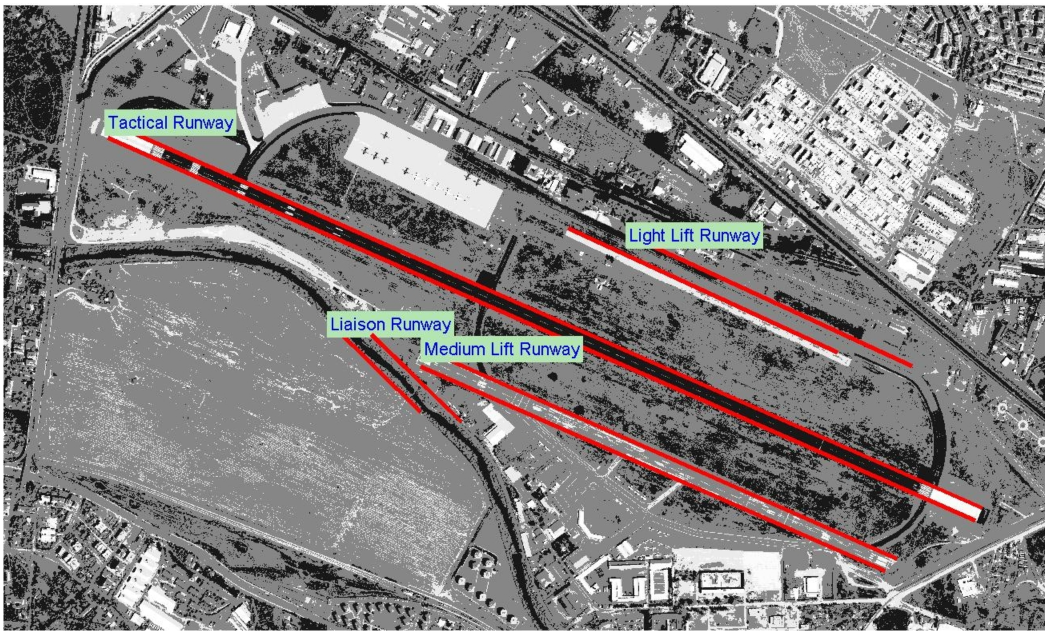


Figure 4.12: Result of running classification tree of Figure 4.6 on Figure 4.7. 3 runways are detected, 1 light lift, 1 medium lift and 1 tactical runway. Also 1 liaison runway is detected incorrectly.

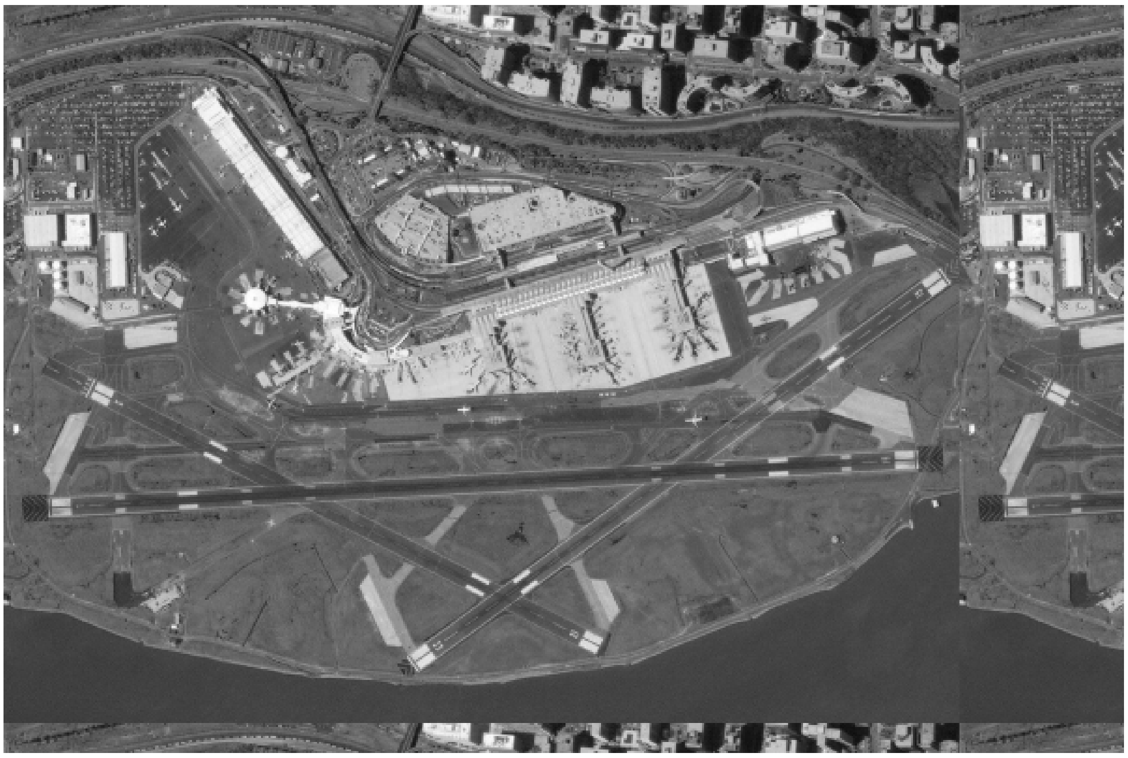


Figure 4.13: An image of Ronald Reagan Washington National Airport. Image Size is 1500x1000 pixels and resolution is 1.8 meters per pixel. Left and top of the image are padded to original image to obtain an image of that size.

