


## AN ABSTRACT OF THE THESIS OF

Martin D. Dill for the degree of Master of Science in Electrical and Computer Engineering presented on June 9, 1997. Title: Improving Motion Estimation with Evolvable Search Algorithms

Redacted for Privacy

Abstract approved: \_\_\_\_\_

 James H. Herzog

Until now the topic of motion estimation, as used in video compression, has been dominated by search methodologies which are modifications of an exhaustive search. This research takes a completely new approach by applying two evolvable search algorithms, the Genetic Algorithm and the Genetic Program, to the area of motion estimation. The main purpose of this research is to determine the applicability of evolvable search methods to the topic of motion estimation. Several methods are studied: in the first application, a Genetic Algorithm is used to determine individual motion vectors one at a time, while the second method explores the use of a Genetic Algorithm to search for all of the motion vectors to correlate two frames simultaneously. To reduce the number of motion vectors required, Genetic Programming is applied to variable block size motion estimation. Finally, this work is expanded by applying it to region motion estimation, which is not restricted to using square or rectangular motion blocks.

©Copyright by Martin D. Dill  
June 9, 1997  
All Rights Reserved

Improving Motion Estimation With Evolvable Search Algorithms

by

Martin D. Dill

A THESIS

submitted to

Oregon State University

in partial fulfillment of  
the requirements for the  
degree of

Master of Science

Completed June 9, 1997  
Commencement June 1998

Master of Science thesis of Martin D. Dill presented on June 9, 1997

APPROVED:

Redacted for Privacy

---

Major Professor, representing Electrical & Computer Engineering

Redacted for Privacy

---

Chair of Department of Electrical & Computer Engineering

Redacted for Privacy

---

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

---

Martin D. Dill, Author



## **ACKNOWLEDGMENT**

Dr. James Herzog of Oregon State University has patiently reviewed and supported this work.

## TABLE OF CONTENTS

	<b><u>Page</u></b>
1. INTRODUCTION.....	1
2. PHILOSOPHY AND REVIEW OF LITERATURE.....	6
2.1. Review of Motion Estimation Algorithm Literature.....	6
2.2 Introduction to Genetic Algorithms and Genetic Programming.....	11
2.3 Rationale for the Effectiveness of Genetic Operators.....	24
3. THESIS CONTRIBUTIONS AND VERIFICATIONS.....	27
3.1 Experimental Overview.....	27
3.2 Block Motion Estimation with Fixed Block Sizes.....	27
3.2.1 Fixed Size Block Motion Estimation, Iterative Method.....	28
3.2.2 Fixed Size Block Motion Estimation, One-Shot Method.....	48
3.3 Block Motion Estimation with Variable Block Sizes.....	55
3.4 Region Based Motion Estimation.....	65
3.4.1 Region Based Motion Estimation, Fixed Block Size Method.....	65
3.4.2 Region Based Motion Estimation, Variable Block Size Method.....	69
4. CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK.....	78
BIBLIOGRAPHY.....	82
APPENDICES.....	89

## LIST OF FIGURES

<b><u>Figure</u></b>	<b><u>Page</u></b>
1.1 Typical MPEG picture sequence .....	4
1.2 Motion prediction example .....	5
2.1 Example GA bit string denoting variable fields .....	11
2.2 GP bit string of solution program for the Positive Root of the Quadratic Equation.....	12
2.3 GP tree illustrating the program for the positive root solution of the quadratic equation .....	13
2.4 Detail of Crossover operation .....	17
2.5 Parent 1 in Crossover operation for example GP .....	19
2.6 Parent 2 in Crossover operation for example GP .....	20
2.7 Child 1 in Crossover operation for example GP .....	21
2.8 Child 2 in Crossover operation for example GP .....	22
2.9 Schemata example .....	25
3.1 Color difference of fixed block size iterative and exhaustive search .....	31
3.2 Number of compares for fixed block size iterative search, Test 1 .....	32
3.3 Number of compares for fixed block size iterative search, Test 2 .....	33
3.4 Number of compares for fixed block size iterative search, Test 3 .....	34
3.5 Number of compares for fixed block size iterative search, Test 4 .....	35
3.6 Number of compares for fixed block size iterative search, Test 5 .....	36
3.7 Number of compares for fixed block size iterative search, Test 6 .....	37
3.8 Number of compares for fixed block size iterative search, Test 7 .....	38

## LIST OF FIGURES (Continued)

<u>Figure</u>	<u>Page</u>
3.9 Number of compares for fixed block size iterative search, Test 8.....	39
3.10 Color difference of improved fixed block size iterative and exhaustive search.....	40
3.11 Number of compares for improved fixed block size iterative search, Test 1 .....	41
3.12 Number of compares for improved fixed block size iterative search, Test 2 .....	42
3.13 Number of compares for improved fixed block size iterative search, Test 3 .....	43
3.14 Number of compares for improved fixed block size iterative search, Test 4 .....	44
3.15 Number of compares for improved fixed block size iterative search, Test 5 .....	45
3.16 Number of compares for improved fixed block size iterative search, Test 6 .....	46
3.17 Number of compares for improved fixed block size iterative search, Test 7 .....	47
3.18 Number of compares for improved fixed block size iterative search, Test 8 .....	48
3.19 Color difference of fixed block size one-shot and exhaustive search.....	50
3.20 Best fitness per generation for fixed block size one-shot search, Test 1 .....	51
3.21 Best fitness per generation for fixed block size one-shot search, Test 2 .....	51
3.22 Best fitness per generation for fixed block size one-shot search, Test 3 .....	52
3.23 Best fitness per generation for fixed block size one-shot search, Test 4 .....	52
3.24 Best fitness per generation for fixed block size one-shot search, Test 5 .....	53
3.25 Best fitness per generation for fixed block size one-shot search, Test 6 .....	53
3.26 Best fitness per generation for fixed block size one-shot search, Test 7 .....	54
3.27 Best fitness per generation for fixed block size one-shot search, Test 8 .....	54
3.28 Variable block size motion estimation, Step 1 .....	56

## LIST OF FIGURES (Continued)

<b><u>Figure</u></b>	<b><u>Page</u></b>
3.29 Variable block size motion estimation, Step 2 .....	57
3.30 Variable block size motion estimation, Step 3 .....	58
3.31 Variable block size motion estimation, Step 4 .....	59
3.32 Color difference of variable block size and exhaustive search .....	60
3.33 Best fitness per generation for variable block size search, Test 1 .....	61
3.34 Best fitness per generation for variable block size search, Test 2 .....	61
3.35 Best fitness per generation for variable block size search, Test 3 .....	62
3.36 Best fitness per generation for variable block size search, Test 4 .....	62
3.37 Best fitness per generation for variable block size search, Test 5 .....	63
3.38 Best fitness per generation for variable block size search, Test 6 .....	63
3.39 Best fitness per generation for variable block size search, Test 7 .....	64
3.40 Best fitness per generation for variable block size search, Test 8 .....	64
3.41 Region motion estimation, Fixed Block Size Method .....	66
3.42 Region motion estimation, Fixed Block Size Method, frame 1 .....	68
3.43 Region motion estimation, Fixed Block Size Method, frame 2 .....	68
3.44 Region motion estimation, Fixed Block Size Method, frame 3 .....	69
3.45 Region motion estimation, Fixed Block Size Method, frame 4 .....	69
3.46 Region motion estimation Variable Block Size Method, Step 1 .....	71
3.47 Region motion estimation Variable Block Size Method, Step 2 .....	72
3.48 Region motion estimation Variable Block Size Method, Step 3 .....	73
3.49 Region motion estimation Variable Block Size Method, Step 4 .....	74

## LIST OF FIGURES (Continued)

<b><u>Figure</u></b>	<b><u>Page</u></b>
3.50 Region motion estimation, Variable Block Size Method, frame 1 .....	75
3.51 Region motion estimation, Variable Block Size Method, frame 2 .....	76
3.52 Region motion estimation, Variable Block Size Method, frame 3 .....	76
3.53 Region motion estimation, Variable Block Size Method, frame 4 .....	77

## LIST OF TABLES

<b><u>Table</u></b>	<b><u>Page</u></b>
2.1 Example GA population.....	15
2.2 Reproduction operation for example GA.....	16
2.3 Crossover operation for example GA.....	18
2.4 Mutation operation for example GP .....	23

## LIST OF APPENDICES

<b><u>Appendix</u></b>	<b><u>Page</u></b>
APPENDIX A.....	90
APPENDIX B.....	94
APPENDIX C.....	98
APPENDIX D.....	99



## **SOFTWARE**

The software used for this research is available upon request. Please contact:

Martin Dill  
11322 SW Meadowlark Lane  
Beaverton, Oregon 97007  
E-mail: [dills@worldnet.att.net](mailto:dills@worldnet.att.net)

## **DEDICATION**

*This thesis is dedicated to:*

*My wife Karen and my daughter Kristina.*

# **Improving Motion Estimation with Evolvable Search Algorithms**

## **CHAPTER 1 - INTRODUCTION**

This research examines the applicability of evolvable search algorithms (Genetic Algorithms and Genetic Programming) to the topic of motion estimation as used in video compression.

Motion pictures have long fascinated people of all walks of life, whether it is television, a motion picture at a local theater, or a video that was brought home for an evening's enjoyment. In recent times a transition in the medium from analog to digital recording technologies has occurred, which has brought with it many problems. One of the largest problems is the sheer volume of data required to adequately represent motion pictures. As a means for processing these large quantities of data, data compression algorithms have been developed. One of the most important requirements of data compression is to store any length of a video sequence, using conventional computer data storage technologies.

Some of the most exciting forms of data compression used for motion pictures are the MPEG (Moving Pictures Expert Group) compression standards, of which MPEG-1 and MPEG-2 are the most common. One of the more interesting parts of the standards deals with motion estimation. Motion estimation is used to avoid storing the image to computer memory a number of times, as an object moves across the screen. The process of motion estimation just stores the image to computer memory once, when it first appears, along with motion vectors which describe its movement over time.

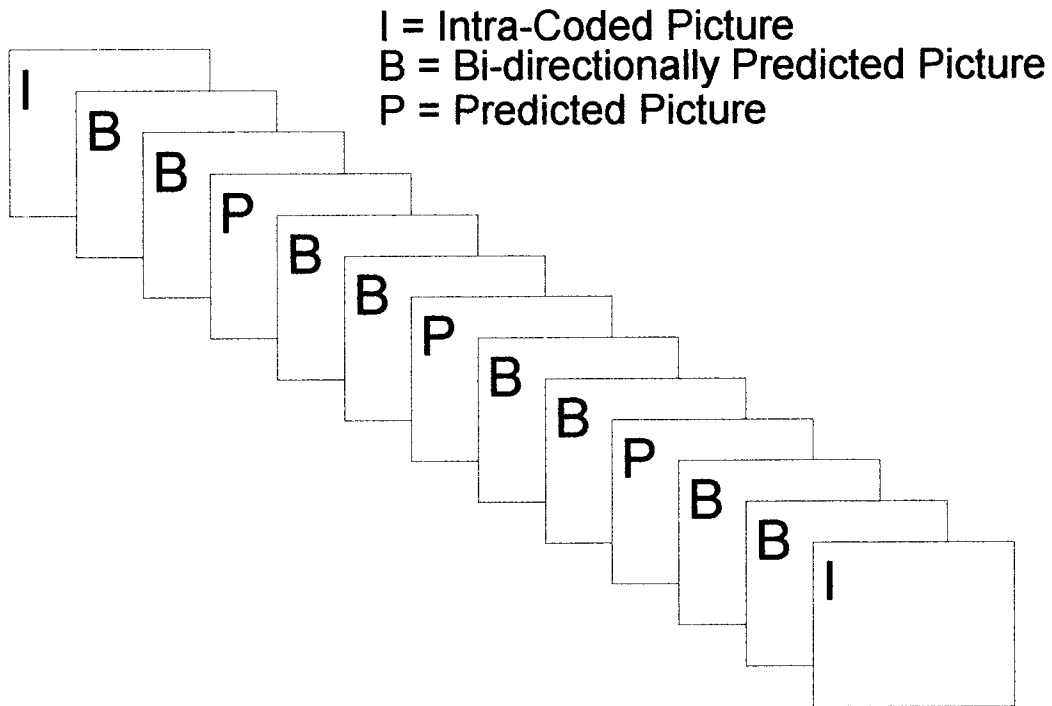
A moving picture is really a sequence of individual still picture frames displayed in sequence one after another. The rate at which these frames are displayed varies. For example a typical motion picture is 24 frames per second while a television image is effectively displayed at about 30 frames per second.

MPEG takes a three dimensional approach to compressing moving pictures. First it can take advantage of intra-frame compression, which means that an individual frame is compressed by itself, in a manner much like that of compressing a digitized photograph. Secondly, MPEG utilizes inter-frame compression, in which a sequence of picture frames are compressed along the temporal axis. It is with these two compression techniques that MPEG achieves its high quality output with minimal data storage requirements.

To implement the inter-frame and intra-frame compression modes, MPEG uses three techniques to encode the individual pictures in a moving picture sequence. These are I, P, and B Frames. The I Frame is the intra-coded frame; this means that it contains all of the data of an individual frame and does not depend on the content of the previous or the following frames. The first frame of any sequence to be displayed must be an I Frame. The P and B Frames are inter-coded frames. The P Frame is a predicted frame and uses motion estimation vectors from the previous I or P Frame (whichever is closer). The B Frame is a bi-directional predicted frame, which uses motion estimation vectors from the previous P or I Frame (whichever is closer) and the following P or I Frame (whichever is closer). Typically a P Frame is about one-third the size of an I Frame and a B Frame is about one-third the size of a P Frame. So from a compression point of view, it is advantageous to use as many P and B Frames as possible.

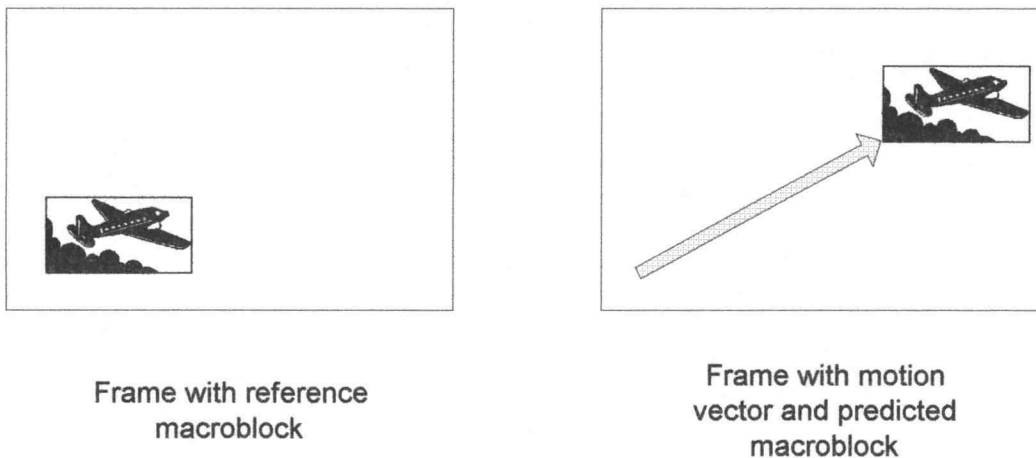
A typical display sequence is illustrated in Figure 1.1. The sequence starts with an I Frame, then displays two B Frames, a P Frame, two B Frames, another P Frame, and finally another I Frame. In this arrangement, an I Frame is displayed every twelve frames. This is a practical upper limit on how many inter-frames can be displayed between intra-coded frames. The reason for this practical upper limit is that when a decoding device starts to decode a stream, a delay of more than half a second before any data can be displayed is generally too much, since the decoder cannot display anything until it receives an I Frame. This issue becomes important in a number of real-world situations. For example when “channel surfing” on a digital television system, which uses MPEG or a similar technology, the consumer does not want to wait very long for the picture to appear after the change of channels.

Figure 1.1 Typical MPEG picture sequence



The temporal compression component is achieved through motion estimation. Motion estimation is done on a macro-block level. (A macro-block is a square block of picture data, 16 by 16 pixels.) This process is performed in the following manner. A macro-block is chosen in the reference frame. If the selected macro-block is part of a moving object, then it may also exist in the next frame, but possibly at a different location and slightly modified. By comparing the reference frame to the predicted frame it is possible to determine the coordinate differences of the two macro-blocks. This difference can be represented as a vector in which the X and Y components represent the horizontal and vertical displacement of the macro-block, respectively. So when the decoder has a reference macro-block along with its associated motion vector, then it can move the macro-block to the proper location in the predicted frame.

Figure 1.2 Motion prediction example



Motion estimation is the single largest determinant in producing a compression ratio which can be achieved in an MPEG data stream of a given video quality. Since motion estimation is a very time consuming and computationally intensive process, the difference between a good and mediocre motion estimator can radically affect the amount of computer memory required for a motion picture, as the compression ratios achieved by a video compressor can be largely improved. This research will explore motion estimation using both a Genetic Algorithm (GA) and a Genetic Program (GP).

## **CHAPTER 2 – PHILOSOPHY AND REVIEW OF LITERATURE**

The topic of motion estimation has a broad application. Motion estimation is not only used in MPEG, but in a number of other video compression methods as well. Additionally, motion estimation is even used in many applications not related to data compression; it is used in any application that needs to determine motion from several individual pictures. For example, motion estimation can be used in an air traffic control system that needs to determine the velocities of aircraft from individual radar images.

MPEG uses motion estimation for macro-blocks. In other words, the motion estimator looks at a square block of pixels and tries to find the best match in the next frame. Other applications may require a different approach, for example they may need to estimate motion for regions or previously identified objects in the image. A different application may use the motion of an object from frame to frame to help identify the shape of the object, i.e. determine the object's boundaries.

### **2.1 Review of Motion Estimation Algorithm Literature**

There are a number of recent research papers available on the subject of motion estimation. This can be attributed to the importance of the topic with today's new digital video mediums. Many of the articles deal directly with improving the estimation of block motion vectors. For example Jehng et al. (1992) implemented a motion estimator in hardware with a motion estimation algorithm called 3HAS. This algorithm takes several steps to find motion vectors by successively narrowing down the search area, three times.



Wong (1995) presents a heuristic based motion estimation technique which reduces both the number of search locations, via sub-sampling, as well as the number of operations to perform at each location, with a simplified signature. This research does not compare actual blocks of pixels, but signatures of blocks of pixels. It is claimed that this process has several advantages: all pixel points contribute to signatures, only sixteen values are needed to represent a block (16x16), and signature values of overlapping blocks can be easily computed. It was shown that this algorithm is 69 times faster than that of an exhaustive search.

Liu and Zaccarin (1993) developed an algorithm that produces the same quality of output as an exhaustive search, but reduces the computation by a factor ranging from 8 to 16. This is accomplished by first determining a sub-sampled motion field by estimating the motion vectors for a fraction of the blocks. Then only a fraction of the pixels at any location are used to determine these vectors. An alternating pattern of sub-sampled pixels are used to help maintain motion vector accuracy. The sub-sampled motion field is then interpolated and a motion vector determined for each block of pixels.

Chan and Siu (1995) improved on the work of Liu and Zaccarin. An alternating pattern of sub-sampled pixels are not used to determine motion vectors, but to actually interpret the pixel data. Pixels that are most representative of the block in which they are found are selected. It was found that high activity in the luminance signal, which indicates edges and texture, is one of the main contributors to the matching criterion. So by examining a pixel and its neighbors, it is possible to determine the main pixels in a block and use those for the block matching process.

One of the newer areas of video compression is region-based motion estimation. Region based motion estimation does not suffer from the blockiness often associated with block based algorithms, such as MPEG, and provides better compression ratios. In region based motion estimation actual image regions are first identified, by various means, and then motion vectors are associated with these regions, rather than with arbitrary blocks of pixels. Dang et al. (1995) proposed a three-step algorithm for region-based representation and motion estimation. In the first step intensity based image segmentation is used to partition an image into various regions. The second step consists of motion estimation for these regions, and in the third step, called region fusion, adjacent regions with the same motion vectors are merged together and the initial boundaries adjusted.

Karczewicz et al. (1995) took a similar approach, but used a polynomial model of a motion vector to allow for more variations in the type of motion (not limited to simple x and y displacements). A method for reducing the number of coefficients of the polynomial without severely impacting the quality of the video was also provided.

Zhang et al. (1995) tackled the problem of image segmentation. In this research, three techniques were combined to overcome the shortcomings of each individual method. The first, motion field based segmentation, yields segmentation that differentiates objects moving with different motions. The second, gray level based segmentation, is robust and results in natural segmentation along object boundaries. The third, change detection, is good at providing an initial coarse segmentation into motion and stationary segments. It was found that the entire algorithm (consisting of these three techniques) worked best when the sequence contained a highly textured background and involved complex motion. Further investigation of this algorithm is needed for various

types of image sequences. This illustrates the complexity of performing image segmentation for motion estimation.

A different approach from the previous literature is taken in this research to solve the motion estimation problem. A GA and GP are implemented to find the appropriate motion vectors to translate objects from one picture frame to the next.

No previous references as to the use of a GA or a GP for motion estimation were found by the author. One of the few applications of a GA to image processing was a study by Cavicchio (1970) in which a GA was used in a pattern recognition problem. In this research, a GA was used to find a set of detectors (or patterns) which would in turn be used to classify an image digitized on a 25 by 25 grid. It was found that despite the enormous search space, the GA considerably outperformed other algorithms for creating detectors. Gritz et al. (1995) used a GP to generate controller programs to animate articulated figures. They had some success with this technique, but found that the generated programs were rather "brittle" (the programs were only suited for a particular task, rather than a general skill, and were sensitive to initial conditions). They are continuing their work in an attempt to make the programs more general and robust.

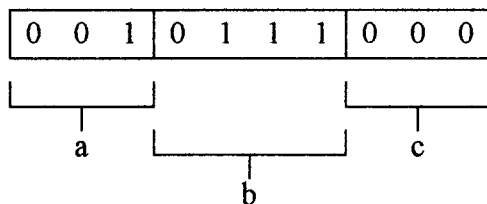
For the first part of this research a GA is utilized to determine simple motion vectors for block based motion estimation. In the second part of the research, a GA and a GP will be used to aid in region-based motion estimation, not just to determine motion vectors, but to actually find the largest possible regions. GAs and GPs provide a certain simplicity to finding solutions to complex problems since there is no need to specify an algorithm for solving the actual problem. All that is required is a way of ranking possible solutions. These types of search algorithms are particularly well suited to the problem of

dealing with image data because of their implicit parallelism, which makes heuristically searching very large search spaces possible.

## 2.2 Introduction to Genetic Algorithms and Genetic Programming

As Darwinian evolution over the millennia has well adapted organisms to the natural environment, these same principles can be applied to evolution in the artificial environment. Within a system of our own creation on a computer, selective breeding techniques and survival of the fittest can work together to create robust artificial organisms. Genetic operators are a means by which artificial evolution is applicable to software enhancement. The Genetic Algorithm (GA) optimizes for problem parameters, while the Genetic Program (GP) derives optimized data structures, in the form of software code. Both the GA and GP have the same genetic operators. However the chromosome structure is different. For a GA, the chromosome “bit string” is composed of fields, which are an encoding for problem parameters. (Refer to Figure 2.1.)

Figure 2.1 Example GA bit string denoting variable fields



Whereas for a GP, the chromosome bit string, which looks similar to that of the GA, represents a data structure. For example the quadratic equation:

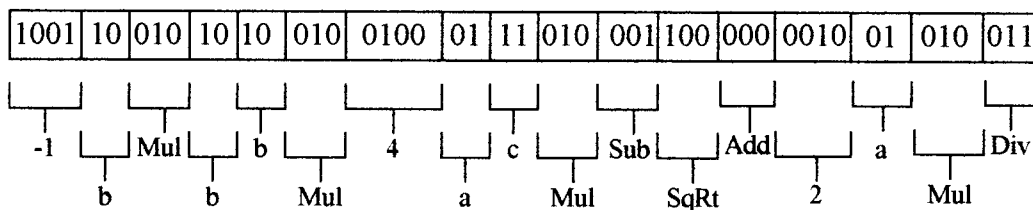
$$ax^2 + bx + c$$

solved for  $x$ , yields:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

consists of mathematical operators and variables and can be represented by the bit string shown in Figure 2.2. The data structure contained within the bit string, is encoded with different fields representing both functions (mathematical and logical operators) and terminals (input variables), in an ordered tree structure.

Figure 2.2 GP bit string of solution program for the positive root of the quadratic equation



Where,

Add = Addition

Sub = Subtraction

Mul = Multiplication

Div = Division

SqRt = Square Root

Equation in Reverse Polish Form:

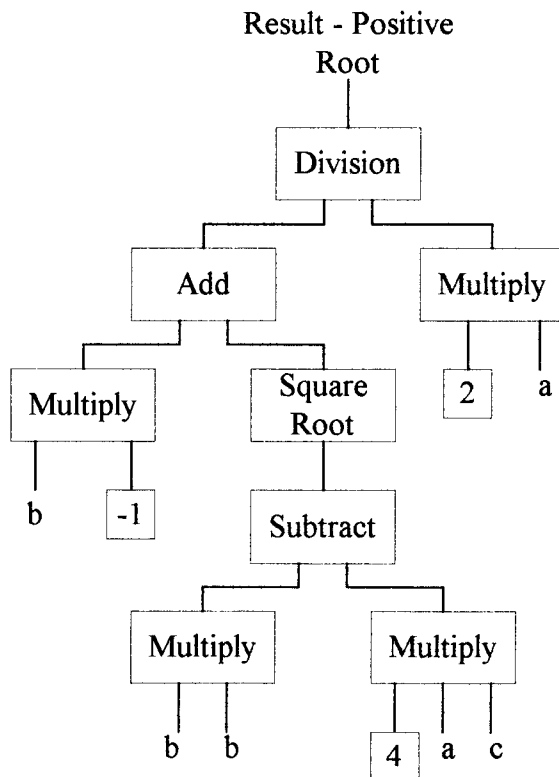
(((-1)(b) multiply) (((b)(b) multiply) ((4)(a)(c) multiply) subtract) sqrt-root) add)  
 ((2)(a) multiply) divide

Simplified Equation in Mathematical Form:

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

With the GP data tree structure, software programs or mathematical/logical equations can be evolved. For example the GP tree illustrating the positive root solution for the quadratic equation encoded in the previous bit string is shown in Figure 2.3.

Figure 2.3 GP tree illustrating the program for the positive root solution of the quadratic equation



The standard, most basic operators for a simple GA/GP are Reproduction, Crossover, and Mutation. The following sections will briefly detail these operations and demonstrate them with an example.

Before the artificial evolution can begin, an initial population of solutions must be created. Population creation can be performed by random binary number generation, user selection of “good” solutions, or some combination of both. The “fitness” or merit of these initial solutions are then evaluated and assigned numeric values. Following the creation of the initial population, the standard genetic operators can be applied for a number of generations.

For demonstration purposes an example is given to illustrate all of the basic operations for a single generation of a GA. Optimization will be performed for the purpose of finding values such that the variable  $x$  is made as large as possible. Note that the GA is not searching for the roots of any specific equation, but is just searching for the largest possible value of  $x$  which it can generate. The fitness function of the GA will simply plug the GA's values into the quadratic equation and return the value of  $x$  as the fitness.

The bit string given encodes the values for the variables  $a$ ,  $b$ , and  $c$  in the positive root solution to the quadratic equation, as shown in Figure 2.1, where the variables  $a$  and  $c$  are restricted to a three-bit representation (values 0-7) and variable  $b$  has a four-bit representation (values 0-15). (For the purposes of this example, several solution restrictions are applied. To avoid division by zero, variable  $a$  is restricted from being equal to zero. Also, the  $b^2-4ac$  root is always positive, to avoid the condition of a negative root.) The example population and corresponding fitness values are given in Table 2.1.