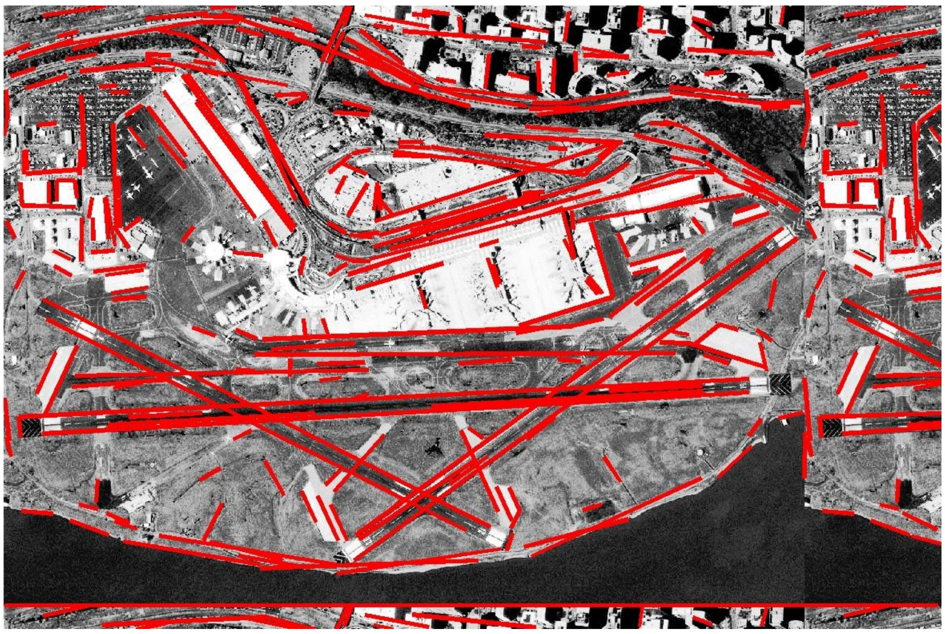


Figure 4.14: Line features created by InputNode (4.1.3) on Figure 4.13.

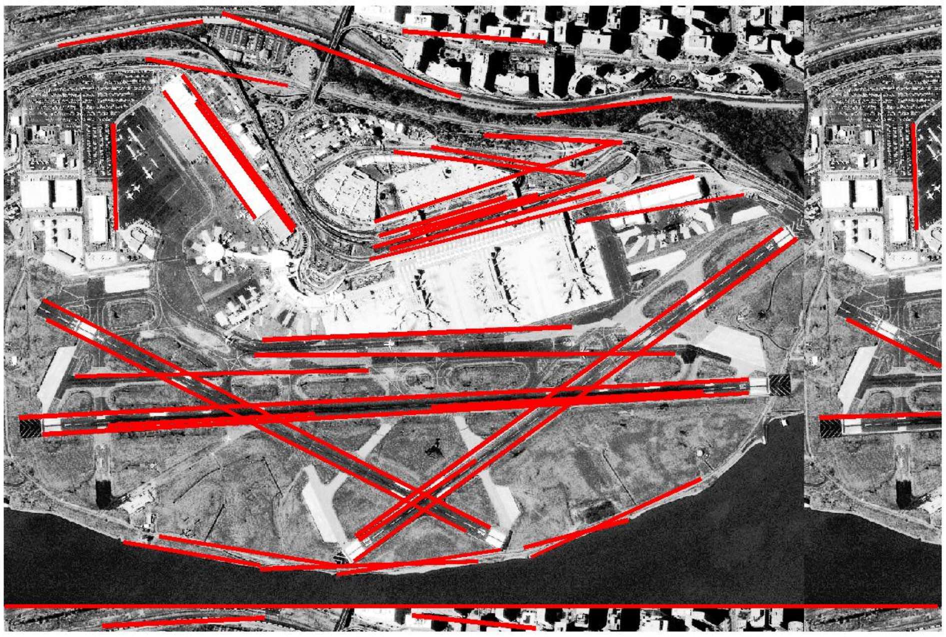


Figure 4.15: Parallel line features created by FindParallelsNode (4.1.4) on Figure 4.14.

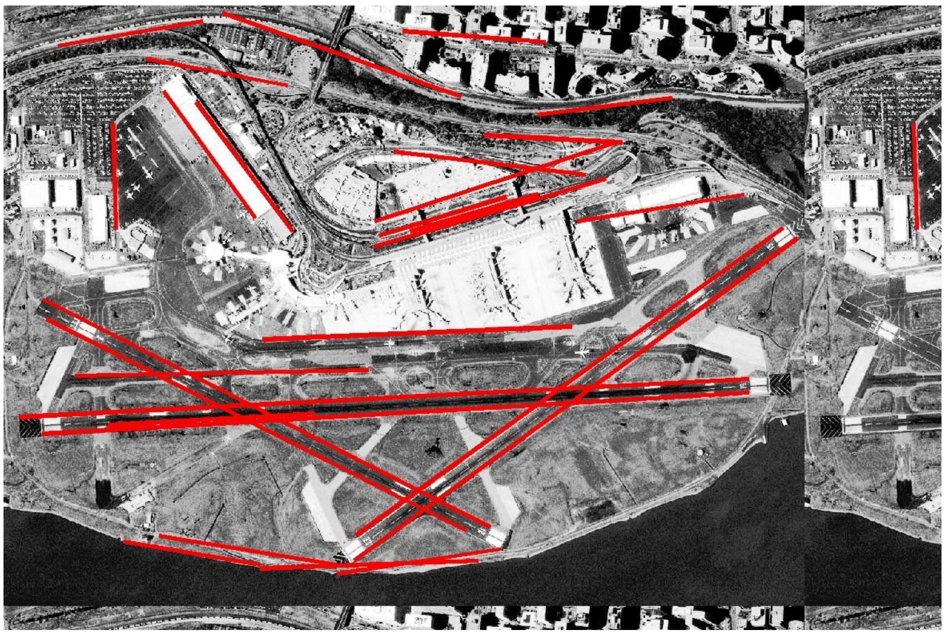


Figure 4.16: Parallel lines of Figure 4.15 eliminated with length difference.

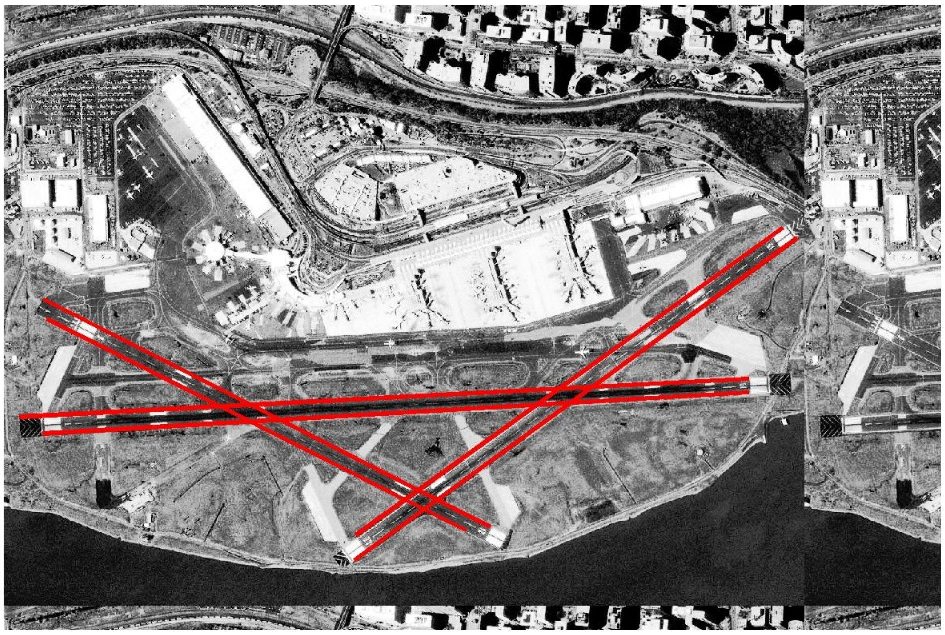


Figure 4.17: Parallel lines of Figure 4.16 eliminated with SIFT ratio around the lines.

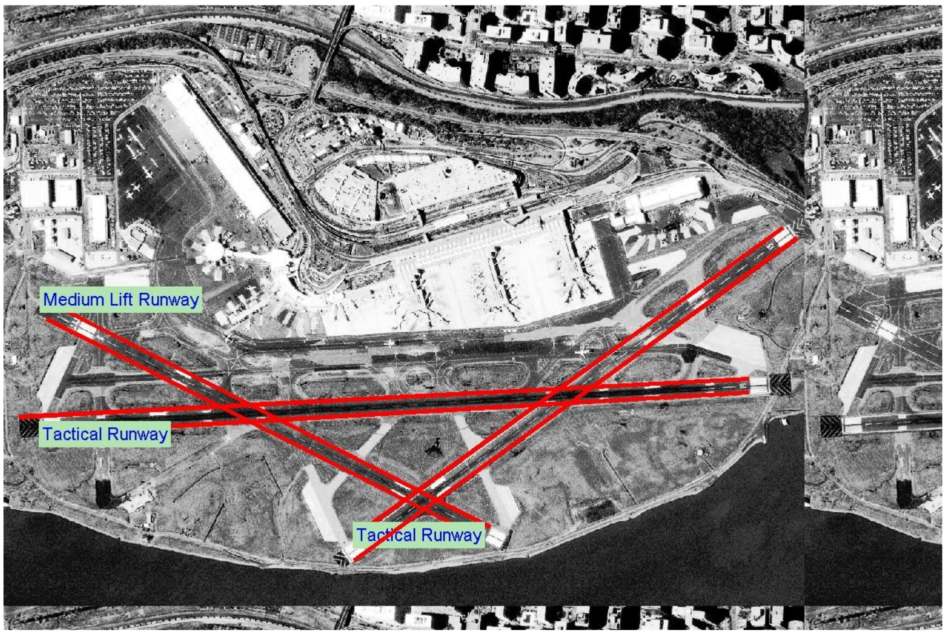


Figure 4.18: Result of running classification tree of Figure 4.6 on Figure 4.13. 1 medium and 2 tactical runways are detected.