Table 2.1 Example GA population

Population Member	Bit Strings a-b-c	Fitness X
1	001-0111-000	0.0
2	011-1000-100	-0.67
3	101-1001-011	-0.44
4	001-0010-001	-1.0
5	011-0110-010	-0.42
6	100-0111-011	-0.75
7	010-1110-111	-0.54
8	100-1011-101	-0.57
9	111-1100-010	-0.18
10	111-1101-110	-0.86

Total Fitness = -5.43

Average Fitness = -0.54

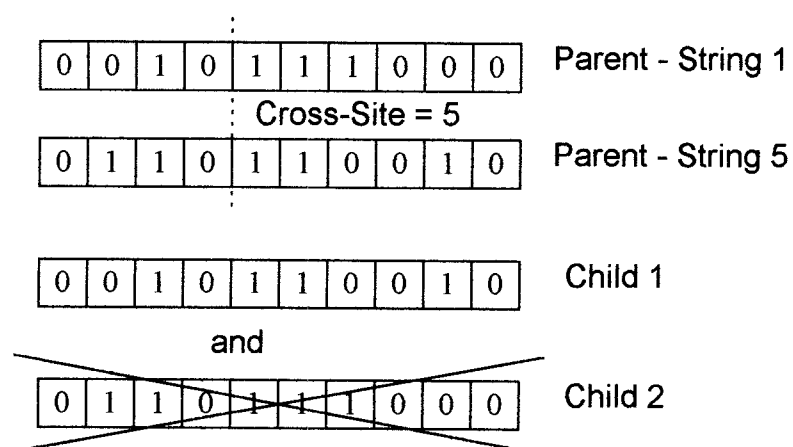
To begin breeding solutions in the artificial environment, the Reproduction operator must be applied. In Reproduction, parent strings are selected. There are a number of methods for implementing this operator. However, one of the most common is Tournament Selection. Loosely based on wild animal mating contests, in Tournament Selection two randomly selected strings “duel” each other based on fitness. The string with the larger fitness is declared the “winner” and thus is eligible to produce offspring. This process is illustrated in the continuation of the previous example, in Table 2.2.

Table 2.2 Reproduction operation for example GA

<b>Reproduction Case</b>	<b>Tournament Selection</b>	<b>New Generation (Winners)</b>
1	String 1 vs. 4	String 1
2	String 9 vs. 10	String 9
3	String 5 vs. 6	String 5
4	String 1 vs. 2	String 1
5	String 3 vs. 7	String 3
6	String 8 vs. 1	String 1
7	String 2 vs. 5	String 5
8	String 4 vs. 3	String 3
9	String 6 vs. 8	String 8
10	String 10 vs. 9	String 9

The Crossover operator actually produces offspring. In this process, two strings from the pool of eligible parents, (the results of the Reproduction operation), are randomly selected. Then, at randomly selected bit sites, the two strings exchange “chromosomes”, thus forming two offspring. For example, as shown in detail in Figure 2.4, String 1 and String 5 mate. Their cross site is randomly selected as bit position 5 (counting from the right, starting with zero). The results of this mating are two child strings. However, in this particular implementation, the second child is always rejected, so as to maintain the initial population size. (Both “children” strings could be selected if the mating process were reduced by half. Both methods are commonly employed.)

Figure 2.4 Detail of Crossover operation



The Crossover operation is continued in the same manner for the entire population.

Mate strings and cross-sites are chosen randomly. This is illustrated in the continuation of the previous example, shown in Table 2.3. Notice that the total fitness values have improved after the crossover operation from that of the original population, from -5.43 to -3.87, respectively.

Table 2.3 Crossover operation for example GA

New Generation (Offspring)	String Population	Chosen Mate	Mate String	Cross-Site	Result String	Fitness of Result
String 1	001-0111-000	String 5	011-0110-010	5	001-0110-010	-0.35
String 9	111-1100-010	String 1	001-0111-000	0	111-1100-010	-0.18
String 5	011-0110-010	String 6	100-0111-011	6	011-0111-011	-0.57
String 1	001-0111-000	String 4	001-0010-001	1	000-0111-001	-0.15
String 3	101-1001-011	String 8	100-1011-101	8	100-1011-101	-0.57
String 1	001-0111-000	String 9	111-1100-010	3	001-0110-010	-0.35
String 5	011-0110-010	String 2	011-1000-100	7	011-1000-100	-0.67
String 3	101-1001-011	String 1	001-0111-000	4	101-1011-000	0
String 8	100-1011-101	String 7	010-1110-111	8	110-1110-111	-0.73
String 9	111-1100-010	String 3	101-1001-011	2	111-1100-011	-0.3

Total Fitness (after Crossover) = -3.87

Average Fitness (after Crossover) = -0.387

In a GP, the bit string crossover is performed in the same manner, but with rules maintaining the tree structure. For the GP tree structure, an entire tree branch, starting from the node chosen for the crossover reference and proceeding to the root of the tree, is swapped and joined to the selected node of the opposite parent. For example, with the solution to the quadratic equation as the goal of the GP breeding process, let us observe the breeding of the following trees in Figures 2.5 and 2.6, where the branches selected for the crossover exchange are indicated by the dashed lines. The resulting “child” trees are shown in Figures 2.7 and 2.8.

Figure 2.5 Parent 1 in Crossover operation for example GP

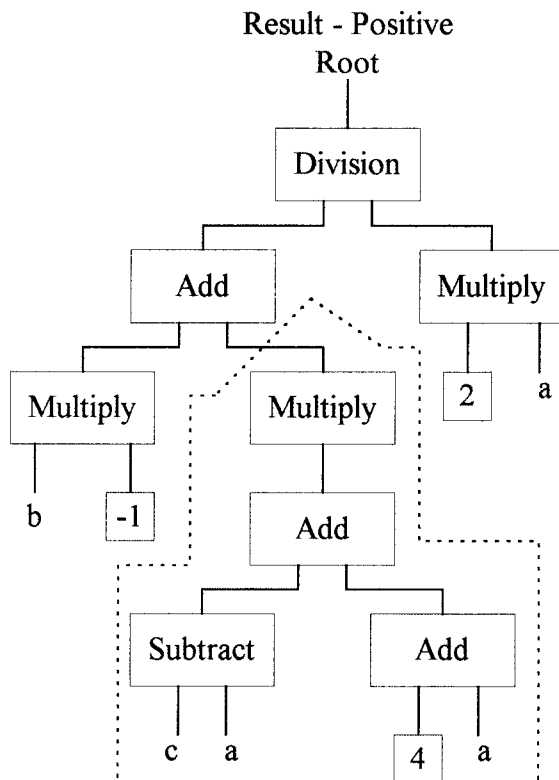


Figure 2.6 Parent 2 in Crossover operation for example GP

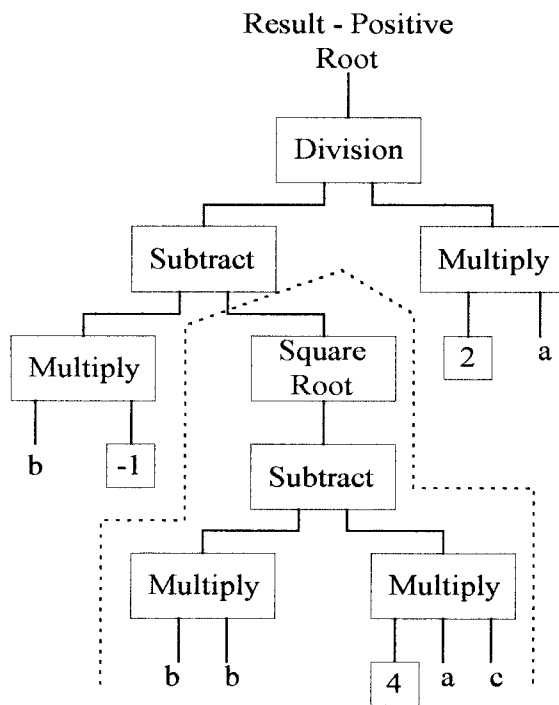


Figure 2.7 Child 1 in Crossover operation for example GP

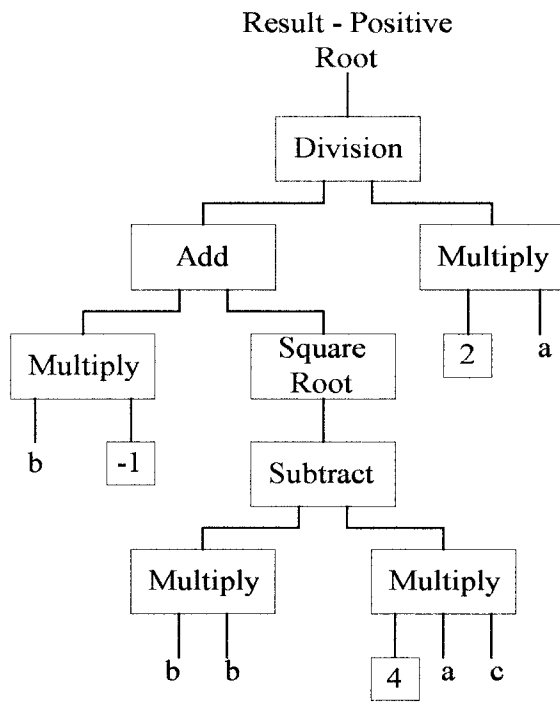
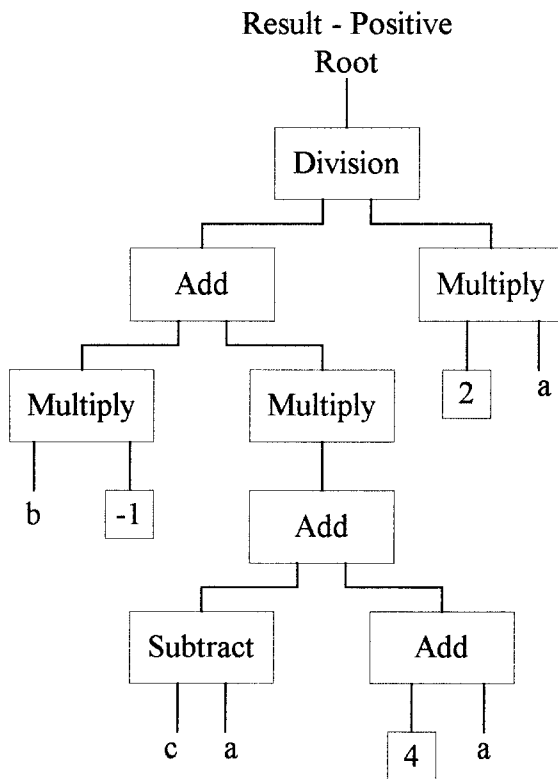


Figure 2.8 Child 2 in Crossover operation for example GP



The final standard operator for GA and GP is Mutation. The Mutation operator introduces a genetic diversity factor. Mutations are random changes in the binary genetic code, which can introduce some non-inherited characteristics, particularly important as a population may be converging prematurely. Generally this operation is performed by selecting at a given probability, a single bit, from all the bit strings in the population, and inverting it. In a binary system, the inversion changes a 0 logic state to a 1, and vice versa. In GP, as a precaution against undefined tree structures, the code is written such that when the inversion takes place, a function is exchanged only for another function and a terminal is exchanged only for another terminal. A number of GP



studies have shown that effective rates of mutation are small<sup>1,2,3</sup>. As “survival of the fittest” determines the outcome of the evolutionary process, the effects of mutation which are shown to be harmful are quickly eliminated, whereas those which are beneficial are propagated through the population. Continuing with the previous example, the mutation probability rate is arbitrarily set at 1/100 bits. Therefore in Table 2.4 it is shown that a single bit, within the population of ten bit strings of ten bits each, is inverted. Note that in this example, the total and average fitness values have been improved by the Mutation operator.

Table 2.4 Mutation operation for example GP

Crossover Population	Result Fitness	Mutation String	Gen=1 Population	Fitness
001-0110-010	-0.35	xxx-xxxx-xxx	001-0110-010	-0.35
111-1100-010	-0.18	xxx-xxxx-xxx	111-1100-010	-0.18
011-0111-011	-0.57	xxx-xxxx-xxx	011-0111-011	-0.57
000-0111-001	-0.15	xxx-xxxx-xxx	000-0111-001	-0.15
100-1011-101	-0.57	xxx-xxxx-xxx	100-1011-101	-0.57
001-0110-010	-0.35	xxx-xxxx-xxx	001-0110-010	-0.35
011-1000-100	-0.67	xxx-xxxx-0xx	011-1000-000	0
101-1011-000	0	xxx-xxxx-xxx	101-1011-000	0
110-1110-111	-0.73	xxx-xxxx-xxx	110-1110-111	-0.73
111-1100-011	-0.3	xxx-xxxx-xxx	111-1100-011	-0.3

Total Fitness (after Mutation) = -3.2

Average Fitness (after Mutation) = -0.32

<sup>1</sup> John R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection (Cambridge: The MIT Press, 1992), pp. 599-600.

<sup>2</sup> Koza, pp. 105-106.

<sup>3</sup> David E. Goldberg, Genetic Algorithms in Search, Optimization, & Machine Learning (New York: Addison-Wesley Publishing Company, Inc.), p. 33.

With the completion of the Reproduction, Crossover, and Mutation operations, a new generation is created. The artificial evolution of a GA or a GP is conducted over the “millennia”, as the process is run for a number of generations. This number of generations can be either preset or based on the optimization performance. Through the simple example given, it is shown that the genetic operators improve the merit of the encoded solutions. The formal theory demonstrating the effectiveness of the genetic process is given as the Schemata Theorem<sup>4</sup> and is based on probability and pattern recognition.

### 2.3 Rationale for the Effectiveness of Genetic Operators

There is much information, about the characteristics of effective and non-effective solutions, present within the genetic population. Encoded within each generation is a history of its “race”. This history pertains to the successes and failures of solution characteristics, inherent in the genetic codes (or chromosomes) of the ancestors, which are pronounced in the current generation. Selective breeding pressures cause the combination of the successful characteristics to be combined, as strings of high merit breed with others of like fitness.

The characteristics in the genetic codes, which are transmitted through breeding, are represented with bit patterns known as schemata. (Schema is the singular form; schemata is the plural form.) In the binary system, all the elements of bit strings can be represented by the following set  $\{0, 1, *\}$ , where \* represents a don’t care, but must be composed of the elements (0,1). For example, given the schemata  $\{000*1**11\}$ , Figure 2.9 shows the strings which are represented. The matching elements are underlined.

---

<sup>4</sup> Goldberg, pp. 30-33.

Figure 2.9 Schemata example

Strings	Schemata
<u>000010011</u>	<u>000*1**11</u>
<u>000010111</u>	
<u>000011011</u>	
<u>000011111</u>	
<u>000110011</u>	
<u>000110111</u>	
<u>000111011</u>	
<u>000111111</u>	

For a string with length  $l$  and cardinality  $k$  ( $k=2$  for a binary encoding), there are  $(k+1)^l$  schemata. For a population of size  $n$ , there is an upper bound on the number of schemata as  $n2^l$ , where the bound depends on the particular population's diversity. It is these schemata which are processed in parallel through the application of the genetic operators.

The standard genetic operators of Reproduction, Crossover, and Mutation all effect the propagation of schemata to the next generation, but in varying degrees. As the Reproduction operator determines the pool of potential parents, it is the most important determinant of schemata propagation. With the implementation of a selection bias based on higher fitness, it is probable that these "good" parents will produce good or better offspring. The inherent qualities which make these selected parent strings superior are contained within their schemata, and as such, should be combined with the good characteristics of their mate, producing offspring which inherit a number of high-quality characteristics. Offspring of poor quality should die out quickly, and due to the high-performance breeding pressures, the poor characteristics inherent in their schemata should decline in number throughout the population rapidly. Through the breeding

process, the propagation and decline of good and bad traits in the schemata can be summarized numerically. “A particular schema grows as the ratio of the average fitness of the schema to the average fitness of the population.”<sup>5</sup> The Reproductive Schema Growth Equation<sup>6</sup> is given for all the schemata existing at one time within a population, as the following:

$$m(H,t+1) = m(H,t) f(H) / \text{avg. } f$$

where,

$m(H,t+1)$        $m$  samples of a unique schema  $H$ , within a population  $A(t)$ , at a time  $t+1$

$m(H,t)$            $m$  samples of schema  $H$  at time  $t$

$f(H)$              average fitness of the strings in schema  $H$ , at time  $t$

avg.  $f$             average population fitness

Thus, it can be demonstrated that through the genetic operators of Reproduction, Crossover, and Mutation the “good” schemata propagate through the population and increase over a number of generations. Consequently the solution traits of high merit are increased in the artificial evolution.

---

<sup>5</sup> Goldberg, p. 30.

<sup>6</sup> Ibid.

## CHAPTER 3 - THESIS CONTRIBUTIONS AND VERIFICATIONS

### 3.1 Experimental Overview

In this research several aspects of applying genetic search methodologies to the motion estimation problem will be explored. The genetic search methodologies will be used to determine motion vectors and motion regions in place of other search methodologies in order to determine their appropriateness to the problem. Fixed block size motion estimation, variable block size motion estimation, and region motion estimation are the areas which will be examined.

The software for the tests includes a C++ implementation of a GA and a GP (refer to Appendix D). These programs are used in the following experiments:

- Block motion estimation with fixed block sizes
- Block motion estimation with variable block sizes
- Region based motion estimation

These tests provide a diverse representation of the motion estimation problem and as such, should provide adequate results indicating whether this research should be pursued further.

### 3.2 Block Motion Estimation with Fixed Block Sizes

The first area of investigation will be the simple block motion estimation, as is used in MPEG and other video compression formats. This is a good starting point because it is fairly simple and the brute force algorithm (exhaustive search) is straightforward, which makes it a good reference for comparison purposes. It is also computationally intensive and thus can benefit from any improvement.

When evaluating motion vectors it must be noted that there are two separate frames, the current frame and the previous frame. The current frame is the starting point. It is divided into blocks for which matches are found in the previous frame. This guarantees coverage of the current frame. If the reference were taken in the opposite direction, dividing the previous frame into blocks and finding matches in the current frame, the current frame may not have the best coverage. This is because there may be more than one correct motion vector for a block. For example, if the previous frame and the current frame both have a large area which is a uniform color or pattern and the actual motion block size is smaller than this pattern, there may be more than one correct motion vector to correlate the blocks between the frames. Therefore the evaluation of motion vectors starts from the current frame and works backwards.

A generic GA is utilized for the motion estimation. It is quite rudimentary, making use of the genetic operators of Reproduction, single point Crossover (previously described), and Mutation. These basic operations are implemented in the same method as is show in the example of Section 2. While a number of other implementation methods exist for the genetic operators, the basic GA has been shown to be quite robust, and there is rarely a need for more exotic features in these operations.

### **3.2.1 Fixed Size Block Motion Estimation, Iterative Method**

The purpose of this experiment is to examine how well a GA can determine the motion vectors to correlate two video frames using fixed size blocks. An iterative approach will be used in which only one motion vector will be computed at a time.

The first step is to divide the video frame into equal sized blocks; blocks of eight pixels by eight pixels will be used. A GA is then instantiated for every block and tries to find a matching block in the previous frame. Every instantiation of the GA uses a population size of 50 and runs a maximum of 20 generations. The GA uses tournament selection, a crossover probability of 0.6, and a mutation probability of 0.0005.

It was chosen to encode the motion vectors into eight-bit strings for the GA. The first four bits are for the horizontal motion vector and the last four bits are for the vertical motion vector. Each four bit vector can have a value from 0 to 15, but since negative as well as positive values are necessary, a 7 is subtracted from the vector value to give the actual displacement value of the motion vector. Thus motion vector ranges are from  $-7$  to  $+8$  for both the horizontal and vertical components. There is no need to allow the motion vectors to have larger values, since the motion from one frame to the next is typically small. Larger vectors would increase the search space, making it more difficult to find correct vectors. The GA fitness function uses these motion vector values to determine merit. This is done by comparing the selected block in the current frame to the same block in the previous frame, displaced by the motion vectors. The red, green, and blue values of the pixels are subtracted, their absolute values taken, and then summed to generate an error value. Since the goal of the GA is to maximize the fitness values, the error value is multiplied by a  $-1$ .

This is very similar to the brute force, exhaustive search, approach with the only difference being that a GA is utilized to find the motion vectors. In both cases the motion vectors for the blocks are determined sequentially, one at a time.

The experimentation for the Fixed Block Size Motion Estimation consisted of eight individual tests. Each test consists of finding the motion vectors to correlate blocks between two consecutive video frames using the method described above. The first four tests use consecutive frames from a video of a woman singing (refer to Appendix A), while the following four tests use consecutive frames from a video sequence of a marching army (refer to Appendix B).

The results of these tests are analyzed in two ways. The first measure of merit simply rates the quality of the end result (i.e., are the motion vectors worthwhile?). This is calculated with the following formula:

$$\sum_{y=0}^{256} \sum_{x=0}^{320} |cR[y][x] - gR[y][x]| + |cG[y][x] - gG[y][x]| + |cB[y][x] - gB[y][x]|$$

Where:

$cR[y][x]$  is the red component of the specified pixel in the current frame

$gR[y][x]$  is the red component of the specified pixel in the generated frame

$cG[y][x]$  is the green component of the specified pixel in the current frame

$gG[y][x]$  is the green component of the specified pixel in the generated frame

$cB[y][x]$  is the blue component of the specified pixel in the current frame

$gB[y][x]$  is the blue component of the specified pixel in the generated frame

This formula simply sums the absolute values of the differences of the red, green, and blue pixel components between the original current frame and the current frame which was built by applying the generated motion vectors to the previous frame. The second measure of merit is the computation time necessary to calculate the motion vectors. The simplest way of determining this is to record the number of times an individual block in



the current frame was compared to a block in the previous frame, as this gives a close approximation of the amount of calculations performed, which can easily be compared to an exhaustive search.

Figure 3.1 illustrates the merit of the motion vectors. It shows the color differences between the current video frame and the video frame generated by the GA's motion vectors (applied to the previous frame). This color difference (error) value is displayed for each of the eight tests. In addition, the color difference between the current video frame and the video frame of the exhaustive search's motion vectors is also displayed. As can be seen, the images generated by the GA's motion vectors have a substantially higher error than the images generated by the exhaustive search's motion vectors.

Figure 3.1 Color difference of fixed block size iterative and exhaustive search

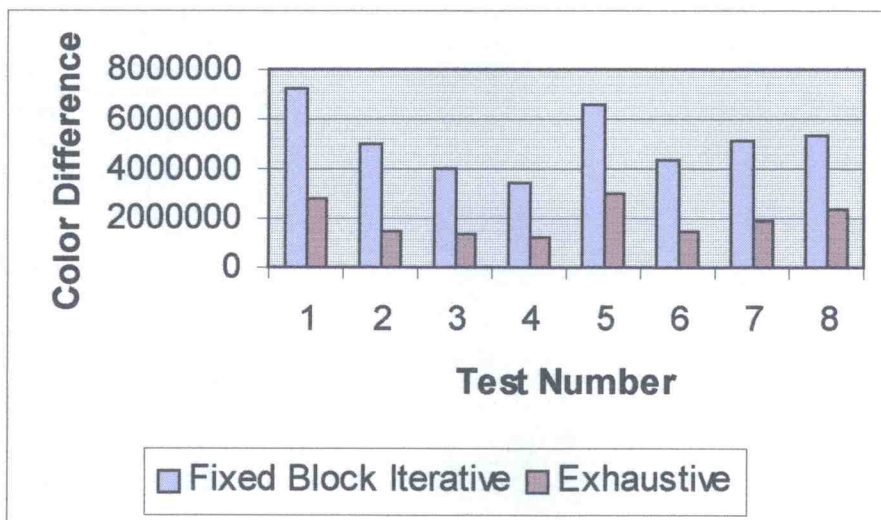


Figure 3.2 shows the number of compares performed for each block in Test 1. The horizontal axis shows the block number (only one block is being processed at a time), while the vertical axis shows the number of block compares performed until the best motion vector for that block was found. The average number of compares performed for each block was 98.86. Since the exhaustive search performs 64 compares for every block, these results are not very good.

Figure 3.2 Number of compares for fixed block size iterative search, Test 1

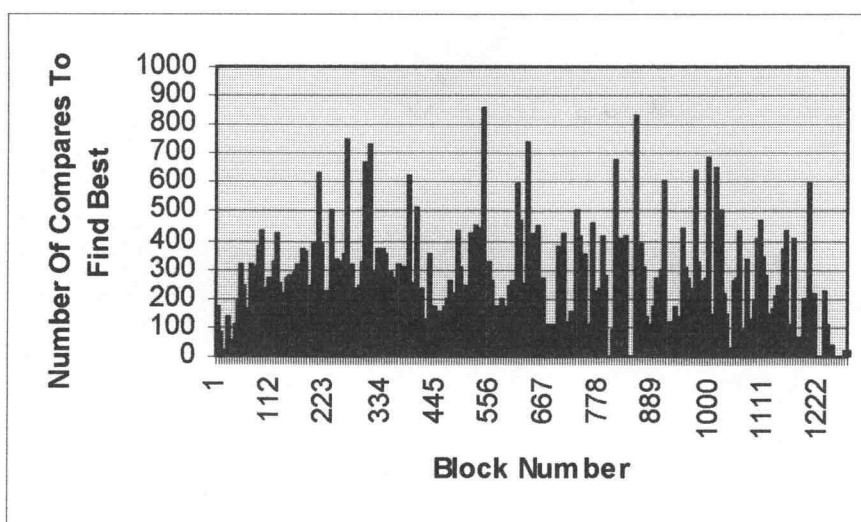


Figure 3.3 shows the number of compares performed for each block in Test 2, until the best motion vectors were found. The average number of compares performed for each block was 97.40.

Figure 3.3 Number of compares for fixed block size iterative search, Test 2

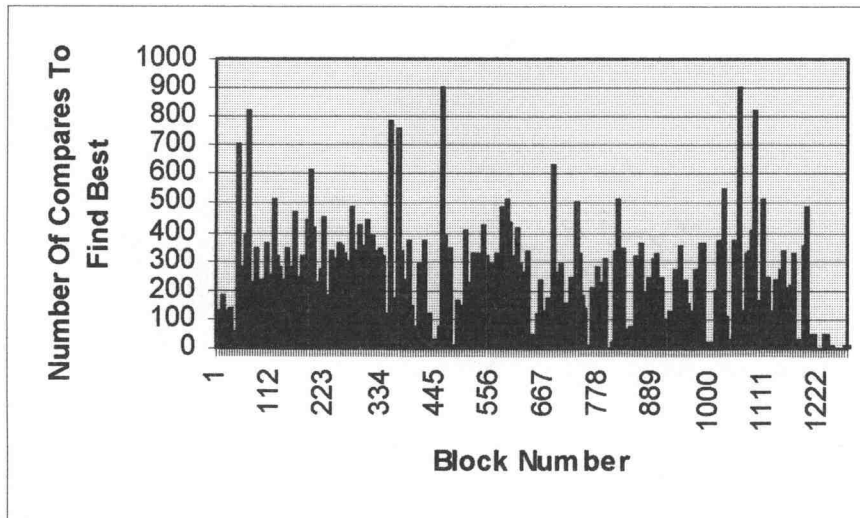


Figure 3.4 shows the number of compares performed for each block in Test 3, until the best motion vector was found. The average number of compares performed for each block was 105.04.

Figure 3.4 Number of compares for fixed block size iterative search, Test 3

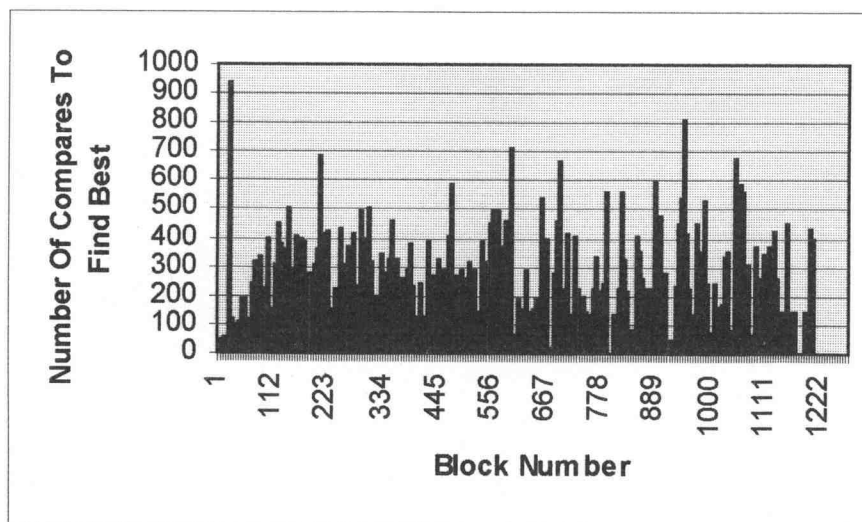


Figure 3.5 shows the number of compares performed for each block in Test 4, until the best motion vector was found. The average number of compares performed for each block was 94.87.

Figure 3.5 Number of compares for fixed block size iterative search, Test 4

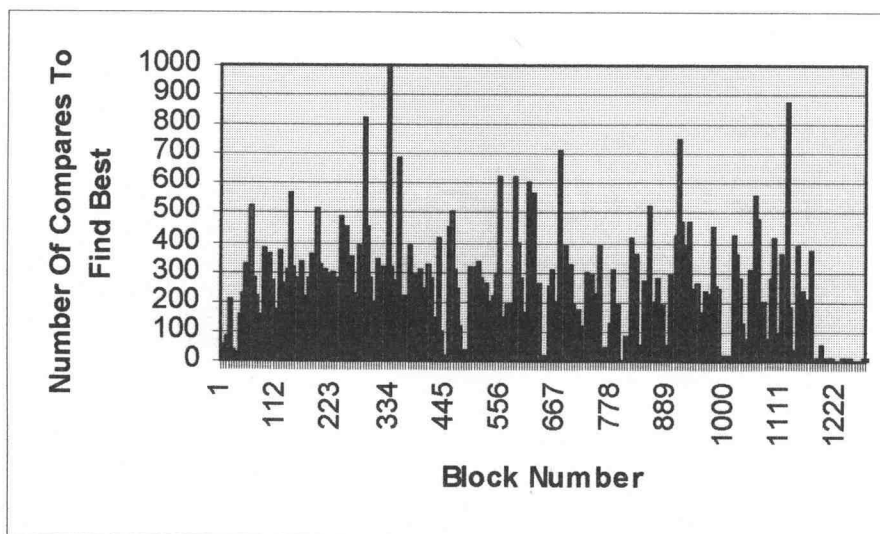


Figure 3.6 shows the number of compares performed for each block in Test 5, until the best motion vector was found. The average number of compares performed for each block was 104.53.

Figure 3.6 Number of compares for fixed block size iterative search, Test 5

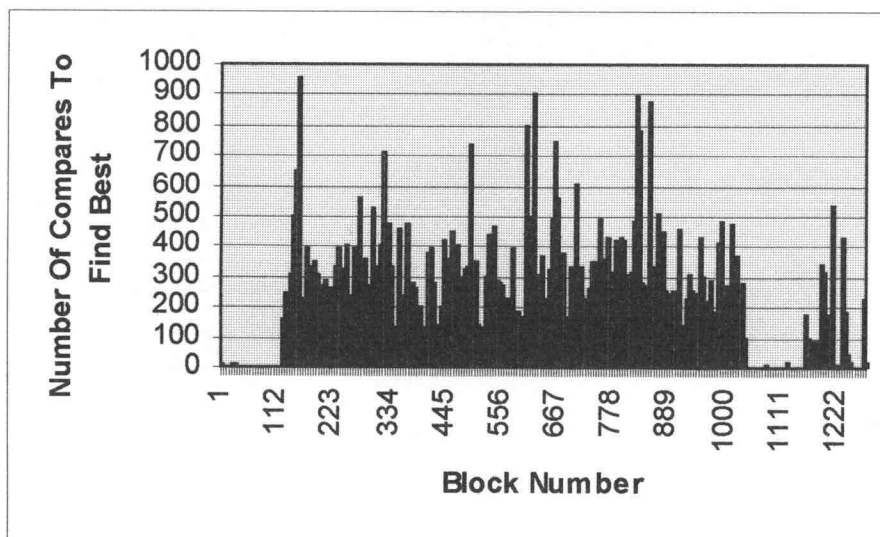


Figure 3.7 shows the number of compares performed for each block in Test 6, until the best motion vector was found. The average number of compares performed for each block was 99.37.

Figure 3.7 Number of compares for fixed block size iterative search, Test 6

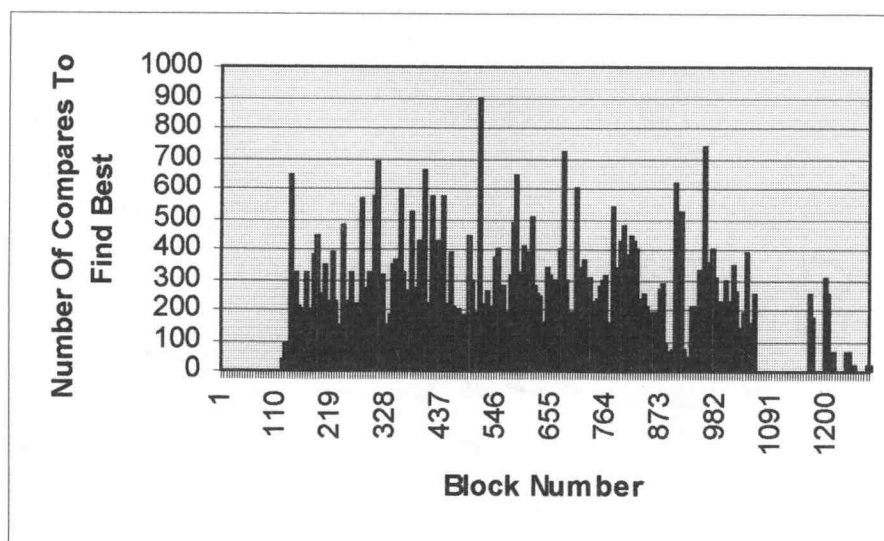


Figure 3.8 shows the number of compares performed for each block in Test 7, until the best motion vector was found. The average number of compares performed for each block was 97.52.

Figure 3.8 Number of compares for fixed block size iterative search, Test 7

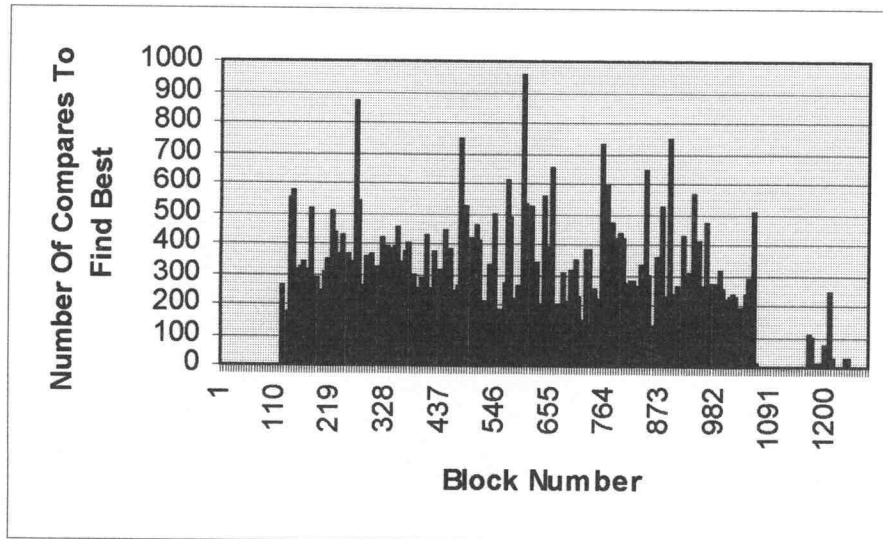
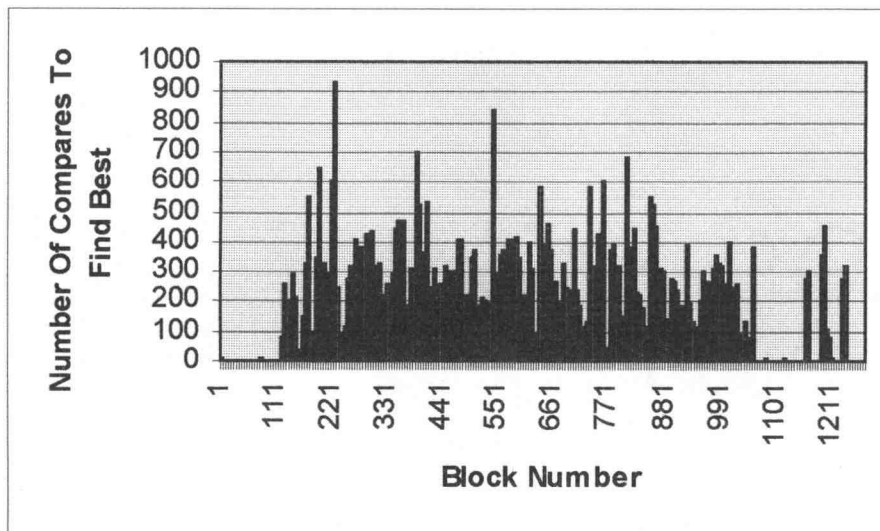


Figure 3.9 shows the number of compares performed for each block in Test 8, until the best motion vector was found. The average number of compares performed for each block was 97.36.



Figure 3.9 Number of compares for fixed block size iterative search, Test 8



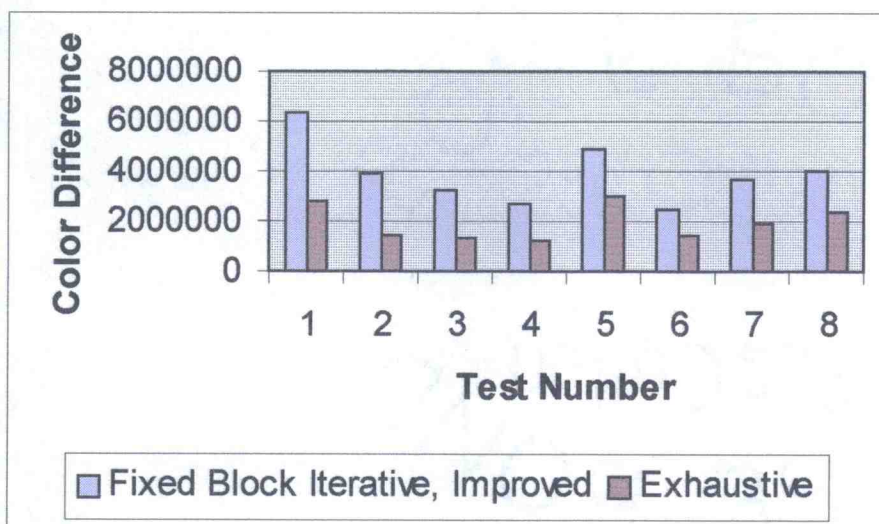
The results of this experiment did not give very promising results for the GA implementation of the Fixed Size Block Motion Estimation with the Iterative Method. The exhaustive search outperformed the GA in both execution speed (the exhaustive search uses 64 compares per block, while the GA uses on average 99.37) and in the quality of results (refer to Figure 3.1). It is, of course, not possible to outperform the quality of the exhaustive search, since it will always find the best solution. A good algorithm, however, should closely approximate the results of the exhaustive search.

One technique commonly used in GAs is to “seed” the initial population with some known good values. The previous experiment initialized the population with random values. The following experiment initializes 2 out of 50 (4%) members of the initial population with null vectors, i.e. vectors of the original, unmoved image.

Figure 3.10 illustrates the merit of the evolved motion vectors. The color differences between the current video frame and the video frame generated by the GA’s

motion vectors (applied to the previous frame) are shown. This color difference (error) value is displayed for each of the eight tests. In addition, the color difference between the current video frame and the video frame generated by the exhaustive search's motion vectors is also displayed. However, as can be seen, the images generated in this experiment have a substantially higher error than the images generated by the exhaustive search's motion vectors.

Figure 3.10 Color difference of improved fixed block size iterative and exhaustive search



Comparing these results to Figure 3.1, there is about a 25% improvement in the color difference values of the video frames generated by the improved GA's motion vectors.

Figure 3.11 shows the number of compares performed for each block in Test 1, until the best motion vector was found. The average number of compares performed for each block was 89.70.

Figure 3.11 Number of compares for improved fixed block size iterative search, Test 1

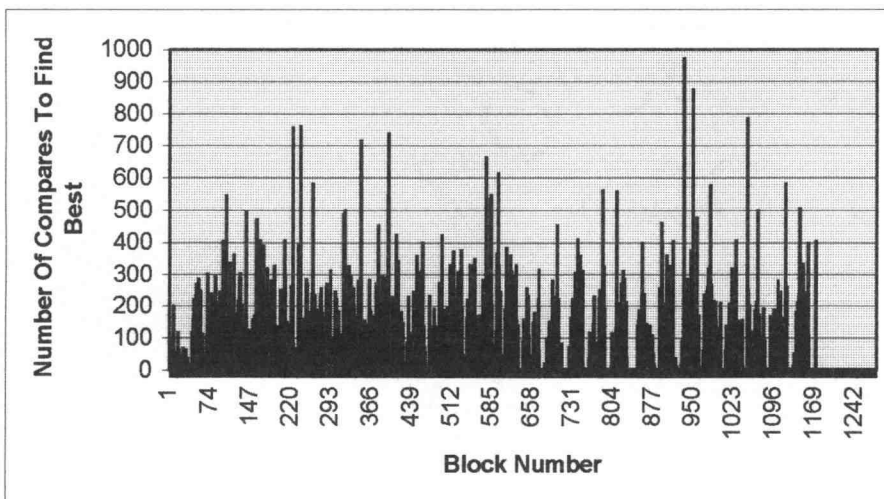


Figure 3.12 shows the number of compares performed for each block in Test 2, until the best motion vector was found. The average number of compares performed for each block was 47.94.

Figure 3.12 Number of compares for improved fixed block size iterative search, Test 2

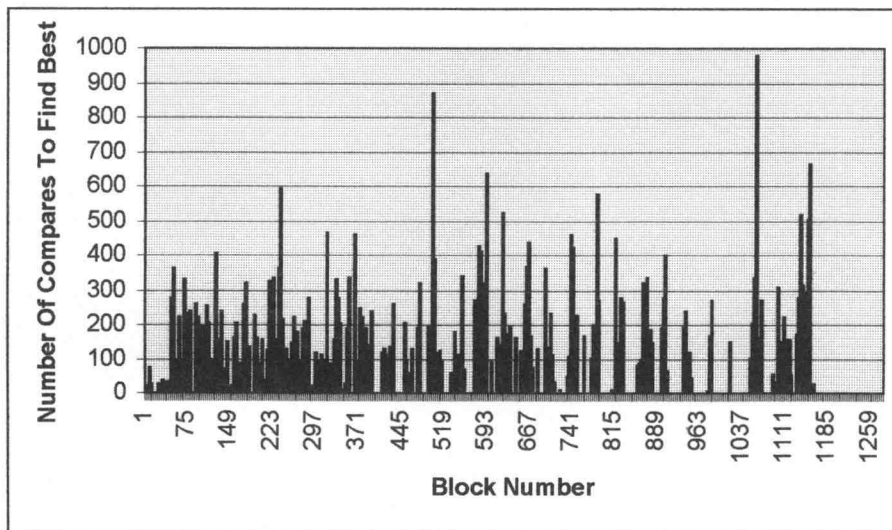


Figure 3.13 shows the number of compares performed for each block in Test 3, until the best motion vector was found. The average number of compares performed for each block was 43.60.

Figure 3.13 Number of compares for improved fixed block size iterative search, Test 3

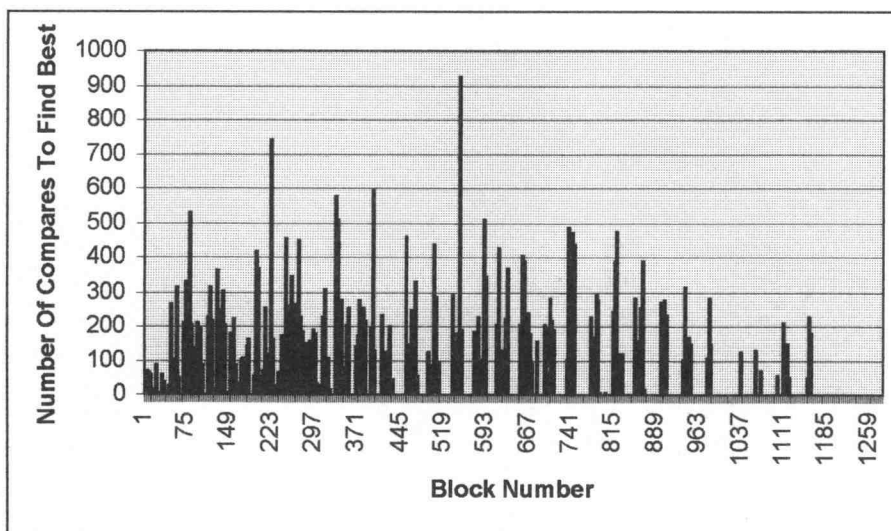


Figure 3.14 shows the number of compares performed for each block in Test 4, until the best motion vector was found. The average number of compares performed for each block was 37.77.

Figure 3.14 Number of compares for improved fixed block size iterative search, Test 4

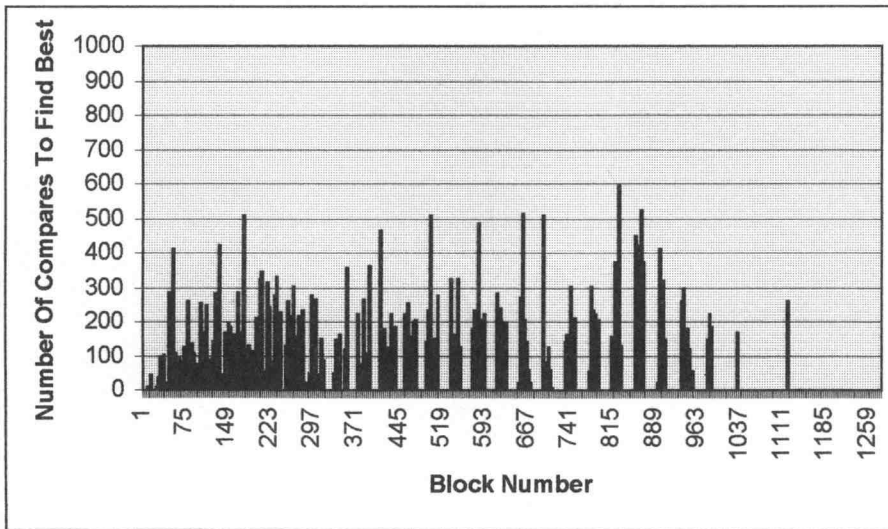


Figure 3.15 shows the number of compares performed for each block in Test 5, until the best motion vector was found. The average number of compares performed for each block was 80.41.

Figure 3.15 Number of compares for improved fixed block size iterative search, Test 5

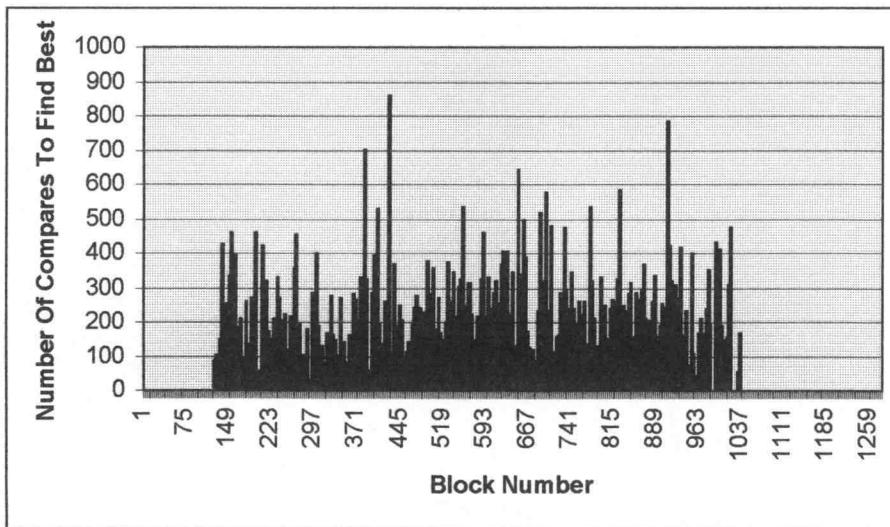


Figure 3.16 shows the number of compares performed for each block in Test 6, until the best motion vector was found. The average number of compares performed for each block was 39.21.

Figure 3.16 Number of compares for improved fixed block size iterative search, Test 6

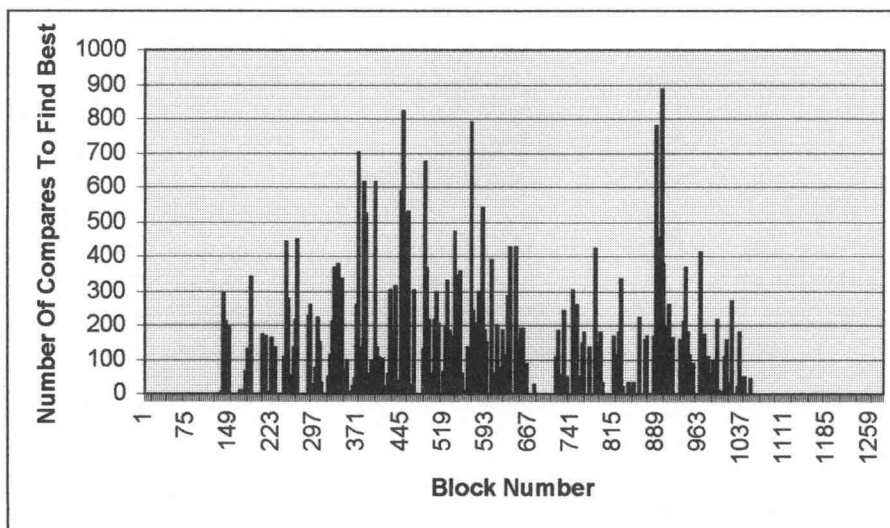


Figure 3.17 shows the number of compares performed for each block in Test 7, until the best motion vector was found. The average number of compares performed for each block was 62.81.



Figure 3.17 Number of compares for improved fixed block size iterative search, Test 7

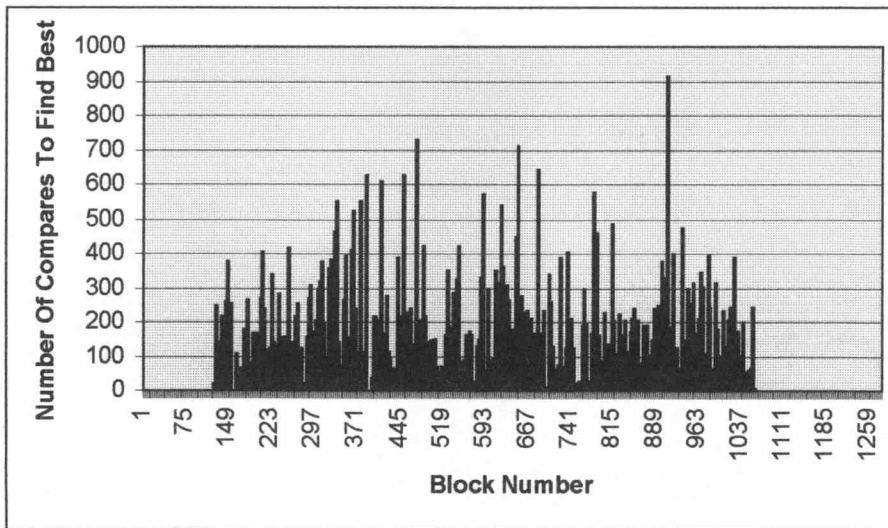
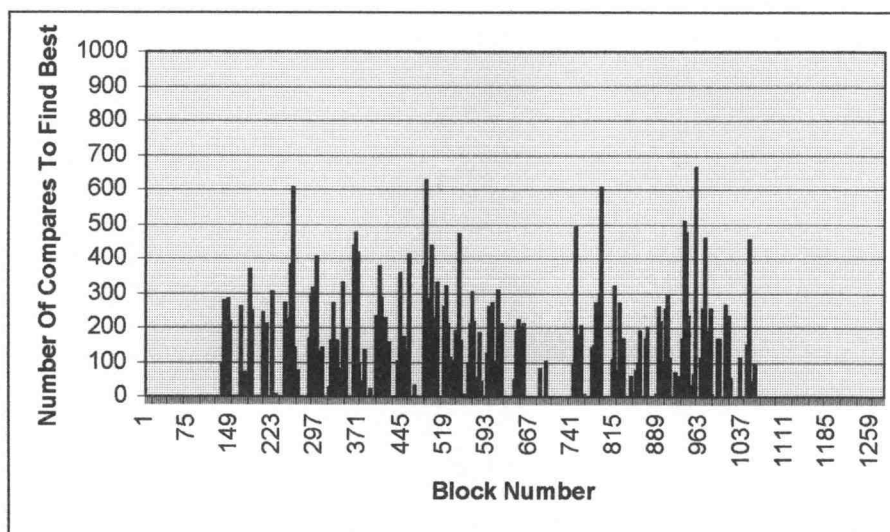


Figure 3.18 shows the number of compares performed for each block in Test 8, until the best motion vector was found. The average number of compares performed for each block was 33.41.

Figure 3.18 Number of compares for improved fixed block size iterative search, Test 8



Figures 3.11 through 3.18 show quite an improvement over the original Fixed Block Size GA search method. The average number of blocks compared until the best was found was 54.35, which is better than the 64 compares required for each exhaustive search block.

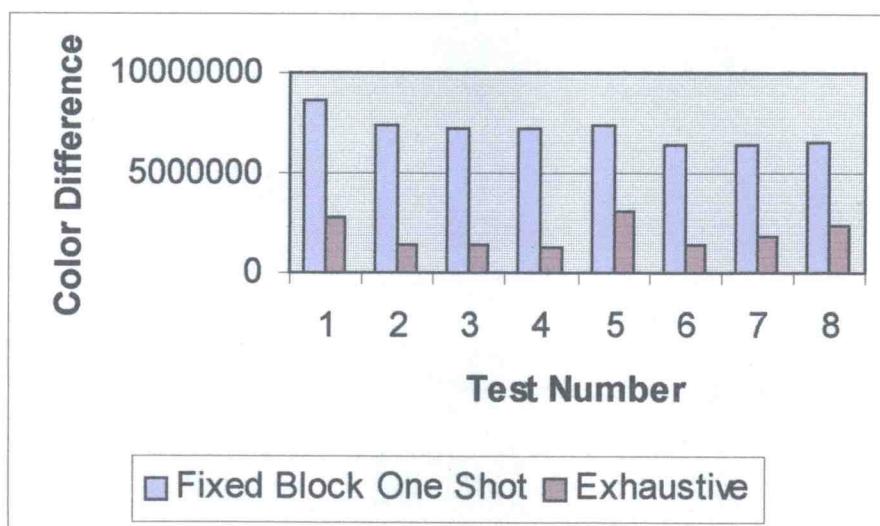
### **3.2.2 Fixed Size Block Motion Estimation, One-Shot Method**

In the previous section a separate GA was used to find the motion vectors for each individual block. But, instantiating a GA for every block is inefficient. A better approach would be to use the GA to search for all of the motion vectors simultaneously. To accomplish this, a modified version of the string encoding used in the previous section is implemented. For this experiment, each string consists of a series of eight-bit strings concatenated together. The number of eight-bit strings concatenated together corresponds to the number of blocks in the current image. For an image consisting of 1280 blocks (40 x 32), the string length will be 8 x 1280 or 10240 bits. (Note that this is

a fairly small image, so the strings can get very long). The fitness function is also modified slightly so that it evaluates the fitness of the motion vectors for each individual block and then sums all of these values together into a total fitness value for the entire image. The GA uses tournament selection, a crossover probability of 0.6, and a mutation probability of 0.0005. The population size was increased to 500 and the number of generations was increased to 50. Test 1 through Test 8 (refer to Appendix A and Appendix B) will be used in this experiment.