CHAPTER 5

A CASE STUDY: URBAN AREA DETECTION IN HIGH RESOLUTION SATELLITE IMAGES

In Section 4 the proposed framework is tested on a case study on detection and classification of airport runways. The aim of this case study is to experiment the proposed framework with a different problem and show that the framework can be applied to different target detection or classification task. In this case study the framework is used to build a software for classification of regions as urban areas or not. For this purpose necessary *decision*, *operator*, *end* nodes which extend the interfaces explained in Section 3.2 are designed and put together in order to form a classification tree. Using these nodes, the image is split into sub-regions. Then, count of lines and SIFT features in each sub-region is calculated and finally, each sub-region is classified as urban or non-urban using this information as shown in Algorithm 11. Results of techniques used in these nodes are given in Section 5.3.

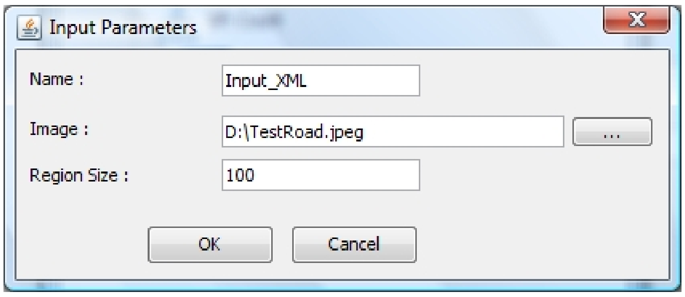
Algorithm 11 Algorithmic description of steps used in urban area detection.

```
1: divide image into sub-regions
2: mark each sub-region as non-urban
3: for all sub-region in image do
4:    find line count in sub-region
5:    if line count is above threshold then
6:       find SIFT feature count in sub-region
7:       if SIFT feature count is above threshold then
8:          mark sub-region as urban
9:       end if
10:   end if
11: end for
```

5.1 Specified Nodes for Urban Area Detection

5.1.1 UrbanInputNode

UrbanInputNode is an *operator* node, and is the root of our classification tree. The node accepts image and the size of regions that the input image is going to be split. In this node image is split to subregions and features defining each subregion is constructed for further use of the rest of the tree. Figure 5.1 shows the user interface of this node.



Name : Input_XML

Image : D:\TestRoad.jpeg

Region Size : 100

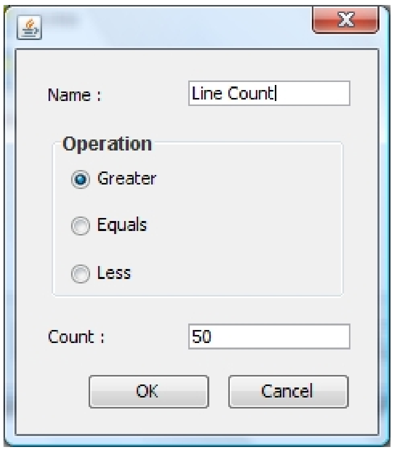
Figure 5.1: UrbanInputNode user interface.

5.1.2 LineCountNode

LineCountNode is a *decision* node. The idea behind the node is that an urban area must dominantly have man made structures and man made structures are generally composed of straight lines, so count of the straight lines can be a good indicator of an urban area. The node's input is a feature that defines the subregion. It first applies edge detection to the subregion as explained in Section 2.2.1, then finds the connected components with the method given in Section 2.2.2. Uses these connected components in edge image to form straight lines with boundary curve segmentation explained in 2.2.3. It then decides if the input region is rejected or accepted based on the count of the lines in the input subregion. The user interface and inputs of the node is given in Figure 5.2.

5.1.3 SiftCountNode

SiftCountNode is a *decision* node used for deciding if the input subregion is part of an urban area using the count of SIFT features described in Section 2.2.5. SIFT detects the important features in an image, since urban areas are made up of man made structures and man made structures usually have corners or distinctive shapes they are bout to produce more features than the rural areas. Given a subregion, this node calculates the count of of SIFT features in that subregion and compares with the input SIFT count in order the accept or decline this subregion. The user interface of this node is the same as LineCountNode of Section 5.1.2 and is given in Figure 5.2.


Name : Line Count

Operation: [✓ Greater] Equals Less

Count : 50

Figure 5.2: LineCountNode and SiftCountNode user interface.

5.2 Putting Nodes Together

Once the nodes described in Section 5.1 are developed then they can be placed in our classification tree to achieve the classification of regions as urban area or not. The tree starts with the UrbanInputNode which splits the input image into the subregions. Then LineCountNode eliminates some of these subregions based on the count of straight lines in this subregion, and at last SiftCountNode makes the final decision about the subregions. The tree constructed using those nodes are given in Figure 5.3.

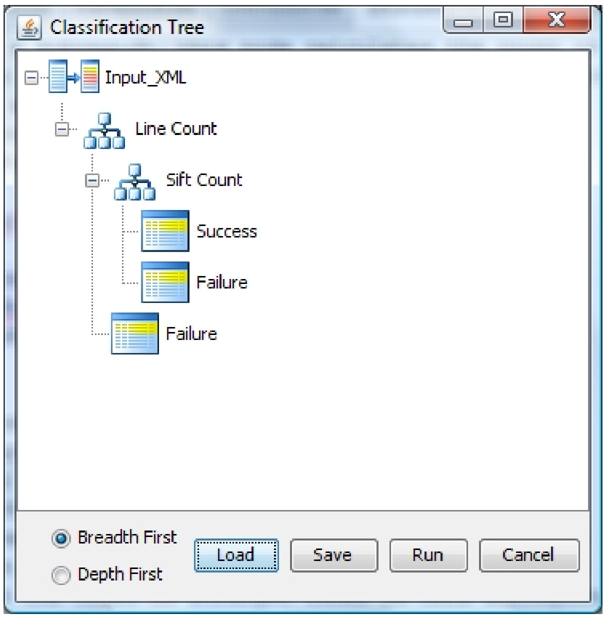


Figure 5.3: Urban area classification tree.

5.3 Results of Classification

After the classification tree is constructed a 3200x2400 panchromatic SPOT image with spatial resolution 10m, given in Figure 5.4, is used to test the tree. Line counts and SIFT feature counts of a sample set of subregions are given in Figure 5.5 and Figure 5.6 respectively. The regions that pass the test of line count are highlighted in Figure 5.7 and the regions that pass the SIFT count elimination are highlighted in Figure 5.8. In this elimination line count limit is set to 50 and SIFT feature count limit is set to 75. As can be seen from this resultant image most of the urban areas are detected correctly and rural areas are eliminated. Results of this classification is given in Table 5.1.

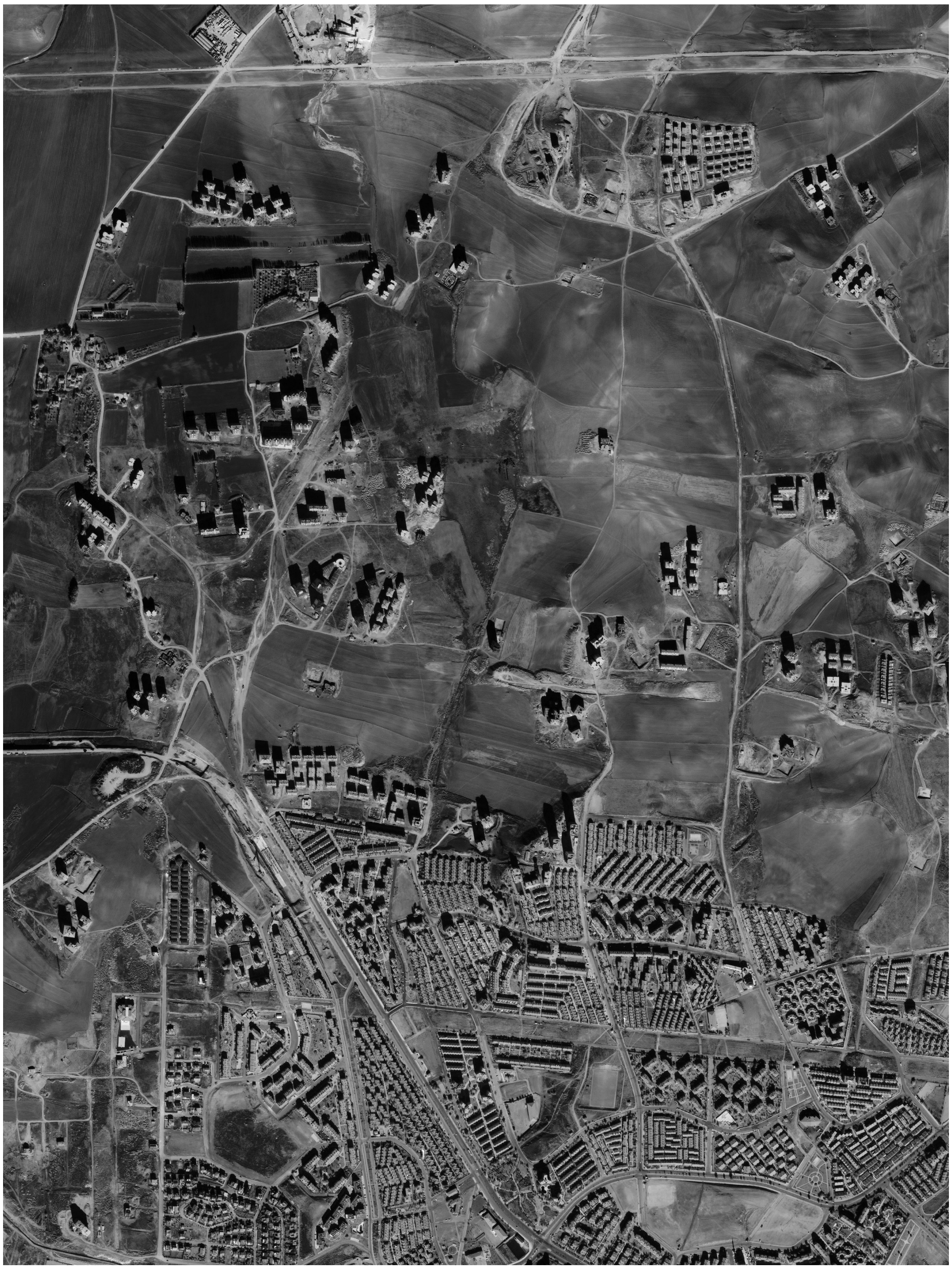


Figure 5.4: A panchromatic SPOT image. Image size is 3200x2400 and spatial resolution is 10 meters per pixel.

Urban region count in ground truth data	292
Urban region count found by classification tree	350
False positive region (not found by our tree) count	22
False negative region (found incorrectly by our tree) count	80
False Positive / Ground Truth	0.075
False Negative / Ground Truth	0.228

Table 5.1: Results of urban area classification. Line count limit is taken as 50, SIFT feature count limit is taken as 75.