Figure 3.19 illustrates the merit of the evolved motion vectors. The color differences between the current video frame and the video frame generated by the GA motion vectors (applied to the previous frame) are shown. This color difference (error) value is displayed for each of the eight tests. In addition, the color difference between the current video frame and the video frame generated by the exhaustive search's motion vectors is also displayed. As can be seen, the images generated by using the GA generated motion vectors have a substantially higher error than the images generated by the exhaustive search's motion vectors. This GA experiment did not perform as well as the GA implementation of Figure 3.10. Although the software is run for more generations, these results are understandable since in this experiment the GA must optimize a 10240 bit string. A much greater amount of computation is required than in the previous experiments.

Figure 3.19 Color difference of fixed block size one-shot and exhaustive search



Figures 3.20 through 3.27 plot the best fitness values found versus the generation number. Each graph shows one of the eight tests. It is interesting to note that each graph shows the current best found fitness steadily increasing, verifying that the GA is actually optimizing the 10240 bit string at a fairly constant rate. But the process is extremely slow due to the large amount of computations. (On a Pentium 133MHz PC, each test took several hours to complete). To compute the fitness of an individual string 1280 block compares are needed. So for each generation there are  $1280 \times 500 = 640,000$  block compares. This is a much greater amount of computation than with the exhaustive search method which requires only  $1280 \times 64 = 81,920$  compares total. The number of compares done by the GA could have been decreased by reducing the size of the GA's population, but it was kept large for the initial experiments since GAs with larger populations tend to work better. The performance of the Fixed Block Size One-Shot method was worse in a number of quality measures than that of the exhaustive search method. Thus, there was no reason to reduce the population size for further experiments.

Figure 3.20 Best fitness per generation for fixed block size one-shot search, Test 1

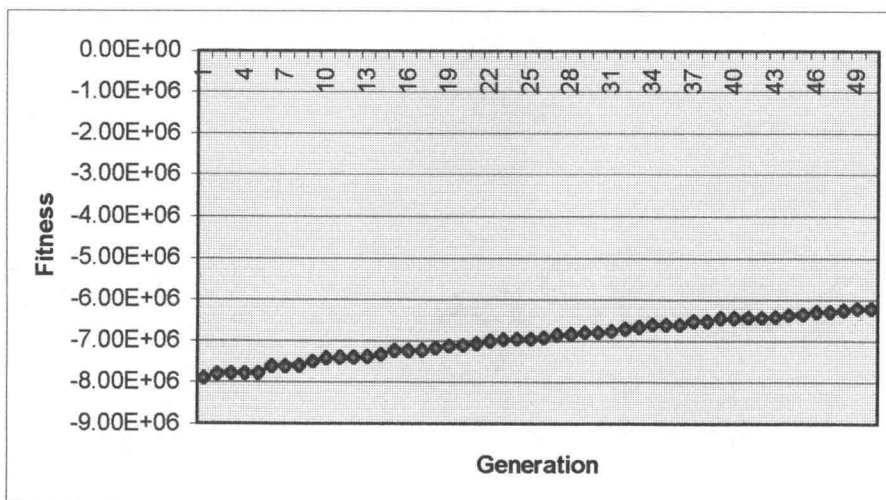


Figure 3.21 Best fitness per generation for fixed block size one-shot search, Test 2

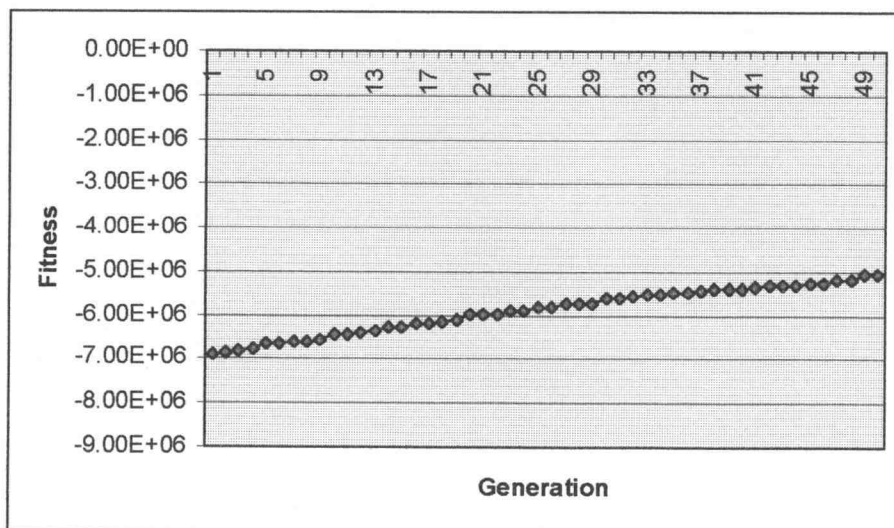


Figure 3.22 Best fitness per generation for fixed block size one-shot search, Test 3

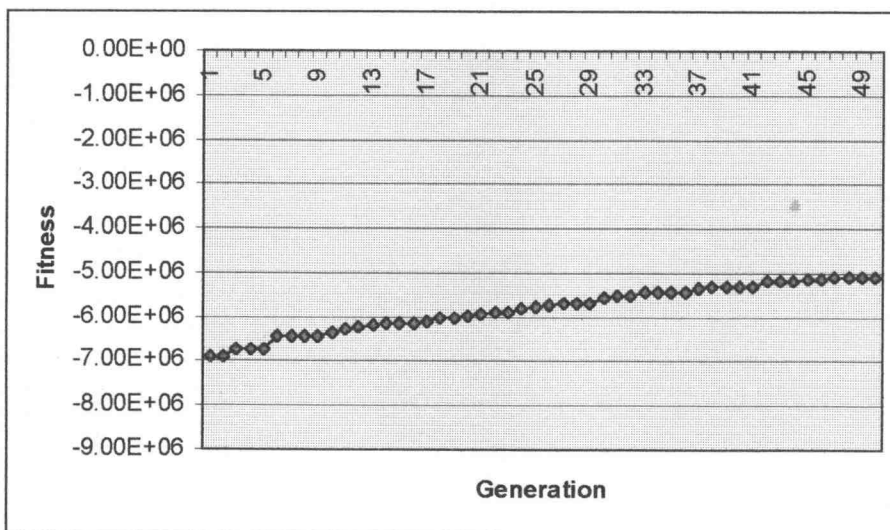


Figure 3.23 Best fitness per generation for fixed block size one-shot search, Test 4

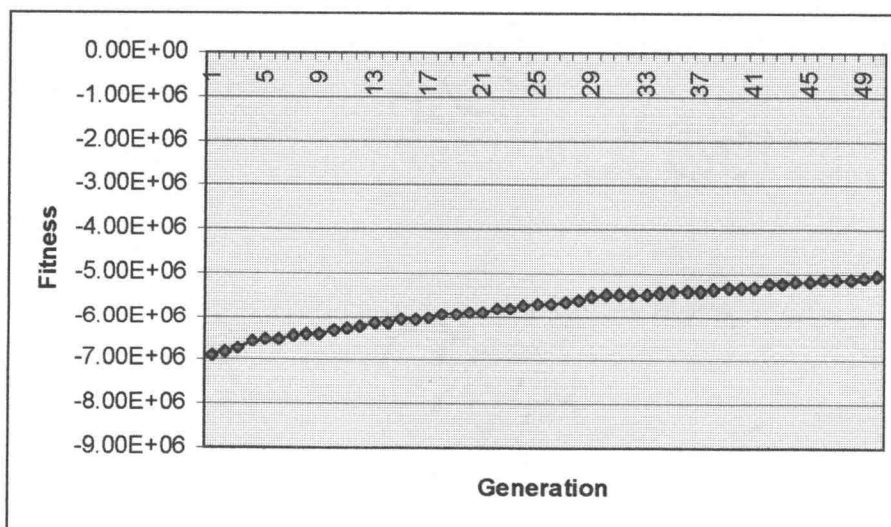


Figure 3.24 Best fitness per generation for fixed block size one-shot search, Test 5

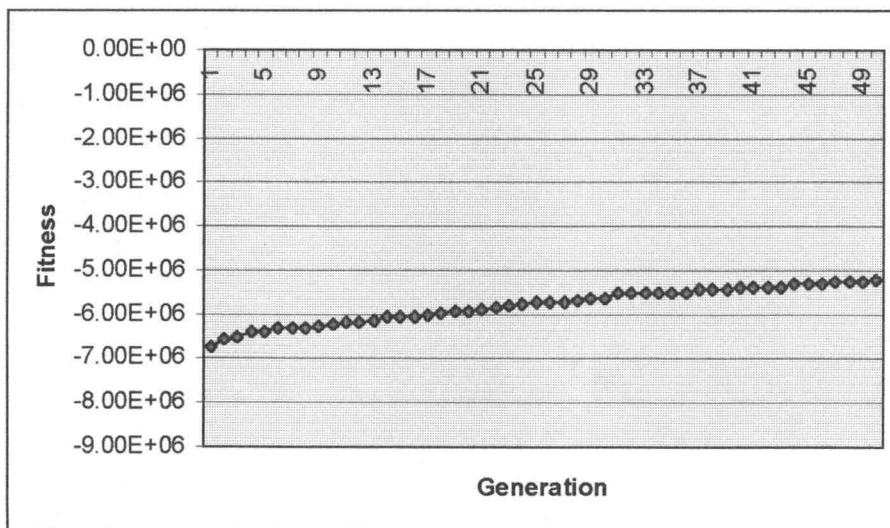


Figure 3.25 Best fitness per generation for fixed block size one-shot search, Test 6

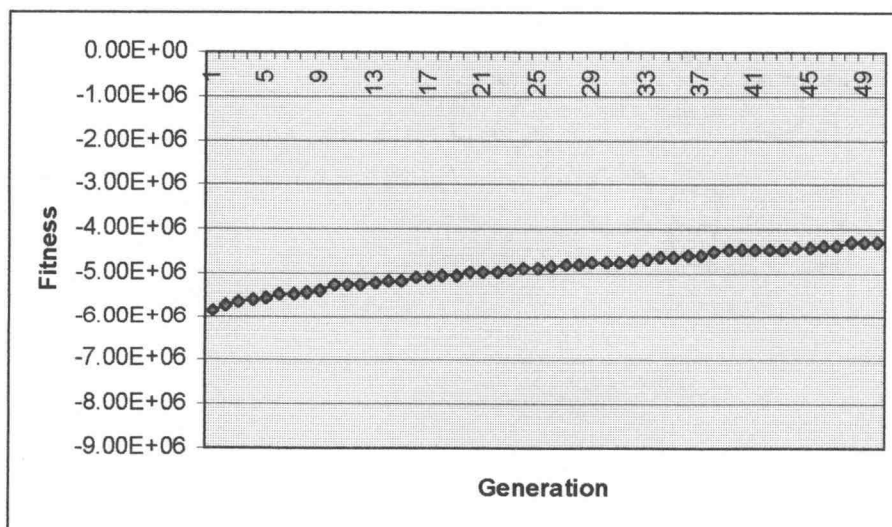




Figure 3.26 Best fitness per generation for fixed block size one-shot search, Test 7

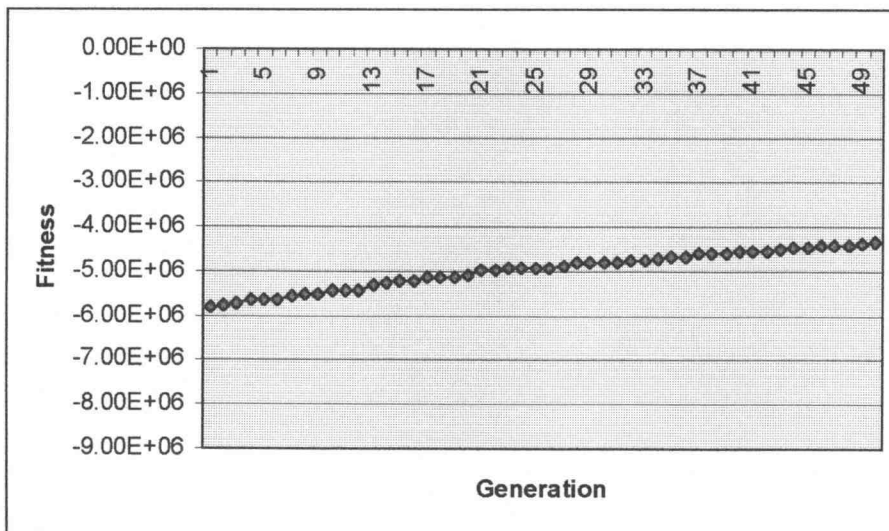
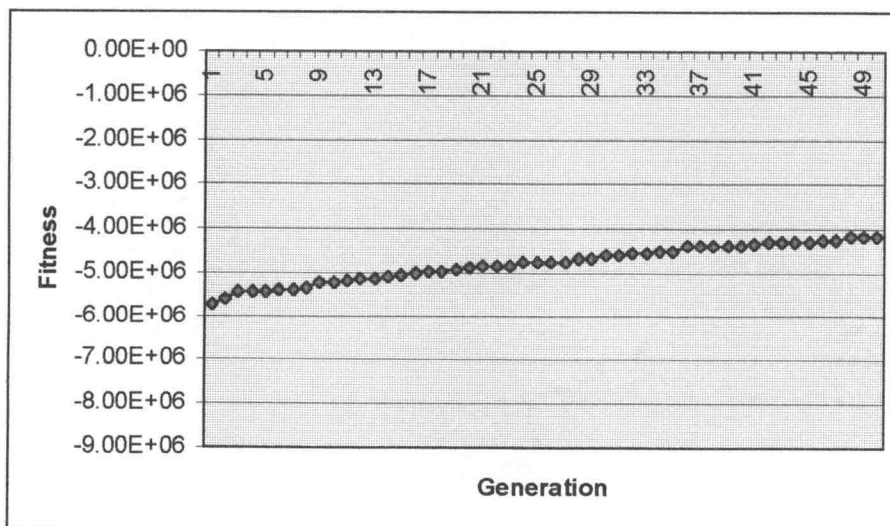


Figure 3.27 Best fitness per generation for fixed block size one-shot search, Test 8





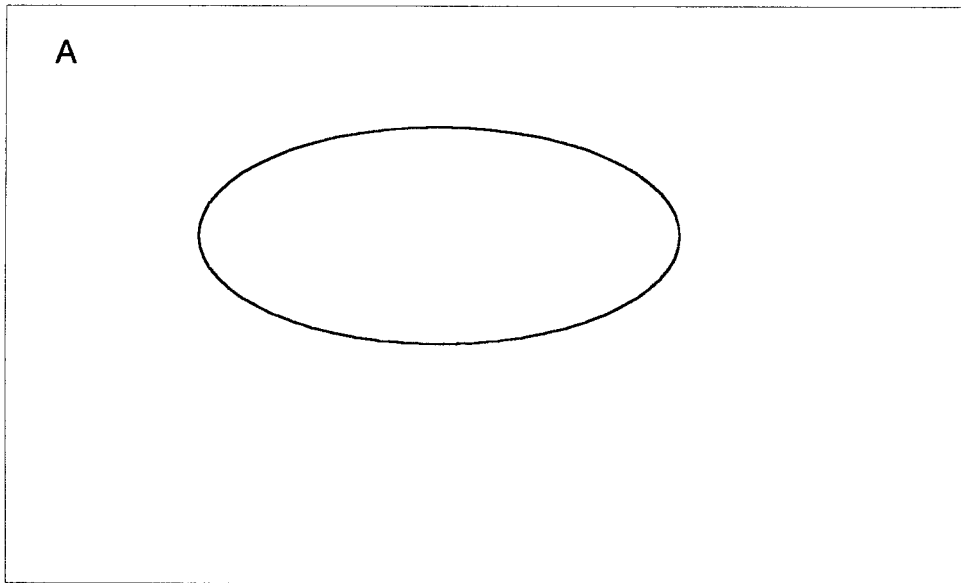
### 3.3 Block Motion Estimation with Variable Block Sizes

Block motion estimation can be generalized to include blocks of variable sizes. The MPEG standard does not allow for this, but using variable block sizes can be more efficient. Since one motion vector is required for each block, fewer motion vectors are required if larger common blocks are found between frames.

One way to represent variable sized blocks common between two frames is with a tree structure where each node represents a common block of the two frames. The root node is a block that covers the entire frame. The root node can have four children, each of these children can have four children, and so on. Each child represents one quarter of the parent block. When the tree is complete, traversing the leaf nodes will enumerate all of the motion estimation blocks and this will provide complete coverage, since all of the blocks will have been checked. The following figures illustrate this algorithm.

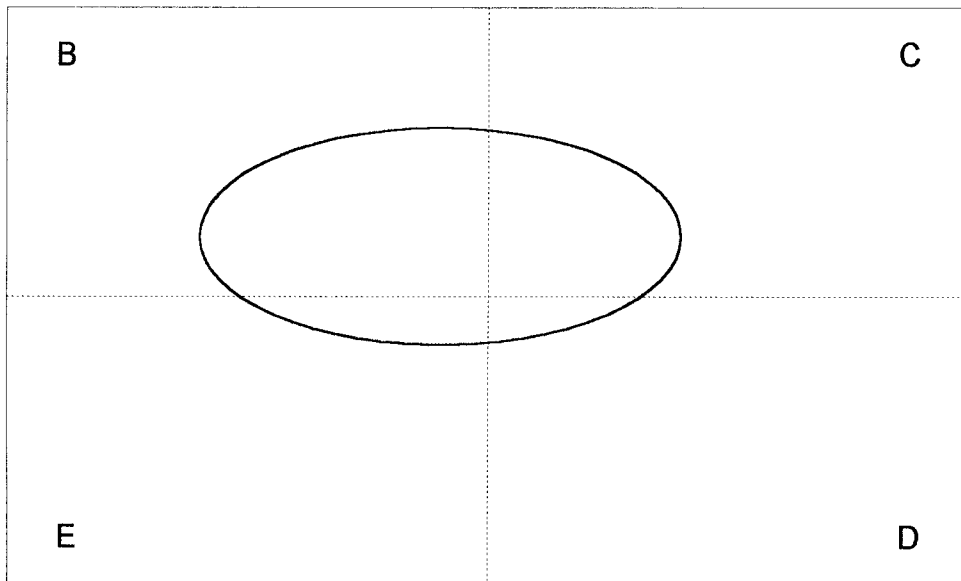
Figure 3.28 represents the root node of the current frame. The root node includes the entire frame. If the previous frame (the frame to which this one is being compared) is the same (with a possible horizontal and/or vertical displacement), then there is no need to have a larger tree. One single motion vector is sufficient to express the differences between the two frames.

Figure 3.28 Variable block size motion estimation, Step 1



If the frame to which the comparison is made is not the same, the current frame must be divided into smaller sections so that these smaller sections can be correlated between the current and the previous frame. Figure 3.29 illustrates how the current frame is divided into four equal sized blocks (four children of the root node), which are compared to same sized blocks in the previous frame. In the case that only a small section of the image has changed, most of the blocks will have a motion vector of zero and will not need to be divided any further.

Figure 3.29 Variable block size motion estimation, Step 2



In Figure 3.29, Block B was the only block for which a good match between the current and the previous frame could not be found. Therefore Block B needs to be further subdivided as is illustrated in Figure 3.30. This process can be repeated until the blocks match or until a minimum block size is reached.

Figure 3.30 Variable block size motion estimation, Step 3

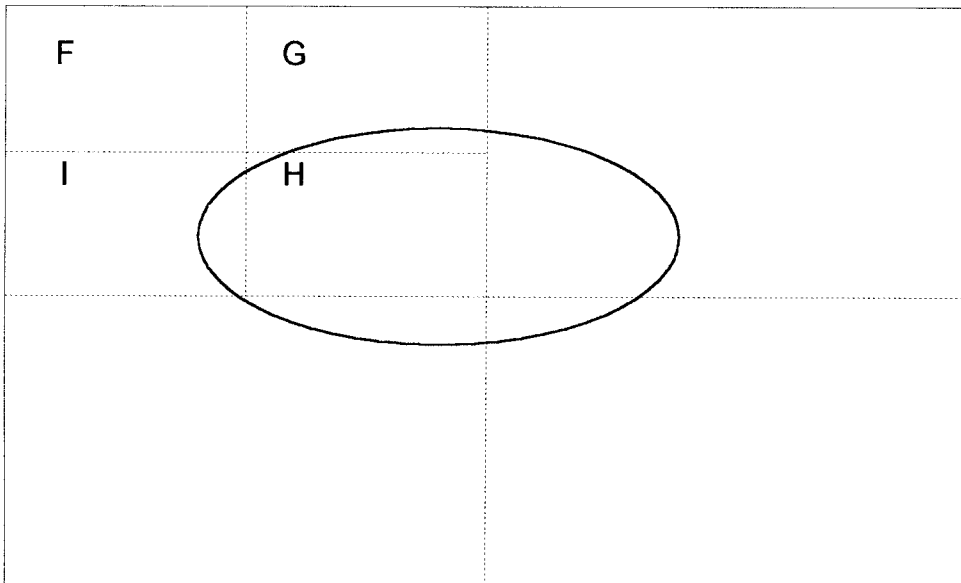
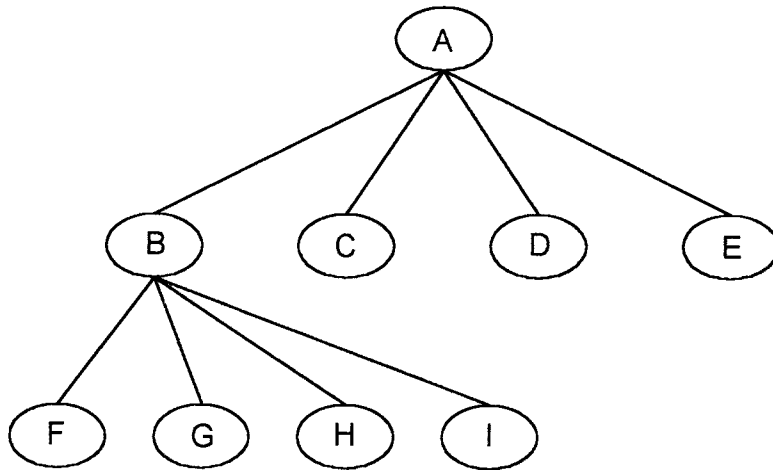


Figure 3.31 represents the variable size block motion estimation tree of Figures 3.28 through 3.30. Each node represents a motion block with its own motion vector. Traversing the leaf nodes of the tree (C, D, E, F, G, H, I) will provide all of the blocks and their associated motion vectors. These vectors can then be used to reconstruct the current frame from the previous frame.

Figure 3.31 Variable block size motion estimation, Step 4



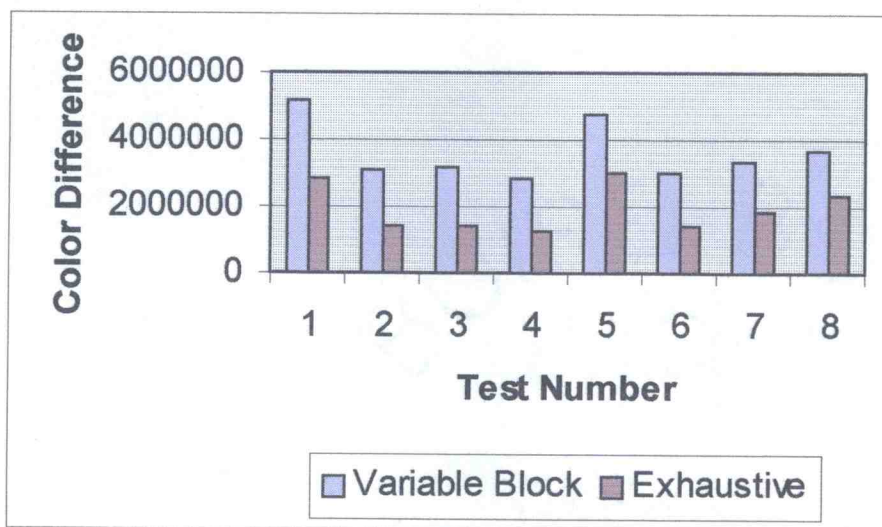
It is possible to use a GA to find the leaf nodes, but this would necessitate using a GA for each individual node; it would be much more effective if the entire tree could be processed at once. A GP is appropriate for this purpose, since it works on tree structures rather than strings. Using a GP, a population of tree structures can be created, each of which represents the entire frame. Then through artificial evolution the population should converge to a tree structure that uses the fewest nodes to represent the best motion blocks. In this manner, it is possible to build the entire tree at once.

In the following experiments, involving the Test 1 through Test 8 video sequences, the GP was run for 50 generations with a population size of 500. The minimum tree height was 2 and the maximum was 6. Every tree node was allowed four sub-nodes (or children). A completely filled out tree of height 6 has 1024 leaf nodes. Since the test image sizes are 320 pixels by 256 pixels (refer to Appendix A and

Appendix B), the smallest possible block is 10 pixels by 8 pixels. The largest possible block is the entire image.

Figure 3.32 illustrates how well the GP performed its variable block size search compared to the exhaustive search. It is interesting to note that this search performed slightly better than the Improved Fixed Block Size Iterative search (Figure 3.10); the average color difference for each test is about 6% less than that of the Improved Fixed Block Size Iterative search. However, this was the slowest of all the search methods tested.

Figure 3.32 Color difference of variable block size and exhaustive search



Figures 3.33 through 3.40 plot the current best fitness found versus the generation number. Each graph shows one of the eight tests. Figures 3.34, 3.35, 3.36, 3.38, 3.39, and 3.40 all only show an increase in the best fitness in the first few generations and then the fitness remains unchanged for subsequent generations. Figures

3.33 and 3.37 only show a very small increase in the best found fitness as the generations progress.