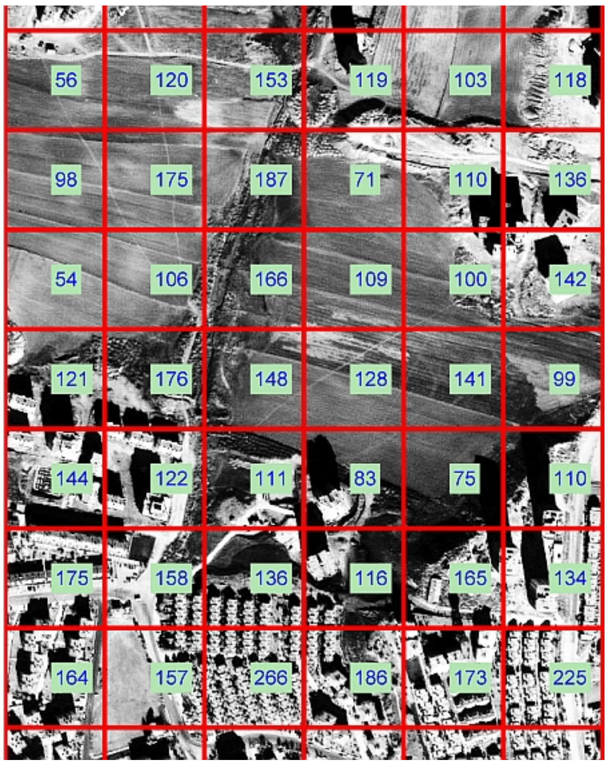


Figure 5.5: Count of lines for a sample set of subregions from Figure 5.4.

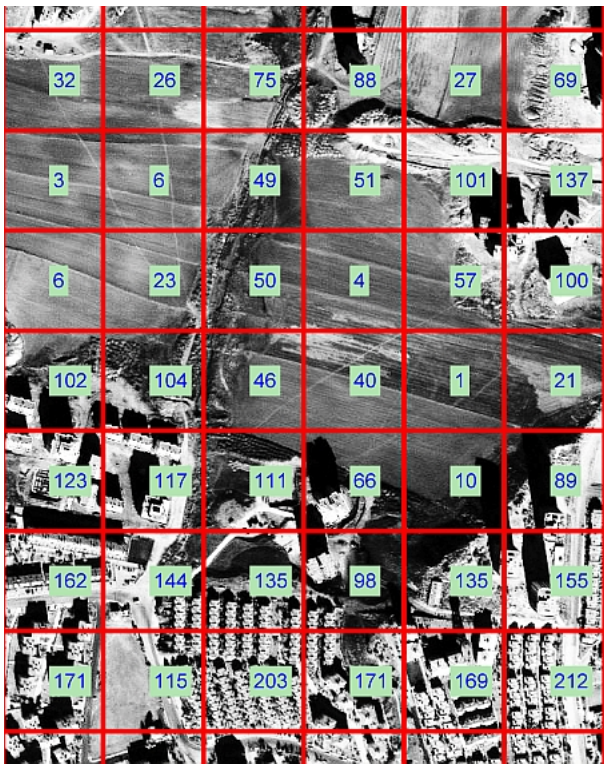


Figure 5.6: Count of SIFT features for a sample set of subregions from Figure 5.4.