Figure 3.33 Best fitness per generation for variable block size search, Test 1

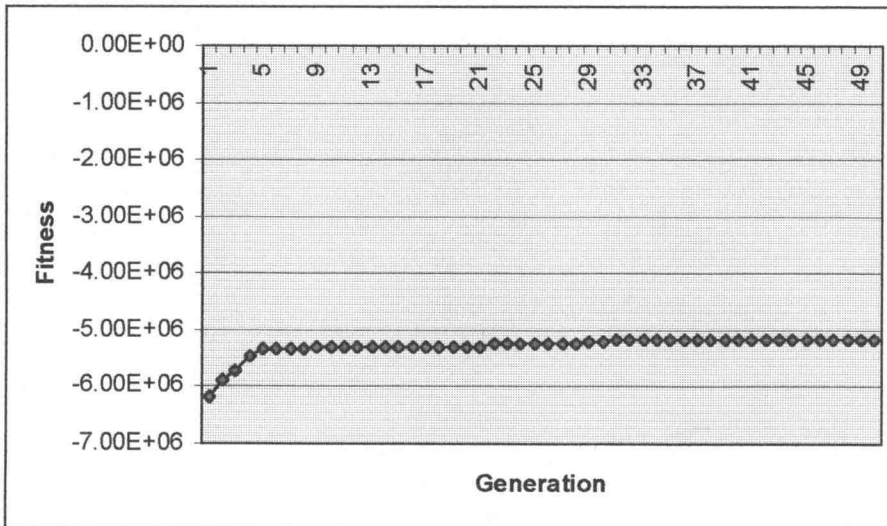


Figure 3.34 Best fitness per generation for variable block size search, Test 2

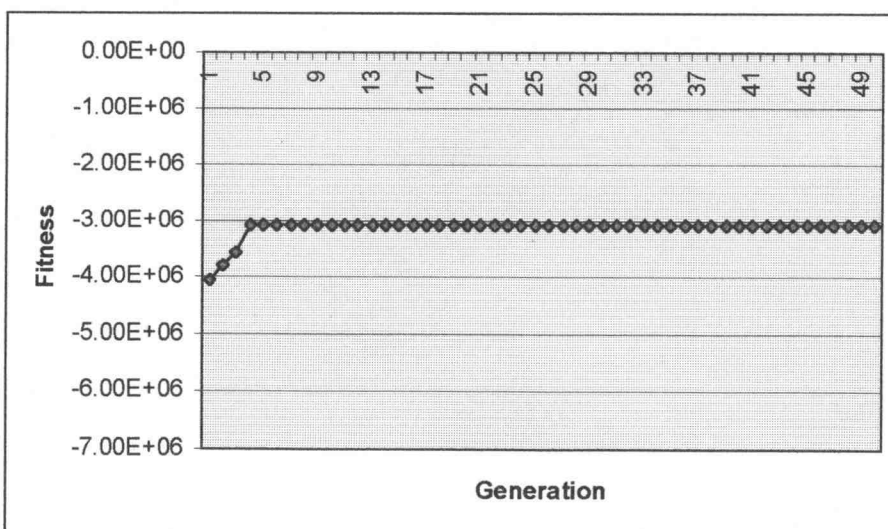


Figure 3.35 Best fitness per generation for variable block size search, Test 3

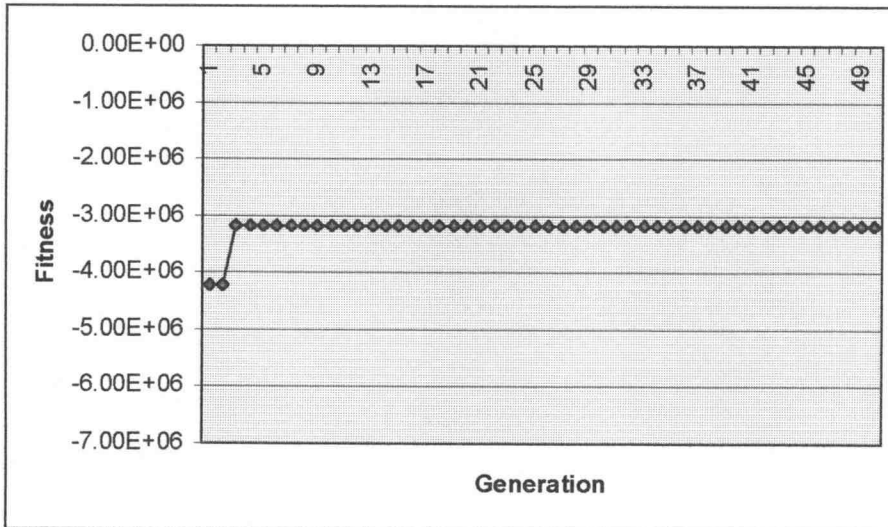


Figure 3.36 Best fitness per generation for variable block size search, Test 4

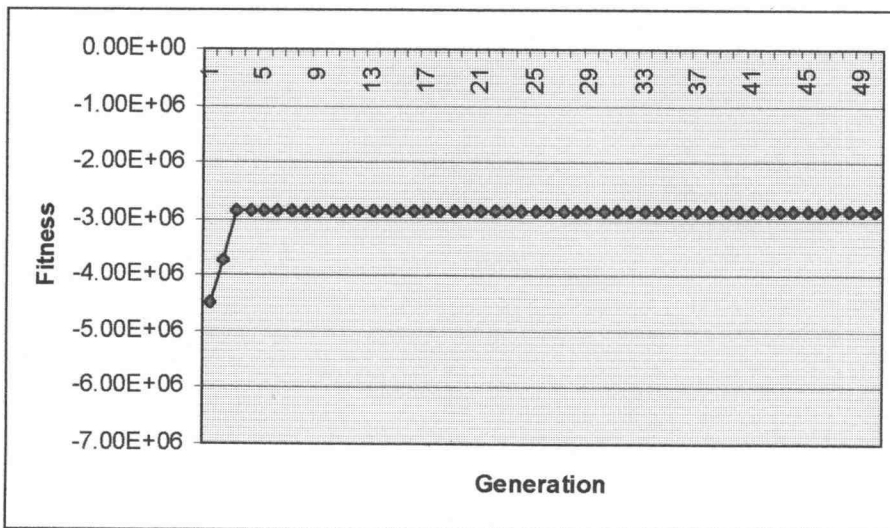




Figure 3.37 Best fitness per generation for variable block size search, Test 5

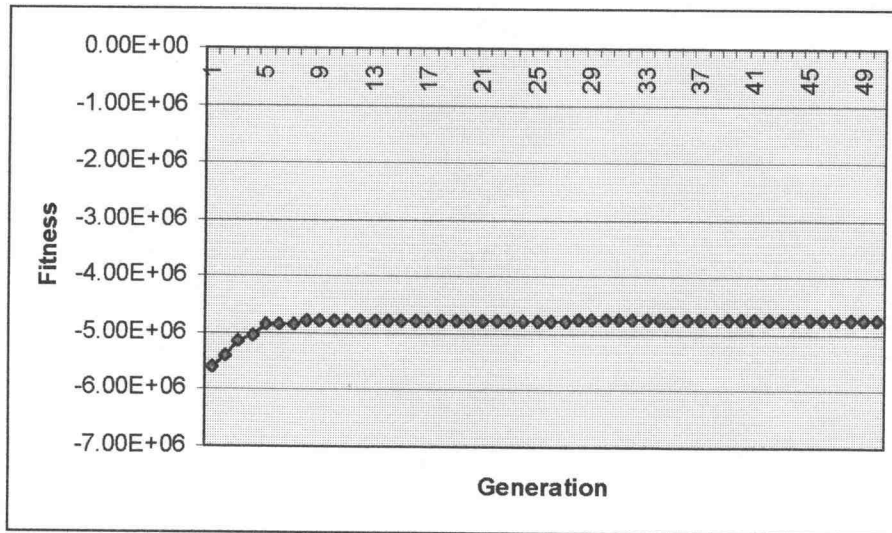


Figure 3.38 Best fitness per generation for variable block size search, Test 6

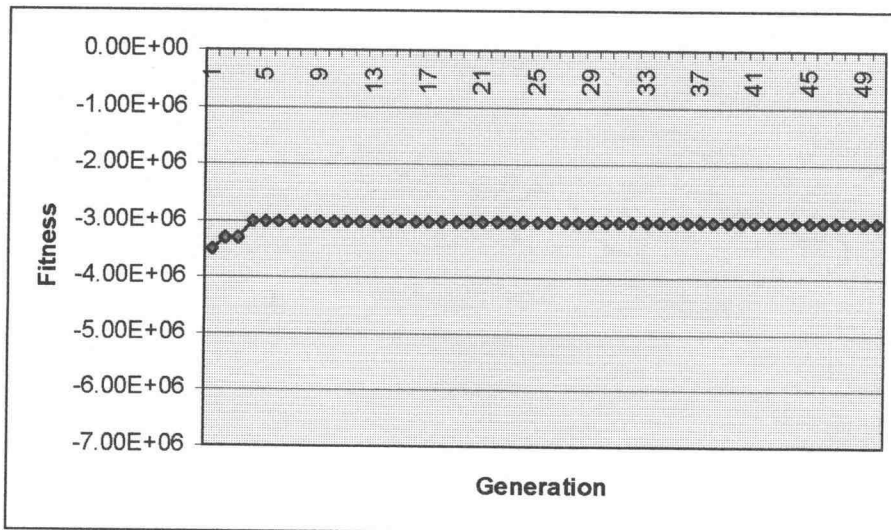


Figure 3.39 Best fitness per generation for variable block size search, Test 7

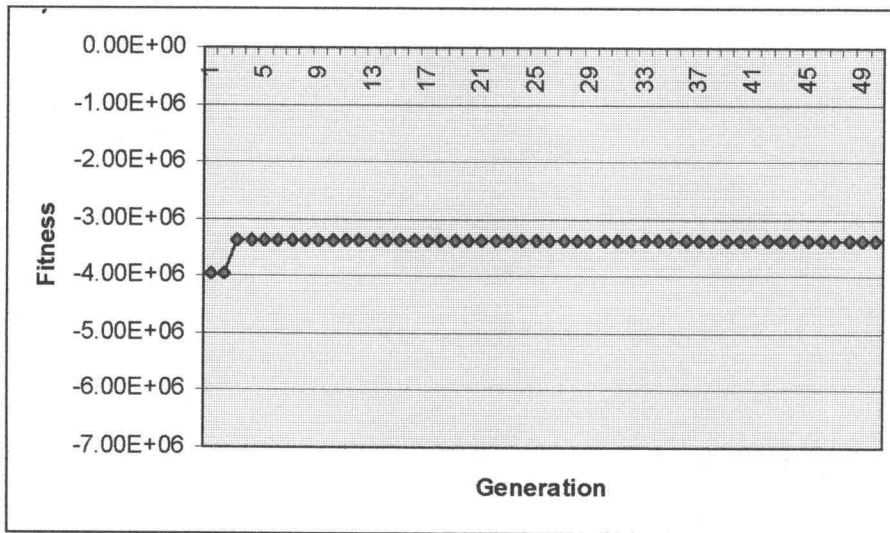
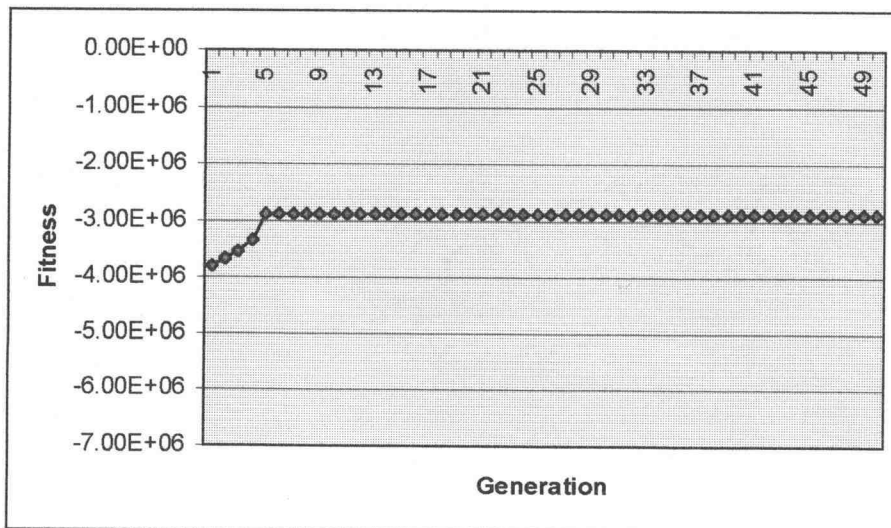


Figure 3.40 Best fitness per generation for variable block size search, Test 8



### **3.4 Region Based Motion Estimation**

The goal of region based motion estimation is to improve on variable block size motion estimation by assigning motion vectors to regions, which are not necessarily squares or rectangles. One of the main motivations to use regions over rectangles is that moving objects are rarely rectangular. If these motion regions can be adequately determined, then the number of motion vectors needed to represent a sequence of moving pictures can be reduced. Region based motion estimation also reduces the “blockiness” (a number of block shapes detectable in the final decompressed image) associated with block based motion estimation at low data rates. But, this problem is more computationally intensive than other motion estimation algorithms since motion boundaries must be determined.

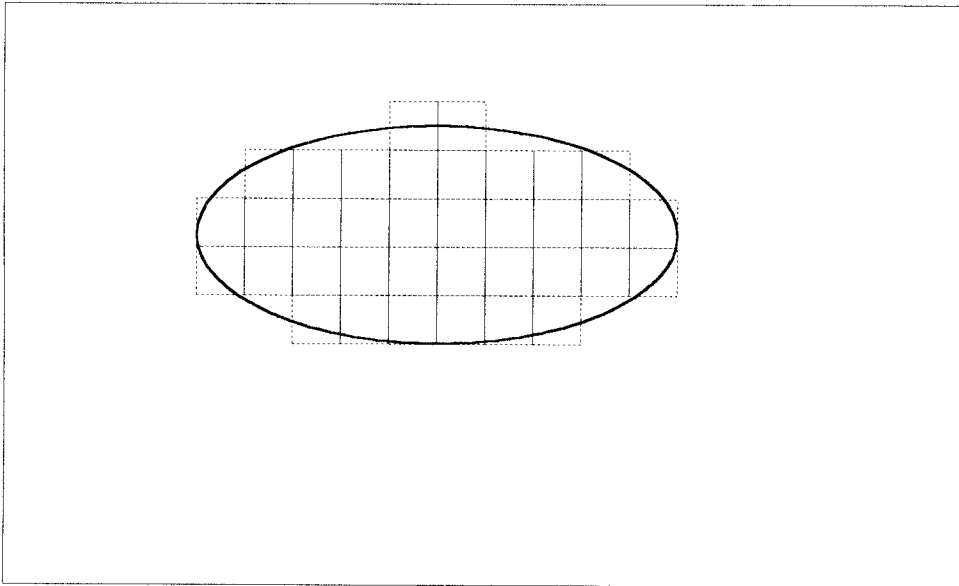
Motion regions can also be used in the process of image segmentation. In image segmentation the physical elements of an image are identified, for example an airplane in an air traffic control system or a visual inspection in a manufacturing quality control system which identifies faulty parts. Image segmentation is typically done using several different techniques combined together, since the use of a single technique (such as using motion) is very limited.

#### **3.4.1 Region Based Motion Estimation, Fixed Block Size Method**

One fairly simple implementation of region based motion estimation is to create regions by combining adjacent blocks that have the same motion vectors. This is pretty simple, but unless fairly small sized blocks are used, the motion estimated image may suffer from some blockiness around the region’s boundaries, as can be seen in Figure

3.41. This process is an improvement over fixed block size motion estimation, where the blockiness can occur throughout the image.

Figure 3.41 Region motion estimation, Fixed Block Size Method



A simple way to implement region based motion estimation with a fixed block size is to use the motion vectors generated by a GA. The blocks with the same motion vectors are combined into groups which represent motion regions. The vectors can be generated by either the iterative method (in which the motion vectors for the blocks are evaluated one at a time) or the one-shot method (in which all of the blocks are processed simultaneously). Once the GA processes all of the blocks, the blocks are sorted into groups which have the same motion vectors. Each of these groups may contain several non-adjacent regions which can be further subdivided if necessary.

Depending on the application, adjacent blocks in these groups can be combined into larger blocks, reducing the overall block count. Another option is to combine all of the blocks in a group and specify the region by its perimeter.

This experiment will start with the Improved Fixed Block Size Iterative Method, which has performed best in this research. The GA is run with the same parameters as in Section 3.2.1: a population size of 50, 20 generations, tournament selection, a crossover probability of 0.6, and a mutation probability of 0.0005. When all of the blocks have been processed, the best motion vectors for each block are stored in a bitmap file such that all blocks which have the same motion vectors are filled with the same color. This creates a visual representation of the motion vectors used to reconstruct a frame. Looking at this visual motion representation it should be possible to recognize moving objects. However, this is a subjective experiment. For the video sequences in Test 1 through Test 4 refer to Appendix A.

Figures 3.42 through 3.45 show a graphical representation of the motion vectors found by the GA. Figures 3.43, 3.44, 3.45 show the head of the woman singer fairly clearly. Figure 3.42 has quite a bit more noise in it.

Figure 3.42 Region motion estimation, Fixed Block Size Method, frame 1

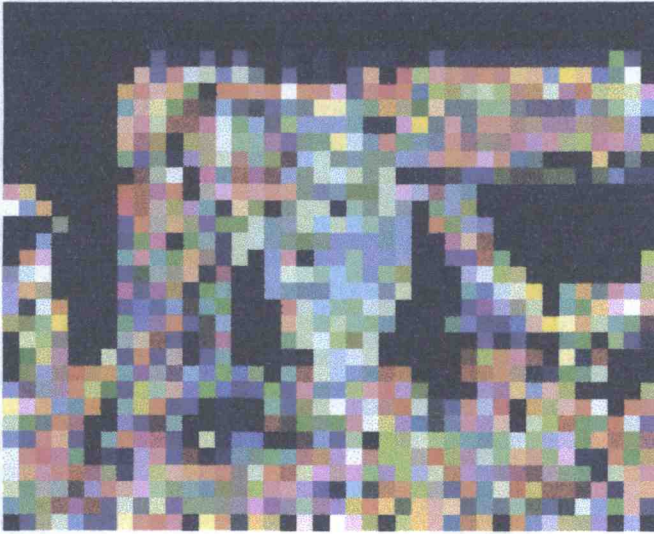


Figure 3.43 Region motion estimation, Fixed Block Size Method, frame 2

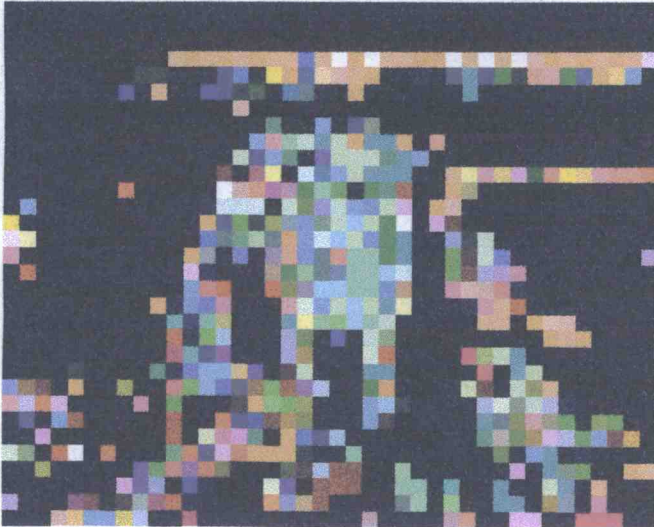


Figure 3.44 Region motion estimation, Fixed Block Size Method, frame 3

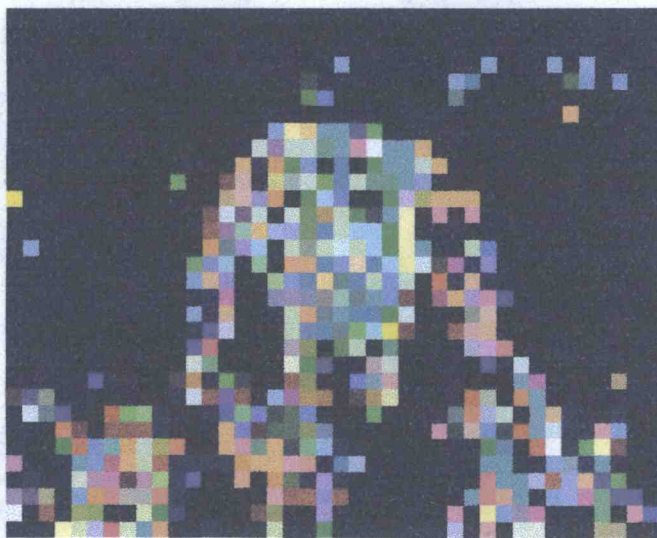
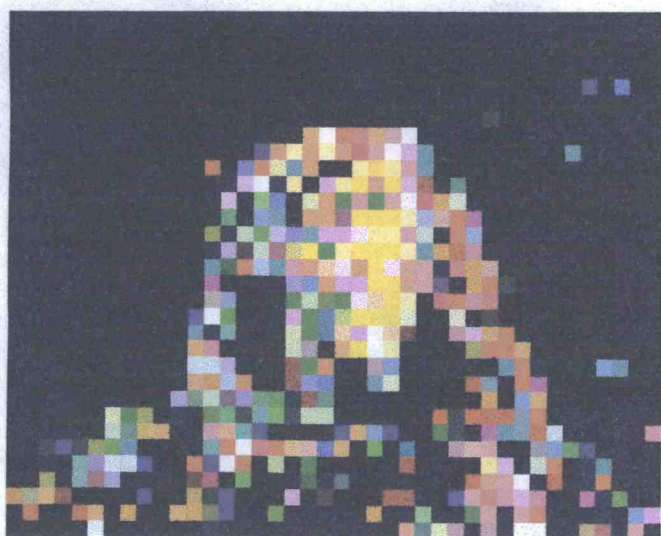


Figure 3.45 Region motion estimation, Fixed Block Size Method, frame 4



### **3.4.2 Region Based Motion Estimation, Variable Block Size Method**

The region based motion estimation in the experiments in this section makes use of variable block sizes. It is best to use the largest possible blocks to minimize the

amount of information that needs to be incorporated in the compressed data stream. These variable sized blocks can be nicely represented in a tree structure, as has been previously described. Once the tree is built, it is traversed and all of the blocks that have the same motion vector are logically grouped together. Note that the blocks do not have to be adjacent, since this is a grouping of all of the regions that have the same motion. If needed, the groups can be divided into groupings which only contain the motion regions in which the blocks are adjacent.

The region based motion estimation Variable Block Size Method differs from the Fixed Block Size Method in that the Variable Block Size Method is a top-down approach, while the Fixed Block Size Method is a bottom-up approach. The Variable Block Size Method is illustrated with Figures 3.46-3.49.

Figure 3.46 shows the current frame with a region denoted, which also exists in the previous frame, but at a different location. To determine the boundaries of this region and how they relate to a similar region in the previous frame, some analysis must be done. First the entire current frame is compared to the entire previous frame and it is determined that these frames are not the same.



Figure 3.46 Region motion estimation Variable Block Size Method, Step 1

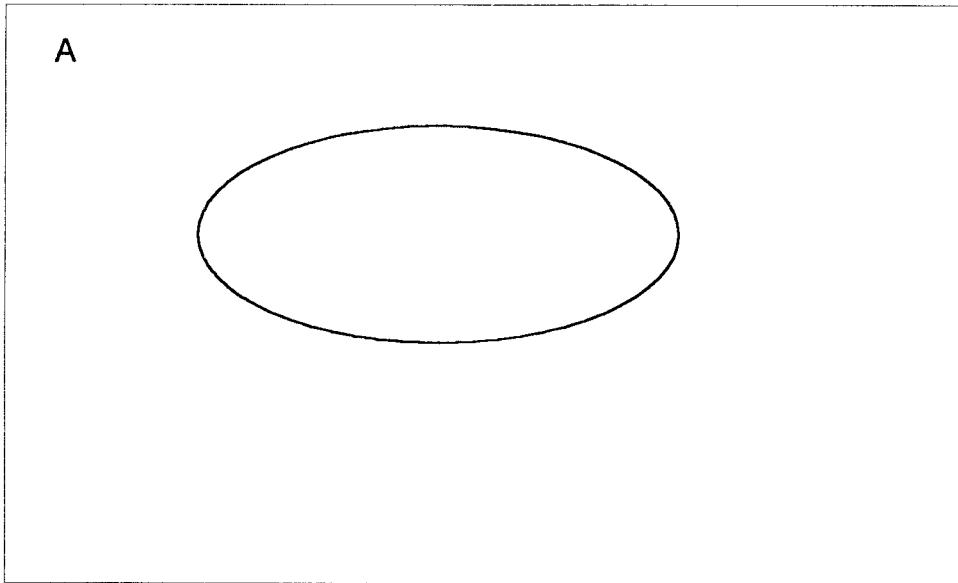


Figure 3.47 shows how the image is next divided into four equal areas in an attempt to find some matching areas between this frame (the current frame) and the previous frame. Since it is found that the areas still do not match, it is necessary for the frame to be subdivided further.

Figure 3.47 Region motion estimation Variable Block Size Method, Step 2

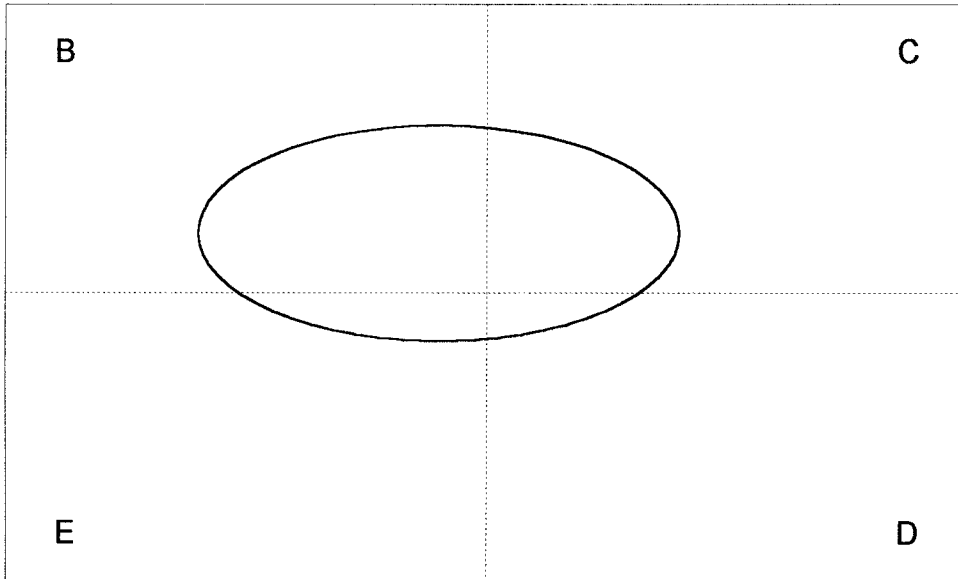
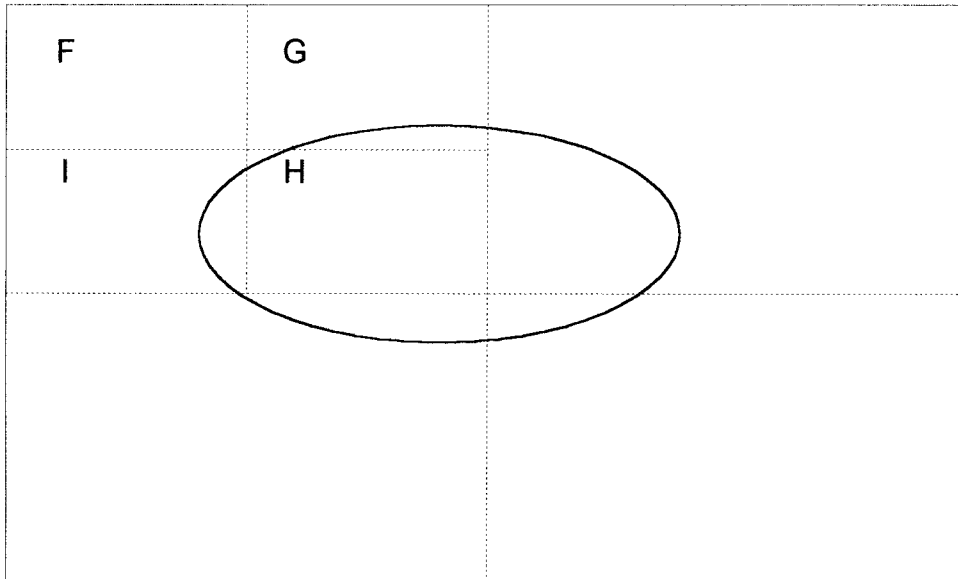


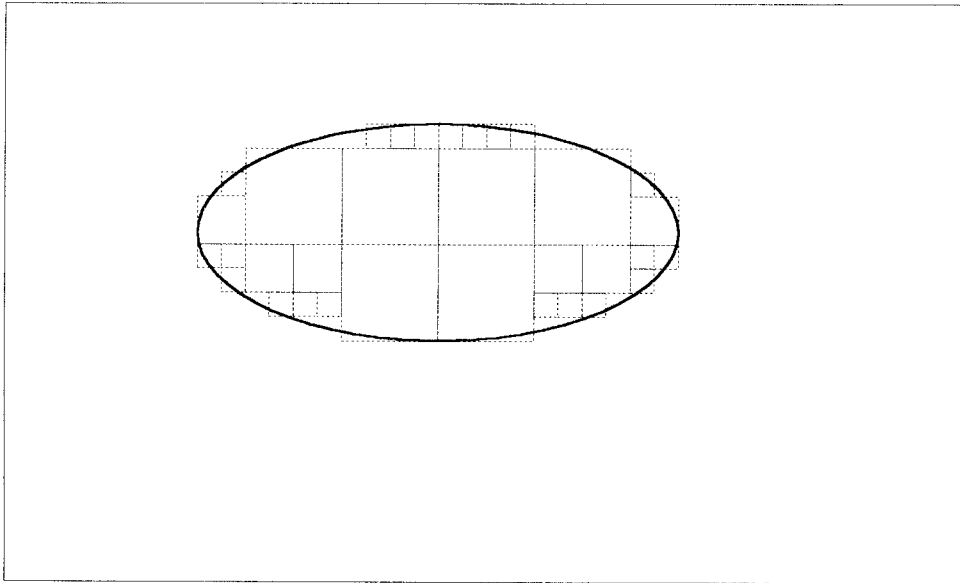
Figure 3.48 details the smaller subdivisions in the image. The process of dividing the image into smaller and smaller subdivisions continues until the motion blocks can be matched to the previous frame or until a minimum motion block size is reached.