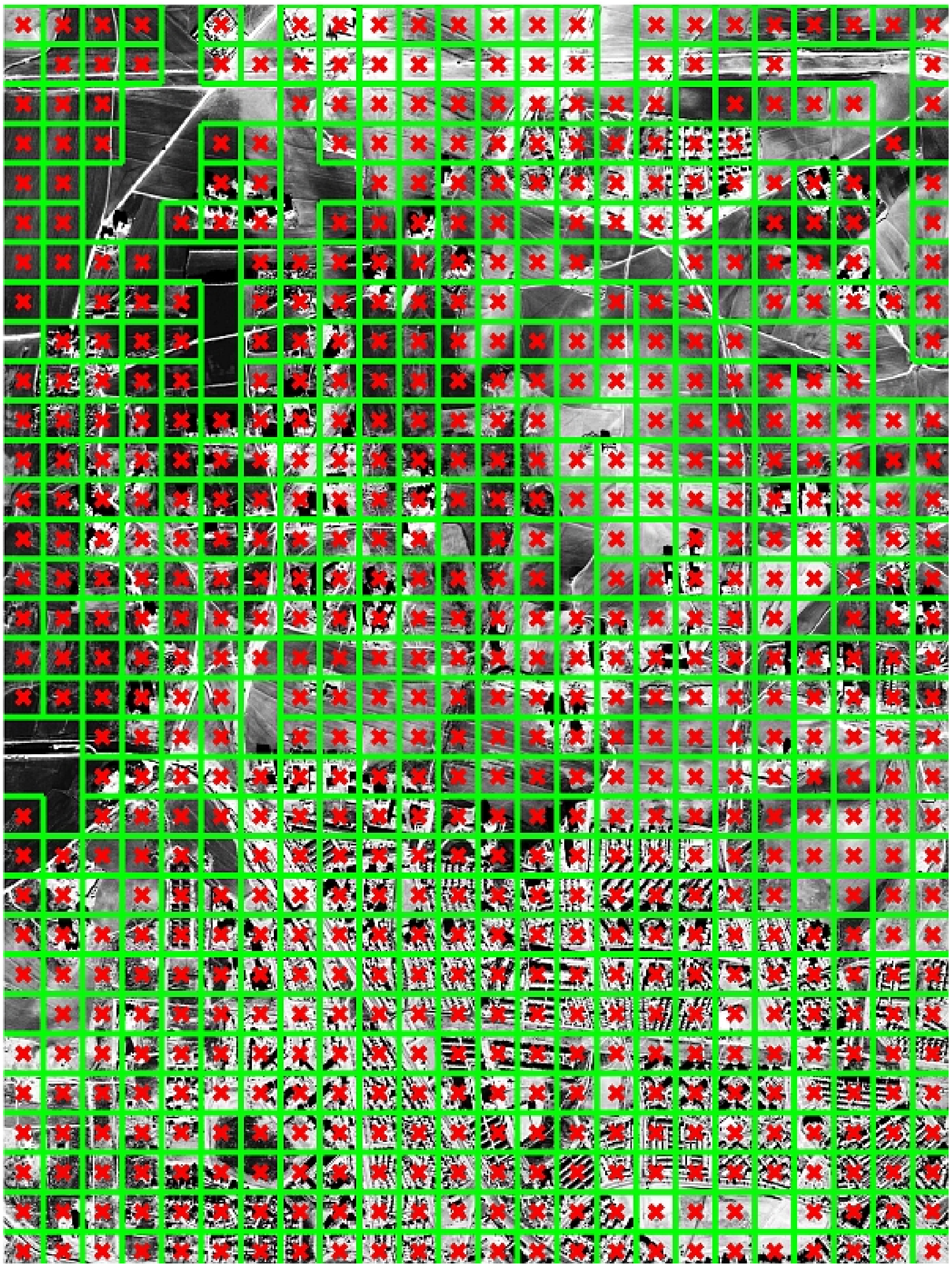


Figure 5.7: The subregions that pass the line count elimination on Figure 5.4 are highlighted.

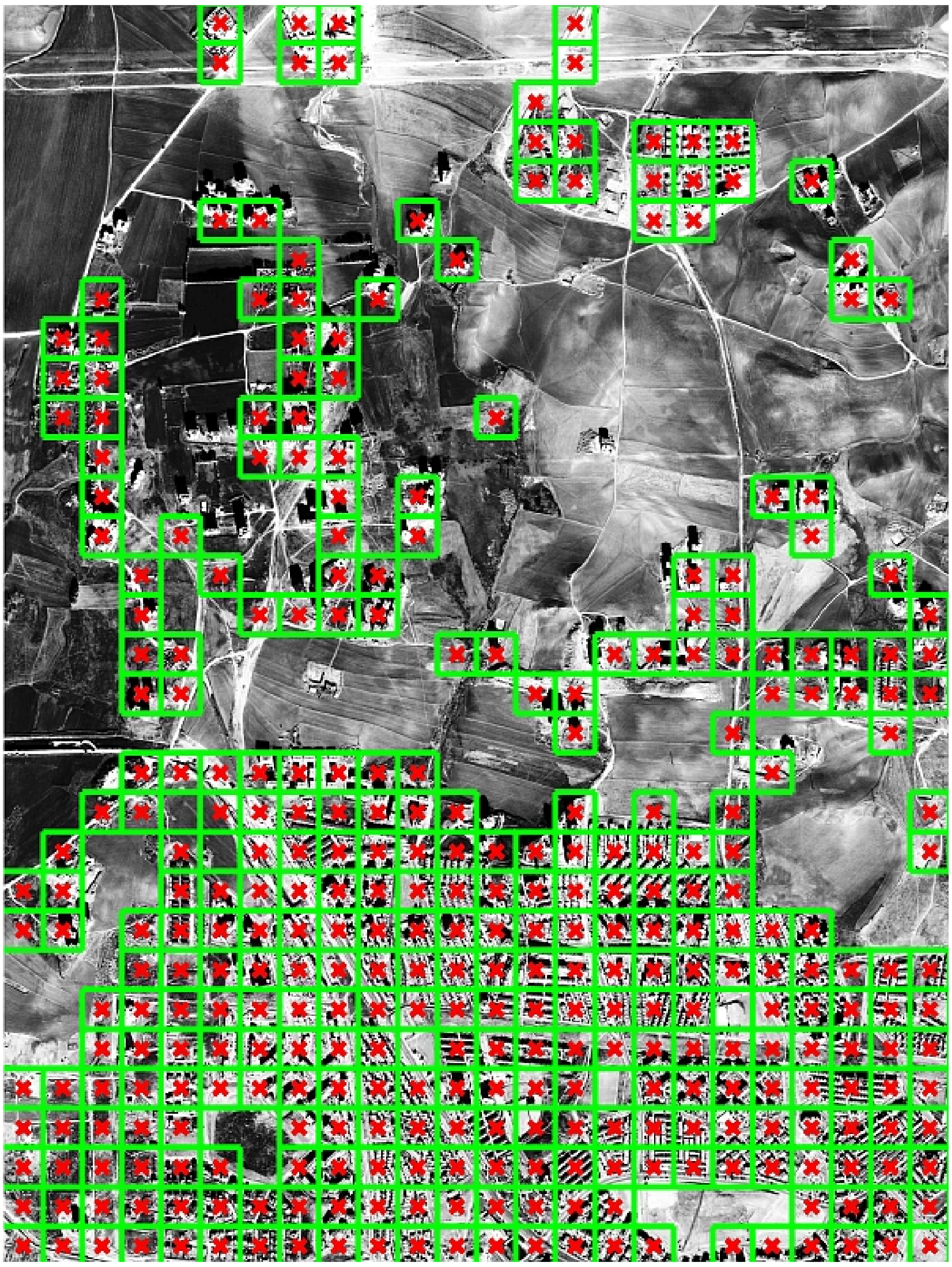


Figure 5.8: The subregions that pass the sift count elimination on Figure 5.7 are highlighted.

CHAPTER 6

CONCLUSION

Improvements in the technology resulted in the higher resolution satellite images, and also the availability of these images increased. Therefore, tasks such as classification or target detection need more and more automated solutions. Based on this need, a programming framework to implement rule-based target detection in images is proposed. We use a decision tree as the knowledge representation and we propose an inference engine to work on this decision tree. Using this framework, one can define his own problem specific nodes and plug these nodes into our framework on runtime. This capability is very useful when one has a well designed and mature library of nodes such that she can re-use already defined nodes in new problems; therefore it will no longer be necessary to solve already solved problems. Hence new expert systems can be developed more rapidly and effort can be canalized to problem specific nodes efficiently. Also the framework separates constructing an expert system into two parts:

- **Develop Nodes:** This is the work of the developer and needs technical background.
- **Construct Knowledge Base:** This is the work of the human expert who knows the details of detection or classification and who arranges nodes in order to achieve desired result. This task does not need technical background but domain knowledge is very useful.

Framework supplies 3 types of nodes:

- **Decision:** these nodes are used for making decisions and guides the search path;
- **Operator:** these nodes are used for creating new features from existing features;
- **End:** these nodes are the leaves of the decision tree and indicate a result has been reached.

The user of framework should extend these nodes in order to create pluggable nodes. The framework starts with an empty decision tree, user interactively adds his nodes to the tree, application guides the user through tree creation steps. Once the tree is constructed, user can run the inference engine to gather the results. Inference engine starts from the root, continues with the path defined by nodes applying each feature generated by any operator node down to this node.

The framework proposed is experimented with two case studies. The first case study is on *"Airport Runway Detection in High Resolution Satellite Images"*. Specific nodes for this problem are developed and plugged into the framework. Once the nodes are developed, putting these nodes together to construct an expert system for runway detection no longer needs a developer, but only an expert who knows the detection rules. The final expert system not only detects runways but also classifies the runways according to their lengths and widths. For this purpose some of the nodes were re-used extensively which shows even for a single problem re-use is possible and is promising about re-usability. The second case study is on *Urban Area Detection in High Resolution Satellite Images*. This approach used in this case study relies on the fact that urban areas are made up of man made structures and these structures are mostly seen as lines and distinct corners in a satellite image. Image is split into subregions and these subregions are eliminated based on the count of lines and SIFT features. This second study has shown us that the proposed framework can be used for different kind of problems.

As future work, some improvements on the current framework and the specific nodes about case study can be made. First of all, the current software is single threaded which makes it inefficient in multiprocessor machines. Multi-threaded run strategies can be added and features can be run in different threads. In addition, since depth-first search goes level-by-level, user can be informed about the run progress of the tree by highlighting the current tree level. Another informative extension can be giving a simple report on about the run. In a single run many features are constructed and after a run is completed it can be useful for the user to learn the count of features created by nodes or count of features that visited a specific node. Finally, new nodes about the case study can be generated to consolidate the result, for example such nodes can check if there are airplane like figures around the runway or if the region inside the runway has a consistent texture. Furthermore urban area detection can be consolidated by using nodes that check the gray level properties or textural properties of the subregions.

REFERENCES

[1] Ajith Abraham. *Handbook of Measuring System Design*, chapter Rule Based Expert Systems. John Wiley & Sons, Ltd, 2005.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 7 edition, 2001.

[3] Oral Dalay. Interactive classification of satellite image content based on query by example. Master's thesis, Middle East Technical University, January 2006.

[4] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Prentice Hall, 2002.

[5] Pranav Gupta and Anupam Agrawal. Airport detection in high-resolution panchromatic satellite images. *Journal of Institution of Engineers (India)*, 88, May 2007.

[6] Jun Wei Han, Lei Guo, and YongSheng Bao. A method of automatic finding airport runways in aerial images. *2002 6th International Conference on Signal Processing*, 1:731–734, 2002.

[7] Cay S. Horstmann and Gary Cornell. *Core Java^{TM} 2*, volume 1. Prentice Hall, 7 edition, 2004.

[8] Peter Jackson. *Introduction to Expert Systems.* Addison-Wesley, 1990.

[9] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60, 2:91–110, 2004.

[10] S. Sumathi and S.N. Sivanandam. *Introduction to Data Mining And its Applications.* Springer, 2006.

[11] Gökhan Tekkaya. Improving interactive classification of satellite image content. Master's thesis, Middle East Technical University, 2007.

[12] US ARMY Intelligence Center, Fort Huachuca, AZ 85613-6000. *Selecting Entry Zones In Aerial Imagery.* Lecture Notes.

[13] Wikipedia, en.wikipedia.org/wiki/Classification_trees. *Decision Tree.*