Figure 3.48 Region motion estimation Variable Block Size Method, Step 3



After the image is divided into its motion blocks, a pass is made through the tree to logically group the blocks according to their motion vectors. The end result is a region that covers the motion area, as show in Figure 3.49. This method could also be used as a means to identify moving objects in an image sequence.

Figure 3.49 Region motion estimation Variable Block Size Method, Step 4



Note that the image subdivision produced by the region motion estimation of the Variable Block Size Method is similar to the end result of the Fixed Block Size Method if the adjacent blocks in the Fixed Block Size Method are combined into larger blocks. However, the Fixed Block Size Method has the advantage that the block combinations are done as part of the block identification process. The advantage of the Variable Block Size Method over that of the Fixed Block Size Method is that the Variable Block Size Method should be faster. (This is because if the Variable Block Size Method finds an appropriate motion vector for a larger block, it can move on to the next block, while the Fixed Block Size Method must determine the motion vectors for all of the blocks.)

For this experiment the Variable Block Size motion estimation GP code from Section 3.3 was used. Video segments shown in Test 1 through Test 4 (refer to Appendix A) were used in this experiment. The GP used a population size of 500 and ran for 50 generations. The minimum tree height was 2 and the maximum height was 6.

Figures 3.50 through 3.53 are the graphical representations of the motion regions as determined by the GP. These figures illustrate how the GP partitioned the image into sub-blocks. However the GP did not partition the images into the sub-blocks that one would expect. The blocks created by the GP are much too large to allow recognition of the image of the woman singing. The GP appeared to favor smaller trees. The most likely explanation of this behavior is that the GP can minimize the error of the motion vectors more easily if there are fewer of them.

Figure 3.50 Region motion estimation, Variable Block Size Method, frame 1

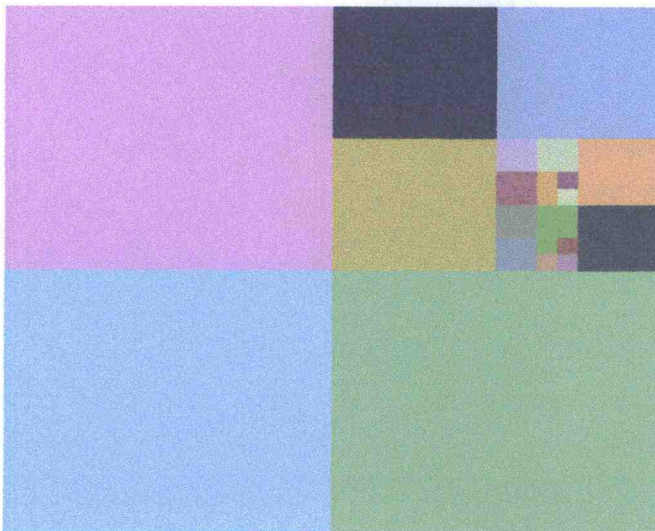


Figure 3.51 Region motion estimation, Variable Block Size Method, frame 2

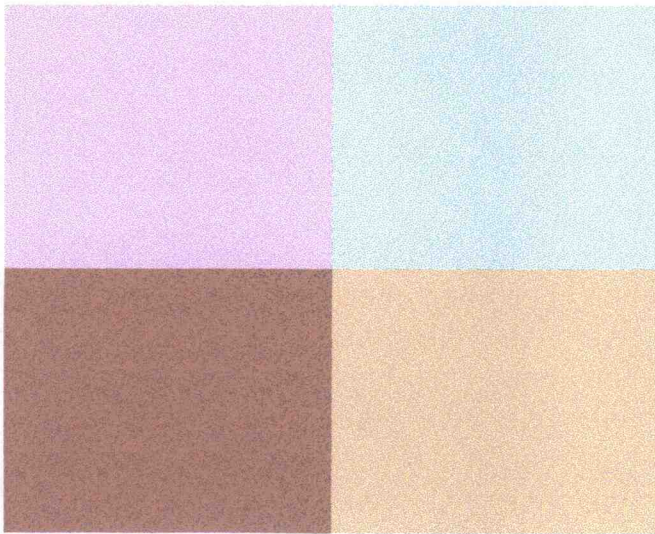


Figure 3.52 Region motion estimation, Variable Block Size Method, frame 3

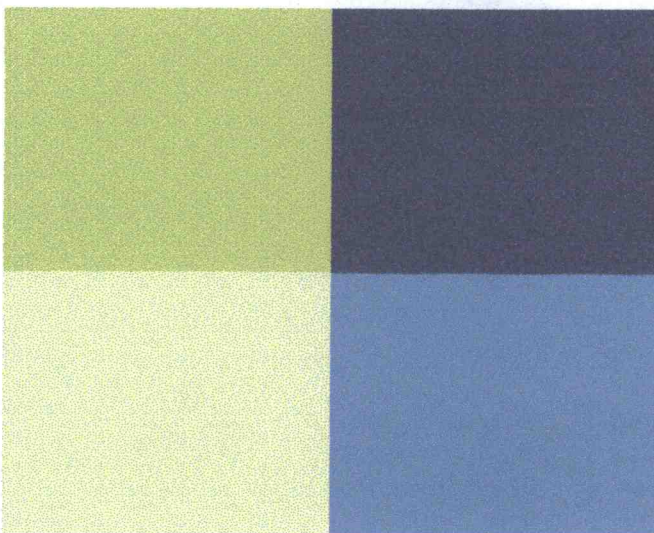
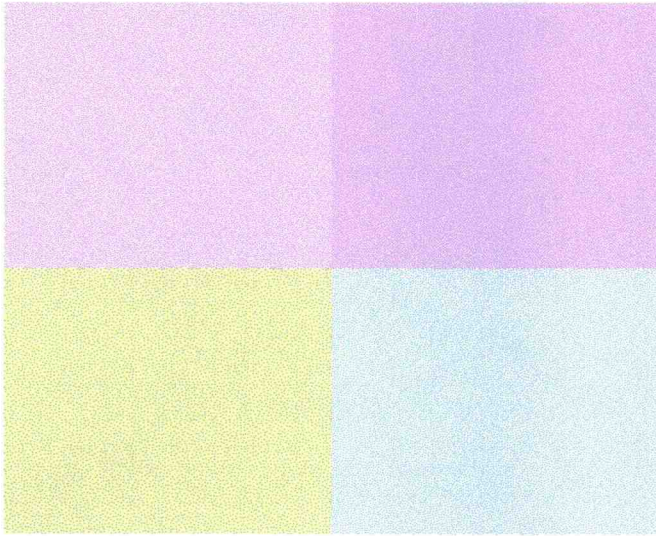


Figure 3.53 Region motion estimation, Variable Block Size Method, frame 4



## **CHAPTER 4 - CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK**

Motion estimation is one of the most important components of video compression. To date all of the previous research has concentrated on improving motion estimation by creating new algorithms which sub-sample the data in various ways to overcome the need for an exhaustive search. In this research a totally new approach to motion estimation was proposed, making use of Genetic Algorithms and Genetic Programs. This is a major deviation from the previous work done in the field because of the way the evolvable search methodologies work. With this approach it is no longer required to develop an algorithm to find optimal motion vectors, since the evolvable search methodologies automatically develop the algorithm.

Fixed block size motion estimation was first examined. The first experiments applied an iterative algorithm in which a GA was used to compute the individual motion vectors for similar sized blocks. This method was found to perform worse than the exhaustive search, both in the speed of execution and in the quality of the resulting motion vectors.

By adding the null motion vectors to the initial population of the GA the performance of the fixed block size motion estimation was improved considerably. The average search time was better than that of the exhaustive search and the quality of the motion vectors was greatly improved from the previous experiments.

Instead of performing an iterative search for fixed block size motion vectors, experiments were conducted using a GA to compute all of the motion vectors to correlate two video frames simultaneously. It was found that although the GA did improve the fitness values of the motion vectors over time, the progression was



extremely slow. The number of blocks of data compared is also considerably larger than the total number of blocks compared with an exhaustive search.

The possibility of using a GP to create variable block size motion vectors was next investigated. (It is advantageous to have the motion blocks be as large as possible, since fewer motion vectors are then required.) A GP was more appropriate for this problem than a GA since the tree structure lends itself to a recursive representation of sub-blocks. Through experimentation it was found that the error between the reference frame and the frame generated by the motion vectors was slightly better than the fixed block size motion estimation with the null vectors added to the initial population. Unfortunately, the speed of execution for this application was the worst of all the experiments.

GAs and GPs were also utilized to determine motion regions. The central goal of this method was to group like motion vectors to determine the shape of a moving object. In three out of four experiments the GA produced motion estimation image results which resembled the head of the woman singer (Appendix A), while the GP failed to accomplish this. This was attributed to the “greediness” of the GP, as it displayed a tendency to use the largest possible blocks and thus not create motion blocks small enough to represent the details of the moving images.

After experimenting with the various forms of motion estimation and evolvable search methods, it can be concluded that although the approach of evolutionary motion estimation is both novel and interesting, it is not practical. There are currently other motion estimation algorithms which generate motion vectors faster and of comparable quality to those found by an exhaustive search.

There are however, a few areas of evolutionary motion estimation which hold promise and should be explored further. One such area is to use a GA to generate an algorithm or pattern, which would then be used to find motion vectors to correlate motion blocks between two frames of a video sequence. Depending on the video sequence, the algorithm or pattern may not produce useful results for the entire video sequence. When it is found that the quality of the vectors produced declines, the GA will have to be re-run to generate a new algorithm or pattern; this could be done automatically if the block matching rate falls below a certain level. This would essentially be a continuation of the work done by Cavicchio (1970) in which a GA was utilized in a pattern recognition problem. In this research a GA generated “detectors” or patterns, which were used to classify a digitized image; the GA was not directly used to process images.

The use of more complex motion vectors also merits further investigation. Rather than just using horizontal and vertical displacement, rotation could be incorporated, or a polynomial could be used to express the motion vector, as in the research by Karczewicz et al. (1995). This should be of greater benefit as the size of motion blocks increase, especially in the case of using variable block sizes.

Additionally, the region based motion estimation methods could be combined with other image segmentation techniques to aid in object identifications. (This is because motion estimation alone only works if the entire object is moving with relation to its background.)

As digital video becomes more prevalent, video compression becomes more important. New algorithms, implemented in both hardware and software, are being created at a rapid pace. New approaches are necessary to increase the speed of compression and playback, as well as the quality of the playback video. Better compression algorithms make it possible to store ever larger quantities of video images on current computer storage devices. This research explored artificial evolutionary algorithms as a new and innovative means of improving motion estimation for video compression.

## BIBLIOGRAPHY

- Barron, J. L. and R. Eagleson. "Binocular Estimation of Motion and Structure from Long Sequences Using Optical Flow Without Correspondence". In Proceedings: International Conference on Image Processing, Washington, D.C., October 23-26, 1995. Volume II. Los Alamitos, California: IEEE Computer Society Press, 1995. pp. 193-196.
- Bonomi, Mauro. "Multimedia and CD-ROM: An Overview of JPEG, MPEG and the Future". CD-ROM Professional. November 1991, pp. 38-40.
- Cavicchio, D. J. "Adaptive search using simulated evolution." Diss. University of Michigan, Ann Arbor 1970.
- Chan, Yui-Lam and Wan-Chi Siu. "A New Block Motion Vector Estimation Using Adaptive Pixel Decimation". In Conference Proceedings: The 1995 International Conference on Acoustics, Speech, and Signal Processing, May 9-12, 1995, Westin Hotel, Detroit, Michigan, USA. Volume 4. Sponsored by The Signal Processing Society of The Institute of Electrical and Electronics Engineers. Piscataway, New Jersey: The Institute of Electrical and Electronics Engineers, 1995. pp. 2257-2264.
- Chang, Hsuan T. and Chung J. Kuo. "An Improved Scheme for Fractal Image Coding". In Proceedings: 1995 IEEE International Symposium on Circuits and Systems, Seattle, Washington, USA, April 30 - May 3, 1995. Volume 3. Sponsored by the IEEE Circuits and Systems Society. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, 1995. pp. 1624-1627.
- Chellappa, Rama. Digital Image Processing. Los Alamitos, California: IEEE Computer Society Press. 1992.
- Chiariglione, Leonardo. "The Development of an Integrated Audiovisual Coding Standard: MPEG". Proceedings of the IEEE. Vol. 83, No. 2 (February 1995), pp. 151-157.

- Dang, Viet-Nam, Abdol-Reza Mansouri, and Janusz Konrad. "Motion Estimation for Region-Based Video Coding". In Proceedings: International Conference on Image Processing, Washington, D.C., October 23-26, 1995. Volume II. Los Alamitos, California: IEEE Computer Society Press, 1995. pp. 189-192.
- Delopoulos, Athanasios N. and Antony G. Constantinides. "Object Oriented Motion and Deformation Estimation Using Composite Segmentation". In Proceedings: International Conference on Image Processing, Washington, D.C., October 23-26, 1995. Volume II. Los Alamitos, California: IEEE Computer Society Press, 1995. pp. 217-220.
- Denney, Thomas S. Jr. "On Estimating 3-D Incompressible Motion". In Proceedings: International Conference on Image Processing, Washington, D. C., October 23-26, 1995. Volume III. Los Alamitos, California: IEEE computer Society Press, 1995, pp. 492-495.
- Duc, Benoit, Philippe Schroeter, and Josef Bigun. "Motion Estimation and Segmentation by Fuzzy Clustering". In Proceedings: International Conference on Image Processing, Washington, D. C., October 23-26, 1995. Volume III. Los Alamitos, California: IEEE computer Society Press, 1995, pp. 472-475.
- Dufaux, Frederic and Fabrice Moscheni. "Motion Estimation Techniques for Digital TV: A Review and a New Contribution". Proceedings of the IEEE. Volume 83, No. 6 (June 1995), pp. 858-876.
- Efstratiadis, S. N., M. G. Strintzis, and A. K. Katsaggelos. "Motion Field Prediction and Restoration for Low Bit-Rate Video Coding". In Proceedings: International Conference on Image Processing, Washington, D.C., October 23-26, 1995. Volume II. Los Alamitos, California: IEEE Computer Society Press, 1995. pp. 213-216.
- Feng, Jian, Kwok-Tung Lo, Hassan Mehrpour, and A. E. Karbowiak. "Adaptive Block Matching Motion Estimation Algorithm Using Bit-Plane Matching". In Proceedings: International Conference on Image Processing, Washington, D. C., October 23-26, 1995. Volume III. Los Alamitos, California: IEEE computer Society Press, 1995, pp. 496-499.

Fok, Yiu-Hung, Oscar C. Au, and Ross D. Murch. "Novel Fast Block Motion Estimation in Feature Subspace". In Proceedings: International Conference on Image Processing, Washington, D.C., October 23-26, 1995. Volume II. Los Alamitos, California: IEEE Computer Society Press, 1995. pp. 209-212.

Goldberg, David E. Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc. 1989.

Gonzalez, Rafael C. and Richard E. Woods. Digital Image Processing. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc. 1992.

Green, William B. Digital Image Processing: A Systems Approach. New York, New York: Van Nostrand Reinhold, Inc. 1989.

Gritz, L. and J. K. Hahn. "Genetic Programming for Articulated Figure Motion". In the Journal of Visualization and Computer Animation. 1995.

Hebert, Thomas J. and Xudong Yang. "A Sequential Algorithm for Motion Estimation from Point Correspondences with Intermittent Occlusions". In Proceedings: International Conference on Image Processing, Washington, D.C., October 23-26, 1995. Volume II. Los Alamitos, California: IEEE Computer Society Press, 1995. pp. 221-224.

Information Technology - Generic Coding of Moving Pictures and Associated Audio Information - Part 2: Video. Draft International Standard. Geneva, Switzerland: International Organization for Standardization and International Electrotechnical Commission, 1994.

Information Technology - Generic Coding of Moving Pictures and Associated Audio Information - Part 3: Audio. International Standard. Geneva, Switzerland: International Organization for Standardization and International Electrotechnical Commission, 1995.

Jehng, Yeu-Shen, Liang-Gee Chen, and Tzi-Dar Chiueh. "A Motion Estimator for Low Bit-Rate Video CODEC". IEEE Transactions on Consumer Electronics. Volume 38, Number 2, May 1992. pp. 60-69.

- Karczewicz, Marta, Jacek Nieweglowski, and Petri Haavisto. "Motion Estimation and Representation for Arbitrarily Shaped Image Regions". In Proceedings: International Conference on Image Processing, Washington, D.C., October 23-26, 1995. Volume II. Los Alamitos, California: IEEE Computer Society Press, 1995. pp. 197-200.
- Kim, Yanghoon, Chong S. Rim, and Byoungki Min. "A Block Matching Algorithm with 16:1 Subsampling and Its Hardware Design". In Proceedings: 1995 IEEE International Symposium on Circuits and Systems, Seattle, Washington, USA, April 30 - May 3, 1995. Volume 1. Sponsored by the IEEE Circuits and Systems Society. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, 1995. pp. 613-616.
- Koza, John R. Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, Massachusetts: The MIT Press, 1992.
- Koza, John R. Genetic Programming II: Automatic Discovery of Reusable Programs. Cambridge, Massachusetts: The MIT Press, 1994.
- Lam, Simon S., Simon Chow, and David K. Y. Yau. "An Algorithm for Lossless Smoothing of MPEG Video". SIGCOMM 94. London, England: Association for Computing Machinery, 1994.
- Laplante, Philip A. and Alexander D. Stoyenko, Editors. Real-Time Imaging: Theory, Techniques, and Applications. Piscataway, New Jersey: IEEE Press, 1996.
- Le Gall, Dideir. "MPEG: A Video Compression Standard for Multimedia Applications". Communications of the ACM. Vol. 34, No. 4 (April 1991), pp. 47-58.
- Lee, Jungwoo. "Optimal Quadtree for Variable Block Size Motion Estimation". In Proceedings: International Conference on Image Processing, Washington, D. C., October 23-26, 1995. Volume III. Los Alamitos, California: IEEE computer Society Press, 1995, pp. 480-483.

- Leek, Matthew R. "MPEG Q&A". CD-ROM Professional. July/August 1994, pp. 41-46.
- Li, J. and X. Lin. "Sequential Image Coding Based On Multiresolution Tree Architecture". Electronic Letters. Vol. 29, No. 17 (August 1993), pp. 1545-1547.
- Liu, Bede and Andre Zaccarin. "New Fast Algorithms for the Estimation of Block Motion Vectors". IEEE Transactions on Circuits and Systems for Video Technology. Vol. 3, No. 2 (April 1993), pp. 148-157.
- Liu, Hain-Ching and Greg Zick. "Automatic Determination of Scene Changes in MPEG Compressed Video". In Proceedings: 1995 IEEE International Symposium on Circuits and Systems, Seattle, Washington, USA, April 30 - May 3, 1995. Volume 1. Sponsored by the IEEE Circuits and Systems Society. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, 1995. pp. 764-767.
- Liu, Hongche, Tsai-Hong Hong, Martin Herman, and Rama Chellappa. "Spatio-Temporal Filters for Transparent Motion Segmentation". In Proceedings: International Conference on Image Processing, Washington, D. C., October 23-26, 1995. Volume III. Los Alamitos, California: IEEE computer Society Press, 1995, pp. 464-467.
- Martin, Graham R., Roger A. Packwood, and Injong Rhee "Variable size block matching motion estimation with minimal error". In Proceedings: Digital Video Compression: Algorithms and Technologies 1996. Volume 2668. San Jose, California: SPIE, 1996, pp. 324-333.
- Nelson, Lee J. "MPEG-1, MPEG-2 & You: Light in the Middle of the Tunnel". Advanced Imaging. November 1994, pp. 28-31, 86.
- Niyogi, Sourabh A. "Spatiotemporal Junction Analysis for Motion Boundary Detection". In Proceedings: International Conference on Image Processing, Washington, D. C., October 23-26, 1995. Volume III. Los Alamitos, California: IEEE computer Society Press, 1995, pp. 468-471.



- Ozer, Jan. "The CD-ROM Publisher's MPEG Primer". CD-ROM Professional. June 1995, pp. 79-94.
- Pan, Davis. "A Tutorial on MPEG/Audio Compression". IEEE Multimedia Magazine. Summer 1995, pp. 60-74.
- Salembier, Philippe, Luis Torres, Fernand Meyer, and Chuang Gu. "Region-Based Video Coding Using Mathematical Morphology". Proceedings of the IEEE. Volume 83, No. 6, (June 1995), pp. 843-857.
- Senda, Yuzo, Hidenobu Harasaki, and Mitsuhiro Yano. "A Simplified Motion Estimation Using an Approximation for the MPEG-2 Real-Time Encoder". In Conference Proceedings: The 1995 International Conference on Acoustics, Speech, and Signal Processing, May 9-12, 1995, Westin Hotel, Detroit, Michigan, USA. Volume 4. Sponsored by The Signal Processing Society of The Institute of Electrical and Electronics Engineers. Piscataway, New Jersey: The Institute of Electrical and Electronics Engineers, 1995. pp. 2273-2276.
- Sun, Huifang. "Hierarchical Decoder for MPEG Compressed Video Data". IEEE Transactions on Consumer Electronics. Vol. 39, No. 3 (August 1993), pp. 559-564.
- Turton, B. C. H. and T. Arslan. "An Architecture for Enhancing Image Processing via Parallel Genetic Algorithms & Data Compression". First IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems Innovations and Applications, University of Sheffield, UK, September 12-14, 1995. pp. 337-342.
- Vasconcelos, Nuno and Andrew Lippman. "Spatiotemporal Model-Based Optic Flow Estimation". In Proceedings: International Conference on Image Processing, Washington, D.C., October 23-26, 1995. Volume II. Los Alamitos, California: IEEE Computer Society Press, 1995. pp. 201-204.
- Wang, Yao, Xia-Ming Hsieh, and Jian-Hong Hu. "Region Segmentation Based on Active Mesh Representation of Motion: Comparison of Parallel and Sequential Approaches". In Proceedings: International Conference on Image Processing, Washington, D.C., October 23-26, 1995. Volume II. Los Alamitos, California: IEEE Computer Society Press, 1995. pp. 185-188.

Watkinson, John. Compression in Video & Audio. Oxford, Great Britain: Focal Press, An imprint of Butterworth-Heinemann Ltd., 1995.

Wong, Yiwan. "An Efficient Heuristic-Based Motion Estimation Algorithm". In Proceedings: International Conference on Image Processing, Washington, D.C., October 23-26, 1995. Volume II. Los Alamitos, California: IEEE Computer Society Press, 1995. pp. 205-208.

Zhang, Kui, Moroslaw Bober, and Josef Kittler. "Motion Based Image Segmentation for Video Coding". In Proceedings: International Conference on Image Processing, Washington, D. C., October 23-26, 1995. Volume III. Los Alamitos, California: IEEE computer Society Press, 1995, pp. 476-479.

**APPENDICES**

## APPENDIX A

The following five frames are the video frames used in Test 1 through Test 4. The video frames are from the Goodtime.avi video distributed with the Microsoft Windows 95 CD-ROM. The frames were individually captured, resized to 320 pixels by 256 pixels with 256 colors and saved as device independent bitmap files.

Each test consists of taking two consecutive frames and finding the motion vectors which relate individual blocks in the second frame to blocks in the first frame. The following table illustrates which frames are used in Test 1 through Test 4.

<b>Test Name</b>	<b>Previous Frame</b>	<b>Current Frame</b>
Test 1	Goodtime.avi, frame 1789	Goodtime.avi, frame 1790
Test 2	Goodtime.avi, frame 1790	Goodtime.avi, frame 1791
Test 3	Goodtime.avi, frame 1791	Goodtime.avi, frame 1792
Test 4	Goodtime.avi, frame 1792	Goodtime.avi, frame 1793

Goodtime.avi, frame 1789



Goodtime.avi, frame 1790



Goodtime.avi, frame 1791



Goodtime.avi, frame 1792



Goodtime.avi, frame 1793



## APPENDIX B

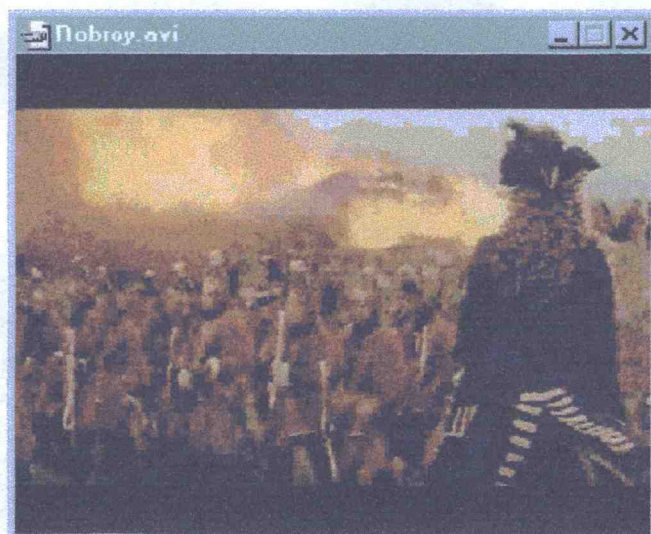
The following five frames are the video frames used in Test 5 through Test 8. The video frames are from the Robroy.avi video distributed with the Microsoft Windows 95 CD-ROM. The frames were individually captured, resized to 320 pixels by 256 pixels with 256 colors and saved as device independent bitmap files.

Each test consists of taking two consecutive frames and finding the motion vectors which relate individual blocks in the current frame to blocks in the previous frame. The following table illustrates which frames are used in Test 5 through Test 8.

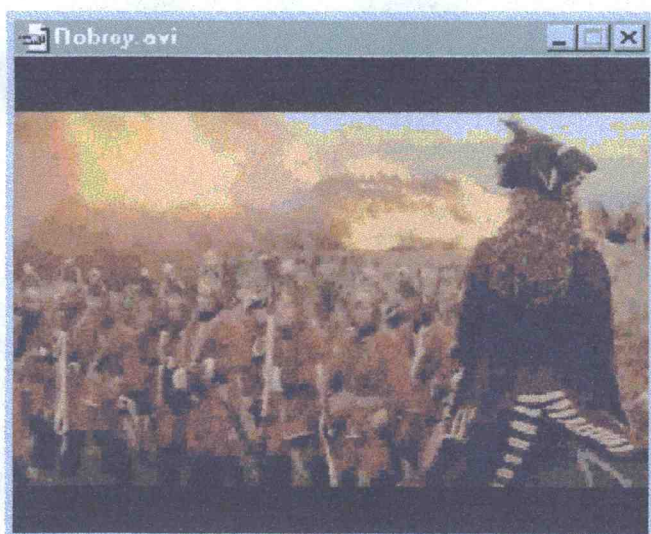
<b>Test Name</b>	<b>Previous Frame</b>	<b>Current Frame</b>
Test 5	Robroy.avi, frame 1736	Robroy.avi, frame 1737
Test 6	Robroy.avi, frame 1737	Robroy.avi, frame 1738
Test 7	Robroy.avi, frame 1738	Robroy.avi, frame 1739
Test 8	Robroy.avi, frame 1739	Robroy.avi, frame 1740



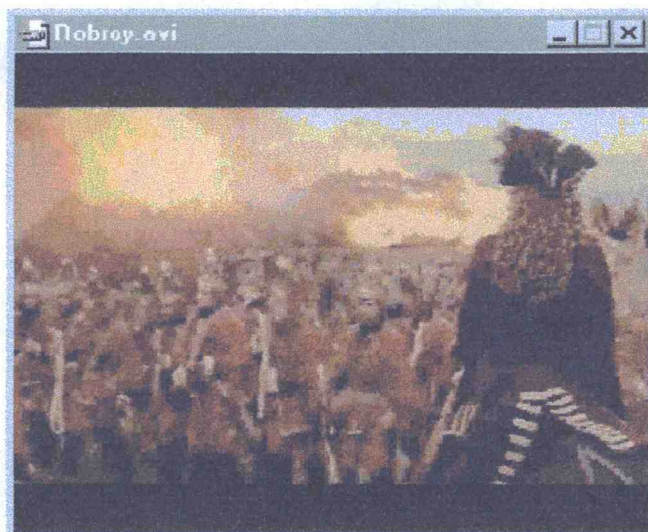
Robroy, frame 1736



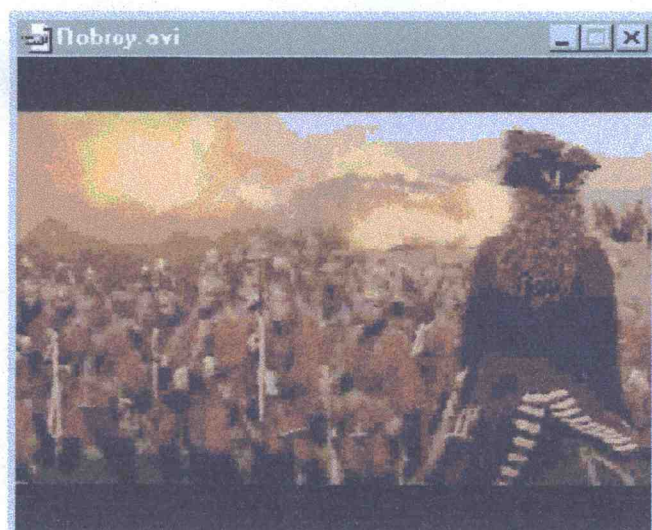
Robroy, frame 1737



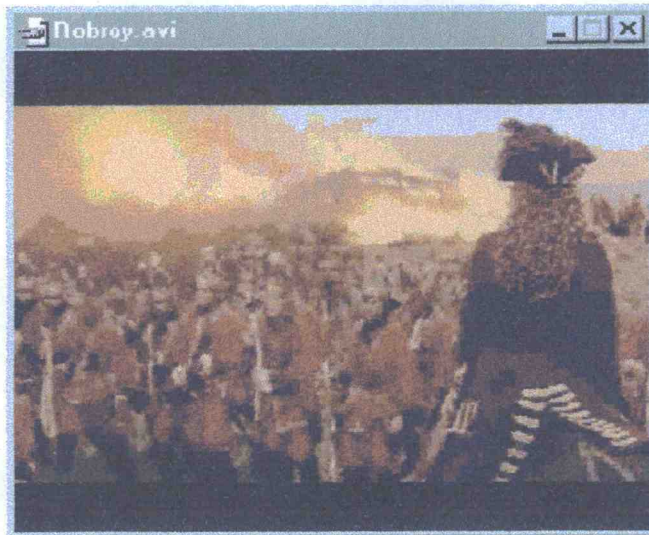
Robroy, frame 1738



Robroy, frame 1739



Robroy, frame 1740



## APPENDIX C

This appendix describes how to run the software used in this research.

There is one executable file named `motion.exe` which contains the necessary code to run any of the tests. The command line options are as follows:

```
motion test curr.bmp prev.bmp outBlock.bmp outVect.bmp > out.txt
```

Where *test* is one of the following:

test	Test description
1	Fixed block size iterative search (normal and enhanced)
2	Fixed block size one-shot search
3	Variable block size search
4	Region motion estimation, fixed block size search
5	Region motion estimation, variable block size search
6	Exhaustive search

The file *curr.bmp* and *prev.bmp* are the Device Independent Bitmap files which represent the current and the previous frame for the specified test (the current implementation is limited to using 256 color bitmap files). The file *outBlock.bmp* is a Device Independent Bitmap file which is created by the `motion.exe` program. This file is the video frame which results from applying the best motion vectors found in the test to the previous frame. The file *outVect.bmp* is only used in the region motion estimation experiments. It is created by `motion.exe` and contains a graphical representation of the motion vectors. The file *out.txt* captures the program's standard output which provides additional information on search performance.

## APPENDIX D

This appendix lists the software used for this research. All of the code is written in standard C++. The compiler used is Borland C++ Builder Standard Edition. The target system was a Pentium 133MHz PC running Microsoft Windows 95. The application was run in Console mode.

### File: ga.h

```
//-----
#ifndef gaH
#define gaH
//-----

// This class a complete Genetic Algorithm implementation. All that
// needs to be supplied is the fitness function (this is an abstract
// base class, so it cannot be instantiated). The way to do this is
// to derive a class from this one, and define the fitness function
// in there.
class GA
{
public:
    GA(unsigned populationSize, unsigned stringLength);
    virtual ~GA();
    virtual void init();
    void work(unsigned numGenerations);
    const char * const getBestString();
    double getBestFitness();
    unsigned getBestGeneration();

protected:
    unsigned populationSize_;
    unsigned stringLength_;

    double selectionProb_;
    double reproductionProb_;
    double mutationProb_;

    unsigned currentGeneration_;

    struct Individual
    {
        char *string_;
        char *tempString_;
        double fitness_;
        double tempFitness_;
    };

    Individual *pool_;
    Individual best_;
    unsigned bestGeneration_;
    bool abortSearch_;

    void evalFitness();
};
```

```

void select();
void reproduce();
void mutate();

// This method needs to be defined by the derived class
// It needs to calculate the fitness of a string, and
// store that value in the fitness_ field of the individual.
virtual void fitness(Individual &indiv) = 0;

virtual void notifyStartGeneration() {};
virtual void notifyEndGeneration() {};
virtual void notifyNewBest() {};
};

#endif
// eof

```

### File: ga.cpp

```

//-----
#include <assert.h>
#include <iostream.h>
#include <memory.h>
#pragma hdrstop

#include "ga.h"
#include "random.h"
//-----

//
// GA::GA()
//
// Constructor for the GA class. The strings are all allocated, but
// not initialized.
//
GA::GA(unsigned populationSize, unsigned stringLength)
: populationSize_(populationSize),
  stringLength_(stringLength)
{
    pool_ = new Individual[populationSize_];
    for (unsigned i=0 ; i<populationSize_ ; i++)
    {
        pool_[i].string_ = new char [stringLength_];
        pool_[i].tempString_ = new char [stringLength_];
        pool_[i].fitness_ = 0.0;
    }
    best_.string_ = new char [stringLength_];
    best_.tempString_ = 0;
    // Initialize the best fitness to a very negative number so that no
    // real fitness would be more negative.
    best_.fitness_ = -1.7e308;
    bestGeneration_ = 0;
    abortSearch_ = false;

    // Setup the default probabilities.
    selectionProb_ = 0.6;
    reproductionProb_ = 0.6;
    mutationProb_ = 0.0005;

```

```

}

//
// GA::~~GA()
//
// The destructor is necessary because the class contains dynamically
// allocated elements.
GA::~~GA()
{
    for (unsigned i=0 ; i<populationSize_ ; i++)
    {
        delete [] pool_[i].string_;
        delete [] pool_[i].tempString_;
    }
    delete [] pool_;
    delete [] best_.string_;
}

//
// GA::init()
//
// This method is used to initialize the strings in the population to
// random bit values.
//
void GA::init()
{
    randomInit();

    currentGeneration_ = 0;

    for (unsigned i=0 ; i<populationSize_ ; i++)
    {
        for (unsigned j=0 ; j<stringLength_ ; j++)
        {
            pool_[i].string_[j] = (char) (randomInt() % 2);
        }
    }
}

//
// GA::work()
//
// This is the main genetic algorithm method. It performs the genetic
// operations for the specified number of generations. It can be
// called several times if desired.
//
void GA::work(unsigned numGenerations)
{
    for (unsigned i=0 ; i<numGenerations && !abortSearch_ ; i++)
    {
        currentGeneration_++;

        notifyStartGeneration();

        evalFitness();
        select();
        reproduce();
        mutate();

        notifyEndGeneration();
    }
}

```

```

//
// GA::getBestString()
//
// Returns a pointer to the best string found so far.
//
const char * const GA::getBestString()
{
    return best_.string_;
}

//
// GA::getBestFitness()
//
// Returns the fitness of the best string found so far.
//
double GA::getBestFitness()
{
    return best_.fitness_;
}

//
// GA::getBestGeneration()
//
// Returns the generation in which the best string found so far
// was found.
//
unsigned GA::getBestGeneration()
{
    return bestGeneration_;
}

//
// GA::evalFitness()
//
// Evaluate the fitness of the entire population of strings.
//
void GA::evalFitness()
{
    for (unsigned i=0 ; i<populationSize_ && !abortSearch_ ; i++)
    {
        fitness(pool_[i]);
        if (pool_[i].fitness_ > best_.fitness_)
        {
            memcpy(best_.string_, pool_[i].string_, stringLength_);
            best_.fitness_ = pool_[i].fitness_;
            bestGeneration_ = currentGeneration_;
            notifyNewBest();
        }
    }
}

//
// GA::select()
//
// Perform the selection operator on the population of strings.
// Tournament selection is used to maintain evolutionary pressure.
//
void GA::select()
{
    for (unsigned i=0 ; i<populationSize_ ; i++)
    {

```



```

    unsigned j;
    j = randomInt() % populationSize_;

    if (pool_[i].fitness_ > pool_[j].fitness_ )
    {
        memcpy(pool_[i].tempString_, pool_[i].string_, stringLength_);
        pool_[i].tempFitness_ = pool_[i].fitness_;
    }
    else
    {
        memcpy(pool_[i].tempString_, pool_[j].string_, stringLength_);
        pool_[i].tempFitness_ = pool_[j].fitness_;
    }
}

for (unsigned i=0 ; i<populationSize_ ; i++)
{
    memcpy(pool_[i].string_, pool_[i].tempString_, stringLength_);
    pool_[i].fitness_ = pool_[i].tempFitness_;
}
}

//
// GA::reproduce()
//
// The reproduction operator. Strings are chosen in pairs, which are
// the subjected to single point crossover.
//
void GA::reproduce()
{
    unsigned mate1;
    unsigned mate2;
    unsigned slice;
    char *temp1;
    char *temp2;
    for (unsigned i=0 ; i<populationSize_ ; i++)
    {
        if ( randomFloat() <= reproductionProb_ )
        {
            mate1 = randomInt() % populationSize_;
            mate2 = randomInt() % populationSize_;
            slice = (randomInt() % (stringLength_ - 1)) + 1;

            temp1 = new char[stringLength_];
            memcpy(temp1, pool_[mate1].string_, slice);
            memcpy(temp1 + slice, pool_[mate2].string_ + slice,
                stringLength_ - slice);

            temp2 = new char[stringLength_];
            memcpy(temp2, pool_[mate2].string_, slice);
            memcpy(temp2 + slice, pool_[mate1].string_ + slice,
                stringLength_ - slice);

            memcpy(pool_[mate1].string_, temp1, stringLength_);
            memcpy(pool_[mate2].string_, temp2, stringLength_);
            delete [] temp1;
            delete [] temp2;
        }
    }
}

//
// GA::mutate()
//

```

```

// The mutation operator.  When a point is chosen for mutation, a
// random bit is used to replace the current one.
//
void GA::mutate()
{
    for (unsigned i=0 ; i<populationSize_ ; i++)
    {
        for (unsigned j=0 ; j<stringLength_ ; j++)
        {
            if ( randomFloat() <= mutationProb_ )
            {
                pool_[i].string_[j] = (char) (randomInt() % 2);
            }
        }
    }
}

// eof

```

## File: gp.h

```

//-----
#ifndef gpH
#define gpH

#include "gpnode.h"
//-----

// This class represents a single GP tree structure.  These are used
// by the GPPool class.  This GP class contains an instance of a
// Node which needs to be defined elsewhere.
class GP
{
    friend ostream & operator <<(ostream & out, const GP & gp);

public:
    GP();
    GP(int minHeight, int maxHeight);
    ~GP();
    GP & operator =(const GP &other);
    unsigned depth(bool refresh=false, unsigned parentDepth=0);
    unsigned length();
    GP * subNode(unsigned index);
    double fitness();
    void crossover(GP *other, double prob, unsigned maxHeight);
    void mutate(double prob, unsigned minHeight, unsigned maxHeight);
private:
    Node node_;
    double fitness_;
    unsigned depth_;
    enum { MAX_CHILDREN=4 };
    GP *children_[MAX_CHILDREN];

    double fitness(unsigned &nodeNumber, unsigned *tPath);
    void traverse(unsigned &length, unsigned index, GP **node);
};

```

```

// This is the Genetic Programming implementation.  It contains all
// of the operators required to evolve genetic programs.
class GPPool
{
    friend ostream & operator <<(ostream & out, const GPPool & pool);

public:
    GPPool(unsigned populationSize, unsigned minTreeHeight,
           unsigned maxTreeHeight);
    virtual ~GPPool();
    virtual void init();
    void work(unsigned numGenerations);
    GP * getBestTree();
    double getBestFitness();
    unsigned getBestGeneration();

protected:
    unsigned populationSize_;
    unsigned minTreeHeight_;
    unsigned maxTreeHeight_;

    double selectionProb_;
    double reproductionProb_;
    double mutationProb_;

    unsigned currentGeneration_;
    unsigned bestGeneration_;

    GP **pool_;
    GP *best_;

    void evalFitness();
    void select();
    void reproduce();
    void mutate();

    virtual void notifyStartGeneration() {};
    virtual void notifyEndGeneration() {};
    virtual void notifyNewBest() {};
};

#endif
// eof

```

## File gp.cpp

```

//-----
#include <assert.h>
#include <memory.h>
#pragma hdrstop

#include "gp.h"
#include "random.h"
//-----

//
// GP::GP()
//

```

```

// Only the root node is made.
//
GP::GP()
{
    fitness_ = 0.0;
    depth_ = 0;

    for (int i=0 ; i<MAX_CHILDREN ; i++)
    {
        children_[i] = 0;
    }
}

//
// GP::GP()
//
// A GP is constructed with every branch of the tree having a depth
// of at least minHeight, but no more then maxHeight.
//
GP::GP(int minHeight, int maxHeight)
{
    //assert(minHeight > 0);
    assert(maxHeight > 0);
    fitness_ = 0.0;
    depth_ = 0;

    bool hasChildren = (minHeight > 1)
        || ((maxHeight > 1) && (randomInt() % maxHeight)!=0);

    for (int i=0 ; i<MAX_CHILDREN ; i++)
    {
        children_[i] = 0;
        if ( hasChildren )
        {
            children_[i] = new GP(minHeight - 1, maxHeight - 1);
        }
    }
}

//
// GP::~~GP
//
// This deletes the nodes of the tree which were dynamically
// allocated.
//
GP::~~GP()
{
    for (int i=0 ; i<MAX_CHILDREN ; i++)
    {
        delete children_[i];
    }
}

//
// GP::operator=()
//
// Assignment operator for the GP class. This needs to be explicitly
// defined since a GP contains dynamically allocated objects.
//
GP & GP::operator =(const GP &other)
{
    if ( this != &other )
    {

```

```

    assert(this);
    assert(&other);
    node_ = other.node_;
    fitness_ = other.fitness_;
    depth_ = other.depth_;

    for (int i=0 ; i<MAX_CHILDREN ; i++)
    {
        delete children_[i];
        children_[i] = 0;
        if ( other.children_[i] )
        {
            children_[i] = new GP(1, 1);
            *children_[i] = *other.children_[i];
        }
    }
    return *this;
}

//
// GP::depth()
//
// Find the depth of the nodes in the tree.
//
unsigned GP::depth(bool refresh, unsigned parentDepth)
{
    if ( refresh )
    {
        depth_ = parentDepth + 1;
        for (int i=0 ; i<MAX_CHILDREN ; i++)
        {
            if ( children_[i] )
            {
                children_[i]->depth(true, depth_);
            }
        }
    }
    return depth_;
}

//
// GP::length()
//
// Find the length of the tree, ie the total number of nodes
// in the tree.
//
unsigned GP::length()
{
    unsigned length = 1;
    for (int i=0 ; i<MAX_CHILDREN ; i++)
    {
        if ( children_[i] )
            length += children_[i]->length();
    }
    return length;
}

//
// GP::subNode()
//
// Return a pointer to the specified node. The "index" argument
// must be in the range of 0..length()-1.

```

```

//
GP * GP::subNode(unsigned index)
{
    unsigned length = 0;
    GP *tempNode = 0;
    GP **node = &tempNode;
    traverse(length, index, node);
    return *node;
}

//
// GP::fitness()
//
// Evaluate the fitness of this GP. The current path is always
// stored in treePath. Some outside routines need to know what
// the path to the node is.
//
static unsigned treePath[20];
double GP::fitness()
{
    unsigned nodeNumber = 0;
    return fitness(nodeNumber, treePath);
}

//
// GP::crossover()
//
// This performs the crossover operation with the current tree,
// and the one specified in the argument list.
//
void GP::crossover(GP *other, double probab, unsigned maxHeight)
{
    // Find a common node which we can switch.
    GP *cross1 = this;
    GP *cross2 = other;
    int branch;
    int height = randomInt() % maxHeight;
    for (int i=0 ; i<height ; i++)
    {
        branch = randomInt() % MAX_CHILDREN;
        if ( cross1->children_[branch]==0 || cross2->children_[branch]==0 )
            break;
        cross1 = cross1->children_[branch];
        cross2 = cross2->children_[branch];
    }

    assert (cross1);
    assert (cross2);

    // Perform the actual crossover.
    GP temp;
    temp = *cross1;
    *cross1 = *cross2;
    *cross2 = temp;
}

//
// GP::mutate()
//
// This method performs a possible mutation on the GP. If a mutation
// occurs, the subtree of the selected node is deleted and replaced
// with a new randomly created tree.
//
void GP::mutate(double probab, unsigned minHeight, unsigned maxHeight)

```

```

{
    // Find a node to mutate ...
    unsigned len = length();
    for (unsigned i=0 ; i<len ; i++)
    {
        if ( probab >= randomFloat() )
        {
            depth(true /*refresh*/);
            GP *node = subNode(randomInt() % length());
            GP newNode(minHeight - node->depth() + 1,
                       maxHeight - node->depth() + 1);
            *node = newNode;

            // Since we modified the tree, update the loop count to
            // correspond to the current tree length.
            len = length();
        }
    }
}

//
// GP::fitness()
//
// This is the recursive fitness method which is called by the
// main fitness routine. It in turn calls the nodes fitness
// function.
//
double GP::fitness(unsigned &nodeNumber, unsigned *tPath)
{
    bool leafNode = true;
    fitness_ = 0.0;
    for (int i=0 ; i<MAX_CHILDREN ; i++)
    {
        if ( children_[i] )
        {
            leafNode = false;
            *tPath = i + 1;
            fitness_ += children_[i]->fitness(nodeNumber, tPath+1);
        }
    }
    *tPath = 0;
    if ( leafNode )
    {
        fitness_ += node_.fitness(nodeNumber++, treePath);
    }
    return fitness_;
}

//
// GP::traverse()
//
// This method traverses the GP.
//
void GP::traverse(unsigned &length, unsigned index, GP **node)
{
    if ( length++ == index )
    {
        *node = this;
    }

    for (int i=0 ; i<MAX_CHILDREN ; i++)
    {
        if ( children_[i] )
        {
            children_[i]->traverse(length, index, node);
        }
    }
}

```

```

    }
}

//
// operator<<()
//
// This is the stream operator for the GP. It is a simple way
// to print out a tree for debugging purposes.
//
ostream & operator <<(ostream & out, const GP & gp)
{
    out << gp.node_;
    out << "(";
    for (int i=0 ; i<GP::MAX_CHILDREN ; i++)
    {
        if ( gp.children_[i] )
        {
            out << *(gp.children_[i]);
        }
    }
    out << ") ";
    return out;
}

//
// GPPool::GPPool()
//
// Constructs a GPPool which is a collection of GPs
//
GPPool::GPPool(unsigned populationSize, unsigned minTreeHeight,
               unsigned maxTreeHeight)
: populationSize_(populationSize),
  minTreeHeight_(minTreeHeight),
  maxTreeHeight_(maxTreeHeight)
{
    pool_ = new GP* [populationSize_];

    // Setup the default probabilities.
    selectionProb_ = 0.6;
    reproductionProb_ = 0.6;
    mutationProb_ = 0.0001;
}

//
// GPPool::~~GPPool()
//
// The destructor is required since the GPPool contains dynamically
// allocated objects.
GPPool::~~GPPool()
{
    for (unsigned i=0 ; i<populationSize_ ; i++)
        delete pool_[i];
    delete [] pool_;
}

//
// GPPool::init()
//
// Initialize the population with random trees, and clear out the
// best GP found so far.

```



```

//
void GPPool::init()
{
    randomInit();

    currentGeneration_ = 0;

    for (unsigned i=0 ; i<populationSize_ ; i++)
    {
        pool_[i] = new GP(minTreeHeight_, maxTreeHeight_);
    }
    best_ = new GP;
    *best_ = *pool_[0];
    bestGeneration_ = 0;
}

//
// GPPool::work()
//
// This is the main genetic programming loop. When called it cycles
// through the indicated number of generations. If desired it can
// be called multiple times.
//
void GPPool::work(unsigned numGenerations)
{
    for (unsigned i=0 ; i<numGenerations ; i++)
    {
        currentGeneration_++;

        notifyStartGeneration();

        evalFitness();
        select();
        reproduce();
        mutate();

        notifyEndGeneration();
    }
}

//
// GPPool::getBestTree()
//
// Return a pointer to the best GP found so far.
//
GP * GPPool::getBestTree()
{
    return best_;
}

//
// GPPool::getBestFitness()
//
// Return the fitness value of the best GP found so far.
//
double GPPool::getBestFitness()
{
    return best_->fitness();
}

//
// GPPool::getBestGeneration()

```

```

//
// Return the generation in which the best GP so far was found.
//
unsigned GPPool::getBestGeneration()
{
    return bestGeneration_;
}

//
// GPPool::evalFitness()
//
// Evaluate the fitness of all of the GPs in the pool.  If a new
// best is found, make a note of it.
//
void GPPool::evalFitness()
{
    for (unsigned i=0 ; i<populationSize_ ; i++)
    {
        if (pool_[i]->fitness() > best_->fitness())
        {
            *best_ = *pool_[i];
            bestGeneration_ = currentGeneration_;
            notifyNewBest();
        }
    }
}

//
// GPPool::select()
//
// Selection operator for the genetic programming.  Tournament selection
// is used to maintain evolutionary pressure.
//
void GPPool::select()
{
    GP *tempPool = new GP[populationSize_];

    for (unsigned i=0 ; i<populationSize_ ; i++)
    {
        unsigned j;
        j = randomInt() % populationSize_;

        if (pool_[i]->fitness() > pool_[j]->fitness() )
        {
            tempPool[i] = *pool_[i];
        }
        else
        {
            tempPool[i] = *pool_[j];
        }
    }

    // Move the temporary pool to the real pool
    for (unsigned i=0 ; i<populationSize_ ; i++)
    {
        // Copy all of the temporary nodes back to main nodes.
        *pool_[i] = tempPool[i];
    }
    delete [] tempPool;
}

//
// GPPool::reproduct()

```

```

//
// Reproduction operator for the genetic programming. This selects
// two nodes at a time, and performs a crossover operation on them.
void GPPool::reproduce()
{
    for (unsigned i=0 ; i<populationSize_ ; i++)
    {
        unsigned mother = randomInt() % populationSize_;
        unsigned father = randomInt() % populationSize_;
        assert (pool_[mother]);
        assert (pool_[father]);
        pool_[mother]->crossover(pool_[father], reproductionProb_,
                                maxTreeHeight_);
    }
}

//
// GPPool::mutate()
//
// Mutation operator for the genetic programming.
//
void GPPool::mutate()
{
    for (unsigned i=0 ; i<populationSize_ ; i++)
    {
        pool_[i]->mutate(mutationProb_, minTreeHeight_, maxTreeHeight_);
    }
}

//
// operator<<()
//
// Stream operator which allows the printing of the best GP found
// so far. This is useful for debugging purposes.
//
ostream & operator << (ostream & out, const GPPool & pool)
{
    out << *(pool.best_);
    return out;
}

// eof

```

## File: bitmap.h

```

//-----
#ifndef bitmapH
#define bitmapH
//-----

// This class is used to store pixel blocks of various sizes.
class Block
{
public:
    Block(unsigned width, unsigned height);

```

```

    ~Block();
    void fill(unsigned char colorIndex);
    unsigned width() { return width_; }
    unsigned height() { return height_; }

    unsigned *pixel_;
    unsigned char *index_;

private:
    unsigned width_;
    unsigned height_;
};

// The BitMap class allows reading, saving, and manipulation of a
// Windows Device Independant Bitmap file. It is currently limited
// to using 256 color bitmap files.
class BitMap
{
public:
    BitMap(const char * file);
    ~BitMap();

    unsigned height();
    unsigned width();
    unsigned char pixelIndex(unsigned x, unsigned y);
    unsigned pixelColor(unsigned x, unsigned y);
    unsigned color(unsigned index);
    Block * getBlock(unsigned x, unsigned y, unsigned width,
                    unsigned height);

    void clear();
    void putBlock(Block *blk, unsigned x, unsigned y);
    void writeFile(const char * file);

protected:
    enum
    {
        DATAPTR_OFFSET = 0x0000000AL,
        WIDTH_OFFSET   = 0x00000012L,
        HEIGHT_OFFSET  = 0x00000016L,
        COLORTABLE_OFFSET = 0x00000036L,
    };

    unsigned *colorTable_;
    unsigned char *pixMap_;
    unsigned char *data_;
};

#endif
// eof

```

### File: bitmap.cpp

```

//-----
#include <assert.h>
#include <fstream.h>
#pragma hdrstop

```

```

#include "bitmap.h"
//-----

//
// Block::Block()
//
// The constructor create a block of the given size. The data
// is not initialized.
//
Block::Block(unsigned width, unsigned height)
: width_(width), height_(height)
{
    pixel_ = new unsigned [height_ * width_];
    index_ = new unsigned char [height_ * width_];
}

//
// Block::~Block()
//
// The destructor is necessary since the Block class contains
// dynamically allocated elements.
//
Block::~Block()
{
    delete [] pixel_;
    delete [] index_;
}

//
// Block::fill()
//
// This routine will fill all of the block's pixels with the given
// color index value.
//
void Block::fill(unsigned char colorIndex)
{
    for (unsigned i=0 ; i<width_ * height_ ; i++)
    {
        index_[i] = colorIndex;
    }
}

//
// BitMap::BitMap()
//
// Constructor for the BitMap class. When a BitMap is constructed
// it is initialized with the data from the specified bitmap file.
//
BitMap::BitMap(const char * file)
{
    // Format of BMP file
    //
    // 0012 - 0015 : width of image
    // 0016 - 0019 : height of image
    // 0036 - 0435 : color table, 256 entries, each entry is four bytes
    //              BB GG RR 00
    // 0436 - eof  : pixel data, from left to right, bottom to top; each
    //              pixel represents an index into the color table
    //
}

```

```

    ifstream in(file, ios::in+ios::binary);

    unsigned char data;
    unsigned totalSize = 0;
    in.seekg(2L);
    for (int i=0 ; i<sizeof(unsigned) ; i++)
    {
        in.get(data);
        totalSize += data << (8 * i);
    }

    in.seekg(0L);
    data_ = new unsigned char [totalSize];
    for (unsigned i=0 ; i<totalSize ; i++)
    {
        in.get(data_[i]);
    }

    colorTable_ = (unsigned *) (data_ + COLORTABLE_OFFSET);
    pixMap_ = data_ + *((unsigned *) (data_ + DATAPTR_OFFSET));
}

//
// BitMap::~BitMap()
//
// The destructor is necessary since the class contains dynamically
// allocated elements.
//
BitMap::~BitMap()
{
    delete [] data_;
}

//
// BitMap::width()
//
// Return the width of the bitmap image in pixels.
//
unsigned BitMap::width()
{
    return *((unsigned *) (data_ + WIDTH_OFFSET));
}

//
// BitMap::height()
//
// Return the height of the bitmap image in pixels.
//
unsigned BitMap::height()
{
    return *((unsigned *) (data_ + HEIGHT_OFFSET));
}

//
// BitMap::pixelColor()
//
// Return the color of the specified pixel. The format of
// the color is as follows:
// 0000 0000 BBBB BBBB GGGG GGGG RRRR RRRR
// It is a 32 bit value with the RGB values each taking up
// one byte. The top eight bits are 0.

```

```

//
unsigned BitMap::pixelColor(unsigned x, unsigned y)
{
    return color(pixelIndex(x, y));
}

//
// BitMap::pixelIndex()
//
// Return the color index value of the specified pixel.
//
unsigned char BitMap::pixelIndex(unsigned x, unsigned y)
{
    assert(x < width());
    assert(y < height());
    return pixMap_[y * width() + x];
}

//
// BitMap::color()
//
// Return the color of the specified index value. The format of
// the color is as follows:
//   0000 0000 BBBB BBBB GGGG GGGG RRRR RRRR
// It is a 32 bit value with the RGB values each taking up
// one byte. The top eight bits are 0.
//
unsigned BitMap::color(unsigned index)
{
    assert(index < 256);
    return colorTable_[index];
}

//
// BitMap::getBlock()
//
// This method returns a pointer to a Block which contains the
// image data copied from the specified coordinates.
//
Block * BitMap::getBlock(unsigned x, unsigned y,
                        unsigned width, unsigned height)
{
    Block *blk = new Block(width, height);
    for (unsigned i=0 ; i<height ; i++)
    {
        for (unsigned j=0 ; j<width ; j++)
        {
            blk->pixel_[i * width + j] = pixelColor(x+j, y+i);
            blk->index_[i * width + j] = pixelIndex(x+j, y+i);
        }
    }
    return blk;
}

//
// BitMap::clear()
//
// This method erases the entire bitmap image by setting all of the
// pixel color index values to 0xff.
//
void BitMap::clear()

```

```

{
    for (unsigned y=0 ; y<height() ; y++)
    {
        for (unsigned x=0 ; x<width() ; x++)
        {
            pixMap_[y*width() + x] = 0xff;
        }
    }
}

//
// BitMap::putBlock()
//
// Copies the specified Block to the bitmap image at the
// specified coordinates.
//
void BitMap::putBlock(Block *blk, unsigned x, unsigned y)
{
    for (unsigned i=0 ; i<blk->height() ; i++)
    {
        for (unsigned j=0 ; j<blk->width() ; j++)
        {
            pixMap_[(y+i) * width() + x + j]
                = blk->index_[i * blk->width() + j];
        }
    }
}

//
// BitMap::writeFile()
//
// Write the bitmap to the specifed file. The resulting file is a
// Windows Device Independant Bitmap file. If the file already
// exists it is overwritten.
//
void BitMap::writeFile(const char * file)
{
    ofstream out(file, ios::out+ios::binary);
    unsigned totalSize = *((unsigned*)(data_+2));
    for (unsigned i=0 ; i<totalSize ; i++)
    {
        out.put(data_[i]);
    }
}

// eof

```

## File: random.h

```

//-----
#ifndef randomH
#define randomH
//-----

// This is used to set the seed to the current time.
void randomInit();

// Return a random integer of 0 to MAX_RANDOM.
unsigned randomInt();

```



```

// Return a random float, in the range of 0 to 1.
double randomFloat();

#endif
// eof

```

### File: random.cpp

```

//-----
#include <assert.h>
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
#pragma hdrstop

#include "random.h"
//-----

//
// randomInit()
//
// This is used to set the seed to the current time.
//
void randomInit()
{
    srand((unsigned)time(0));
}

//
// randomInt()
//
// Return a random integer of 0 to MAX_RAND.
//
unsigned randomInt()
{
    return rand();
}

//
// randomFloat()
//
// Return a random float, in the range of 0 to 1.
//
double randomFloat()
{
    unsigned randMax = (unsigned) RAND_MAX * (unsigned) RAND_MAX;
    unsigned random = (unsigned) rand() * (unsigned) rand();
    return (double) random / (double) randMax;
}

// eof

```

**File: gpnode.h**

```

//-----
#ifndef gpnodeH
#define gpnodeH

#include "bitmap.h"
#include <iostream.h>
//-----

// This is the Node class used by the Genetic Programming portion of
// this program. This class contains the data values which occur at
// every node in a GP tree. So this class is problem specific.
class Node
{
    friend ostream & operator <<(ostream & out, const Node & node);

public:
    // Constructor initializes the nodes displacement vectors
    // to random values.
    Node();

    // This method returns the current fitness value of this
    // node.
    double fitness(unsigned nodeNumber, unsigned *path);

    // Static value used by the fitness function to determine
    // if it should apply the motion vectors to an image.
    static bool useNode_;

private:
    // The most recently calculated fitness value.
    double fitness_;

    // The horizontal displacement motion vector.
    int xMotion_;

    // The vertical displacement motion vector.
    int yMotion_;
};

#endif
// eof

```

**File: motion.cpp**

```

//-----
#include <assert.h>
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#pragma hdrstop

#include "bitmap.h"

```

```

#include "ga.h"
#include "gp.h"
#include "random.h"
//-----

//
// This is a derived class of BitMap. The main difference is that the
// getBlock() and putBlock() methods use block index values rather than
// actual image coordinates.
//
class MyBitMap : public BitMap
{
public:
    MyBitMap(const char *file, unsigned blkWidth, unsigned blkHeight)
        : BitMap(file), blockWidth_(blkWidth), blockHeight_(blkHeight) {}

    unsigned getNumBlocks();
    Block *myGetBlock(unsigned blockIndex, int deltaX=0, int deltaY=0);
    void myPutBlock(Block *blk, unsigned index);
private:
    unsigned blockWidth_;
    unsigned blockHeight_;
};

unsigned MyBitMap::getNumBlocks()
{
    return (width() / blockWidth_) * (height() / blockHeight_);
}

Block *MyBitMap::myGetBlock(unsigned blockIndex, int deltaX, int deltaY)
{
    assert(blockIndex < getNumBlocks());

    int x = (blockIndex % (width() / blockWidth_)) * blockWidth_;
    x += deltaX;
    if ( x < 0) x = 0;
    if ((unsigned) (x + blockWidth_) >= width())
        x = width() - blockWidth_ - 1;

    int y = (blockIndex / (width() / blockWidth_)) * blockHeight_;
    y += deltaY;
    if ( y < 0) y = 0;
    if ((unsigned) (y + blockHeight_) >= height())
        y = height() - blockHeight_ - 1;

    return getBlock(x, y, blockWidth_, blockHeight_);
}

void MyBitMap::myPutBlock(Block *blk, unsigned index)
{
    assert(blk->width() == blockWidth_);
    assert(blk->height() == blockHeight_);

    int x = (index % (width() / blockWidth_)) * blockWidth_;
    int y = (index / (width() / blockWidth_)) * blockHeight_;
    putBlock(blk, x, y);
}

// This is used to store a blocks coordinates along with it's

```

```

// motion vectors. BlockElement represents one block, while
// blockList points to a list of BlockElements which cover the
// entire image under test.
struct BlockElement
{
    BlockElement() {used = false;}
    bool used;
    unsigned index;
    unsigned x;
    unsigned y;
    unsigned width;
    unsigned height;
    int xMotion;
    int yMotion;
};

BlockElement *blockList = 0;

//
// Routine used for sorting the blockList with the qsort() routine.
//
int sortFunc(const void *a, const void *b)
{
    const BlockElement *blk1 = (const BlockElement *) a;
    const BlockElement *blk2 = (const BlockElement *) b;

    if (blk1->xMotion < blk2->xMotion)
        return -1;
    if (blk1->xMotion > blk2->xMotion)
        return 1;

    // At this point we know the xMotion vectors are the same, move
    // on to the next sorting criteria.
    if (blk1->yMotion < blk2->yMotion)
        return -1;
    if (blk1->yMotion > blk2->yMotion)
        return 1;
    return 0;
}

// Global file name pointers
char *currFrameFile = 0;
char *prevFrameFile = 0;
char *outFrameSrcFile = 0;
char *outFrameFile = 0;
char *outFrameFile2 = 0;

// Global picture elements.
MyBitMap *cBmp;
MyBitMap *pBmp;
MyBitMap *oBmp;

BitMap *currBmp;
BitMap *prevBmp;
BitMap *outBmp;

//
// Global routine used to compare two image blocks of the same size.
// An error value is return which indicates by how much the color
// values of all the pixels in the two blocks differ.
//
double compareBlock(Block *blk1, Block *blk2)

```

```

{
    assert(blk1->width() == blk2->width());
    assert(blk1->height() == blk2->height());
    int red1, green1, blue1;
    int red2, green2, blue2;
    unsigned pix1, pix2;
    double colorError = 0.0;
    for (unsigned y=0 ; y<blk1->height() ; y++)
    {
        for (unsigned x=0 ; x<blk1->width() ; x++)
        {
            pix1 = blk1->pixel_[y * blk1->width() + x];
            pix2 = blk2->pixel_[y * blk2->width() + x];
            red1 = pix1 & 0x000000ff;
            red2 = pix2 & 0x000000ff;
            green1 = (pix1 >> 8) & 0x000000ff;
            green2 = (pix2 >> 8) & 0x000000ff;
            blue1 = (pix1 >> 16) & 0x000000ff;
            blue2 = (pix2 >> 16) & 0x000000ff;
            colorError += abs(red1-red2) + abs(green1-green2)
                + abs(blue1-blue2);
        }
    }
    return colorError;
}

////////////////////////////////////
////////// FIXED BLOCK SIZE ITERATIVE MOTION ESTIMATION //////////
////////////////////////////////////

//
// We need to derive a class from GA in order to define the fitness
// function.
//
class MyGA : public GA
{
public:
    MyGA(unsigned populationSize, unsigned stringLength,
          int vectorLength, int vectorRange)
        : GA(populationSize, stringLength),
          vectorLength_(vectorLength), vectorRange_(vectorRange),
          blockCompares_(0), blockComparesToBest_(0) {};
    void setInitialValue(unsigned index, const char * const str);
    void setAnchorBlock(int blk);
    int stringToNum(const char * const string, unsigned length);
    unsigned blockCompares() {return blockComparesToBest_;}
    const int vectorLength_;
    const int vectorRange_;
private:
    int anchorBlock_;
    unsigned blockCompares_;
    unsigned blockComparesToBest_;
    virtual void fitness(Individual &indiv);
    virtual void notifyStartGeneration();
    virtual void notifyEndGeneration();
    virtual void notifyNewBest();
};

//
// MyGA::fitness
//
// This is what evaluates the fitness of the motion vectors.
//

```

```

void MyGA::fitness(Individual &indiv)
{
    // Translate the motion vectors from from binary strings to
    // integer values.
    int colVect = stringToNum(indiv.string_, vectorLength_);
    int rowVect = stringToNum(indiv.string_+vectorLength_, vectorLength_);

    // Get the src and destination blocks
    Block *srcBlock = cBmp->myGetBlock(anchorBlock_);
    Block *destBlock = pBmp->myGetBlock(anchorBlock_, rowVect, colVect);

    // Compute the fitness of the vectors by comparing the blocks.
    indiv.fitness_ = compareBlock(srcBlock, destBlock);
    blockCompares_++;

    delete srcBlock;
    delete destBlock;
    // Change the sign since the GA maximizes the fitness.
    indiv.fitness_ *= -1.0;
}

//
// MyGA::notifyStartGeneration
//
// This method gets called at the start of every generation.
//
void MyGA::notifyStartGeneration()
{
}

//
// MyGA::notifyEndGeneration
//
// This method gets called at the end of every generation.
//
void MyGA::notifyEndGeneration()
{
    cout << "gen: " << currentGeneration_
         << ", fit: " << best_.fitness_ << endl;
}

//
// MyGA::notifyNewBest
//
// This method gets called whenever a new best is found.
//
void MyGA::notifyNewBest()
{
    if ( best_.fitness_ > -0.5 )
        abortSearch_ = true;

    blockComparesToBest_ = blockCompares_;
}

//
// MyGA::stringToNum
//
// This method takes a bit string and converts it into a vector.
//
int MyGA::stringToNum(const char * const string, unsigned length)
{
    int vect = 0;

```

```

    for (unsigned i=0 ; i<length ; i++)
    {
        vect = (vect << 1) + string[i];
    }
    vect -= (vectorRange_ - 1) / 2;
    return vect;
}

//
// MyGA::setInitialValue
//
// This method is used to push a value into the GP's pool of strings.
// This helps bias the population, and give it a starting point.
//
void MyGA::setInitialValue(unsigned index, const char * const str)
{
    assert(index < populationSize_);
    assert(pool_ != 0);
    memcpy(pool_[index].string_, str, stringLength_);
}

//
// MyGA::setAnchorBlock
//
// Set the current block being worked on, so the GA can use this
// information.
//
void MyGA::setAnchorBlock(int blk)
{
    anchorBlock_ = blk;
}

//
// This method is the main routine of the Genetic Algorithm portion
// of the motion estimator.
//
double gaPoint(unsigned numGenerations, unsigned popSize,
               unsigned blockIdx)
{
    const int VECTOR_LENGTH = 4;
    const unsigned STRING_LENGTH = VECTOR_LENGTH * 2;
    const int VECTOR_RANGE = 16;

    // Create the GA and do the work.
    MyGA ga(popSize, STRING_LENGTH, VECTOR_LENGTH, VECTOR_RANGE);
    ga.init();
    ga.setInitialValue(0, "\x00\x01\x01\x01\x00\x01\x01\x01");
    ga.setInitialValue(1, "\x00\x01\x01\x01\x00\x01\x01\x01");
    ga.setAnchorBlock(blockIndex);
    ga.work(numGenerations);

    // Display the best one found.
    const char * const str = ga.getBestString();
    int xVect = ga.stringToNum(str, ga.vectorLength_);
    int yVect = ga.stringToNum(str+ga.vectorLength_, ga.vectorLength_);

    Block *copyBlock = pBmp->myGetBlock(blockIndex, xVect, yVect);
    oBmp->myPutBlock(copyBlock, blockIndex);
    delete copyBlock;

    cout << "blk: " << blockIdx << ", "
         << " deltaX: " << xVect << ", deltaY: " << yVect
         << ", #compares: " << ga.blockCompares() << endl;
}

```

```

        return ga.getBestFitness();
    }

//
// Fixed block size, iterative motion estimation
//
void gaFixedIterative()
{
    const unsigned NUM_GENERATIONS = 20;
    const unsigned POPULATION_SIZE = 50;
    const unsigned BLOCK_SIZE = 8;

    cBmp = new MyBitMap(currFrameFile, BLOCK_SIZE, BLOCK_SIZE);
    pBmp = new MyBitMap(prevFrameFile, BLOCK_SIZE, BLOCK_SIZE);
    oBmp = new MyBitMap(outFrameSrcFile, BLOCK_SIZE, BLOCK_SIZE);
    oBmp->clear();

    // Display the initial conditions.
    cout << "// Fixed Block Size Iterative Motion Estimation." << endl;
    cout << "// Creating GA with:" << endl;
    cout << "// # generations: " << NUM_GENERATIONS << endl;
    cout << "// population size: " << POPULATION_SIZE << endl;

    double totalFitness = 0.0;
    for (unsigned i=0 ; i<cBmp->getNumBlocks() ; i++)
    {
        totalFitness += gaPoint(NUM_GENERATIONS, POPULATION_SIZE, i);
    }
    oBmp->writeFile(outFrameFile);
    cout << "// totalFitness = " << totalFitness << endl;

    delete cBmp;
    delete pBmp;
    delete oBmp;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// FIXED BLOCK SIZE ONE SHOT MOTION ESTIMATION //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//
// We need to derive a class from GA in order to define the fitness
// function.
//
class MyGA2 : public GA
{
public:
    MyGA2(unsigned populationSize, unsigned stringLength,
          int vectorLength, int vectorRange)
        : GA(populationSize, stringLength),
          vectorLength_(vectorLength), vectorRange_(vectorRange),
          blockCompares_(0), blockComparesToBest_(0) {}
    int stringToNum(const char * string, unsigned index);
    unsigned blockCompares() {return blockCompares_;}
    const int vectorRange_;
    const unsigned vectorLength_;
private:
    unsigned blockCompares_;
    unsigned blockComparesToBest_;
    virtual void fitness(Individual &indiv);
    virtual void notifyStartGeneration();
    virtual void notifyEndGeneration();
};

```



```

        virtual void notifyNewBest();
};

//
// MyGA2::fitness
//
// This is what evaluates the fitness of the motion vectors.
//
void MyGA2::fitness(Individual &indiv)
{
    indiv.fitness_ = 0.0;
    int colVect, rowVect;
    Block *srcBlock;
    Block *destBlock;
    for (unsigned i=0 ; i<cBmp->getNumBlocks() ; i++)
    {
        // Translate the motion vectors from from binary strings to
        // integer values.
        colVect = stringToNum(indiv.string_, i);
        rowVect = stringToNum(indiv.string_+vectorLength_, i);

        // Get the blocks to compare.
        srcBlock = cBmp->myGetBlock(i);
        destBlock = pBmp->myGetBlock(i, rowVect, colVect);

        // Compute the fitness of the vectors.
        indiv.fitness_ += compareBlock(srcBlock, destBlock);
        blockCompares_++;

        delete srcBlock;
        delete destBlock;
    }
    // Change the sign since the GA maximizes the fitness.
    indiv.fitness_ *= -1.0;
}

//
// MyGA2::notifyStartGeneration
//
// This method gets called at the start of every generation.
//
void MyGA2::notifyStartGeneration()
{
}

//
// MyGA2::notifyEndGeneration
//
// This method gets called at the end of every generation.
//
void MyGA2::notifyEndGeneration()
{
    cout << "gen: " << currentGeneration_
         << ", fit: " << best_.fitness_ << endl;
}

//
// MyGA2::notifyNewBest
//
// This method gets called whenever a new best is found.
//
void MyGA2::notifyNewBest()

```

```

{
    blockComparesToBest_ = blockCompares_;
}

//
// MyGA2::stringToNum
//
// Converts the bit strings used by the GA into motion vector
// values.
//
int MyGA2::stringToNum(const char * string, unsigned index)
{
    int vect = 0;
    string += index * (vectorLength_ * 2);
    for (unsigned i=0 ; i<vectorLength_ ; i++)
    {
        vect = (vect << 1) + string[i];
    }
    vect -= (vectorRange_ - 1) / 2;
    return vect;
}

//
// Fixed block size, one shot motion estimation.
//
void gaFixedOneShot()
{
    const unsigned BLOCK_SIZE = 8;
    cBmp = new MyBitMap(currFrameFile, BLOCK_SIZE, BLOCK_SIZE);
    pBmp = new MyBitMap(prevFrameFile, BLOCK_SIZE, BLOCK_SIZE);
    oBmp = new MyBitMap(outFrameSrcFile, BLOCK_SIZE, BLOCK_SIZE);
    oBmp->clear();

    const unsigned NUM_GENERATIONS = 50;
    const unsigned POPULATION_SIZE = 500;
    const unsigned VECTOR_LENGTH = 4;
    const unsigned STRING_LENGTH = VECTOR_LENGTH * 2 * cBmp->getNumBlocks();
    const int VECTOR_RANGE = 16;

    // Display the initial conditions.
    cout << "// Fixed Block Size One Shot Motion Estimation." << endl;
    cout << "// Creating GA with:" << endl;
    cout << "// # generations: " << NUM_GENERATIONS << endl;
    cout << "// population size: " << POPULATION_SIZE << endl;
    cout << "// string length: " << STRING_LENGTH << endl;

    // Create the GA and do the work.
    MyGA2 ga(POPULATION_SIZE, STRING_LENGTH, VECTOR_LENGTH, VECTOR_RANGE);
    ga.init();
    ga.work(NUM_GENERATIONS);

    // Display the best one found.
    const char * str = ga.getBestString();
    int xVect, yVect;
    Block *copyBlock;
    for ( unsigned i=0 ; i<cBmp->getNumBlocks() ; i++)
    {
        xVect = ga.stringToNum(str, i);
        yVect = ga.stringToNum(str+ga.vectorLength_, i);

        copyBlock = pBmp->myGetBlock(i, xVect, yVect);
        oBmp->myPutBlock(copyBlock, i);
        delete copyBlock;
    }
}

```

```

        cout << "blk: " << i << ", "
              << " deltaX: " << xVect << ", deltaY: " << yVect
              << ", #compares: " << ga.blockCompares() << endl;
    }
    oBmp->writeFile(outFrameFile);

    delete cBmp;
    delete pBmp;
    delete oBmp;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// VARIABLE BLOCK SIZE MOTION ESTIMATION //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//
// We need to derive a class from GP in order to define the fitness
// function.
//
class MyGPPool : public GPPool
{
public:
    MyGPPool(unsigned populationSize, unsigned minTreeHeight,
             unsigned maxTreeHeight)
        : GPPool(populationSize, minTreeHeight, maxTreeHeight)
        {blockCompares_ = 0;};
    static unsigned blockCompares_;
    static unsigned blockComparesToBest_;
private:
    virtual void notifyStartGeneration();
    virtual void notifyEndGeneration();
    virtual void notifyNewBest();
};

// The total number of compares performed.
unsigned MyGPPool::blockCompares_ = 0;
// The number of compares performed to find the current best.
unsigned MyGPPool::blockComparesToBest_ = 0;

// Used to indicate to the fitness function that it should store
// the motion vectors in the blockList.
bool Node::useNode_ = false;

//
// This is the ostream operator<< for the Node class.
//
ostream & operator <<(ostream & out, const Node & node)
{
    out << '(' << node.xMotion_ << ',' << node.yMotion_ << ')';
    return out;
}

//
// Node::Node
//
// Constructor for the node object.
//
Node::Node()
{
    const VECTOR_RANGE = 16;
    // The X and Y motion vectors can be positive or negative,
    // so VECTOR_RANGE/2 is subtracted to make this possible.

```

```

    fitness_ = 0.0;
    xMotion_ = (randomInt() % VECTOR_RANGE) - (VECTOR_RANGE-1) / 2;
    yMotion_ = (randomInt() % VECTOR_RANGE) - (VECTOR_RANGE-1) / 2;
}

//
// Node::fitness
//
// This method calculates the fitness of the node. It does this by
// applying the motion vectors to the image sequence, and using the
// image differences as a fitness value. The nodeNumber parameter
// is used to determine which block this node refers to.
//
double Node::fitness(unsigned nodeNumber, unsigned *path)
{
    // Start by determining the coordinates of out node
    int x = 0;
    unsigned width = currBmp->width();
    int y = 0;
    unsigned height = currBmp->height();

    int i = 0;
    while ( path[i] )
    {
        switch (path[i])
        {
            case 1:
                width /= 2;
                height /= 2;
                break;
            case 2:
                x += width / 2;
                width /= 2;
                height /= 2;
                break;
            case 3:
                y += height / 2;
                width /= 2;
                height /= 2;
                break;
            case 4:
                x += width / 2;
                y += height / 2;
                width /= 2;
                height /= 2;
                break;
            default:
                assert(!"invalid path");
        }
        i++;
    }

    // Make sure the moved block is in the frame.
    Block *srcBlock = currBmp->getBlock(x, y, width, height);
    int destX = x + xMotion_;
    int destY = y + yMotion_;
    if (destX<0) destX = 0;
    if (destY<0) destY = 0;
    if ( destX + width >= prevBmp->width() )
        destX = prevBmp->width() - width - 1;
    if ( destY + height >= prevBmp->height() )
        destY = prevBmp->height() - height - 1;
    Block *destBlock = prevBmp->getBlock(destX, destY, width, height);

    fitness_ = compareBlock(srcBlock, destBlock);
}

```

```

MyGPPool::blockCompares_++;

if ( useNode_ )
{
    outBmp->putBlock(destBlock, x, y);

    if (blockList)
    {
        blockList[nodeNumber].used = true;
        blockList[nodeNumber].x = x;
        blockList[nodeNumber].y = y;
        blockList[nodeNumber].width = width;
        blockList[nodeNumber].height = height;
        blockList[nodeNumber].xMotion = xMotion_;
        blockList[nodeNumber].yMotion = yMotion_;
    }
}

delete srcBlock;
delete destBlock;

// Change the sign since the GA maximizes the fitness.
fitness_ *= -1.0;
return fitness_;
}

//
// MyGPPool::notifyStartGeneration
//
// This method gets called at the start of every generation.
//
void MyGPPool::notifyStartGeneration()
{
}

//
// MyGPPool::notifyEndGeneration
//
// This method gets called at the end of every generation.
//
void MyGPPool::notifyEndGeneration()
{
    cout << "gen: " << currentGeneration_
         << ", fit: " << best_->fitness() << endl;
}

//
// MyGPPool::notifyNewBest
//
// This method gets called whenever a new best is found.
//
void MyGPPool::notifyNewBest()
{
    blockComparesToBest_ = blockCompares_;
}

//
// Variable block size motion estimation.
//
void gpVariable()
{
    currBmp = new BitMap(currFrameFile);
}

```

```

prevBmp = new BitMap(prevFrameFile);
outBmp = new BitMap(outFrameSrcFile);
outBmp->clear();

const unsigned NUM_GENERATIONS = 50;
const unsigned POPULATION_SIZE = 500;
const unsigned MIN_TREE_HEIGHT = 2;
const unsigned MAX_TREE_HEIGHT = 6;

// Display the initial conditions.
cout << "// Variable Block Size Motion Estimation." << endl;
cout << "// Creating GP with:" << endl;
cout << "// # generations: " << NUM_GENERATIONS << endl;
cout << "// population size: " << POPULATION_SIZE << endl;
cout << "// min tree height: " << MIN_TREE_HEIGHT << endl;
cout << "// max tree height: " << MAX_TREE_HEIGHT << endl;

// A height of 7 generates a tree with 4096 leaf nodes (5461 nodes
// total), which covers a 64x64 image.
MyGPPool pool(POPULATION_SIZE, MIN_TREE_HEIGHT, MAX_TREE_HEIGHT);
pool.init();
pool.work(NUM_GENERATIONS);
GP * best = pool.getBestTree();

cout << "#compares: " << pool.blockComparesToBest_ << endl;
cout << "The best found is:" << endl << *best << endl;
cout << "in generation: " << pool.getBestGeneration() << endl;
cout << "with fitness is: " << pool.getBestFitness() << endl;

// Write the data to an output file.
Node::useNode_ = true;
best->fitness();
Node::useNode_ = false;
outBmp->writeFile(outFrameFile);

delete currBmp;
delete prevBmp;
delete outBmp;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////// FIXED BLOCK SIZE REGION MOTION ESTIMATION //////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//
// This method is the main routine of the Genetic Algorithm portion
// of the motion estimator.
//
double gaFixedRegionPoint(unsigned numGenerations, unsigned popSize,
                          unsigned blockIdx)
{
    const int VECTOR_LENGTH = 4;
    const unsigned STRING_LENGTH = VECTOR_LENGTH * 2;
    const int VECTOR_RANGE = 16;

    // Create the GA and do the work.
    MyGA ga(popSize, STRING_LENGTH, VECTOR_LENGTH, VECTOR_RANGE);
    ga.init();
    ga.setInitialValue(0, "\x00\x01\x01\x01\x00\x01\x01\x01");
    ga.setInitialValue(1, "\x00\x01\x01\x01\x00\x01\x01\x01");
    ga.setAnchorBlock(blockIdx);
    ga.work(numGenerations);

    // Display the best one found.

```

```

const char * const str = ga.getBestString();
int xVect = ga.stringToNum(str, ga.vectorLength_);
int yVect = ga.stringToNum(str+ga.vectorLength_, ga.vectorLength_);

Block *copyBlock = pBmp->myGetBlock(blockIndex, xVect, yVect);
oBmp->myPutBlock(copyBlock, blockIndex);
delete copyBlock;

blockList[blockIndex].index = blockIndex;
blockList[blockIndex].xMotion = xVect;
blockList[blockIndex].yMotion = yVect;

cout << "blk: " << blockIndex << ", "
      << " deltaX: " << xVect << ", deltaY: " << yVect
      << ", #compares: " << ga.blockCompares() << endl;

return ga.getBestFitness();
}

//
// Fixed block size region estimator.
//
void gaFixedRegion()
{
    const unsigned NUM_GENERATIONS = 20;
    const unsigned POPULATION_SIZE = 50;
    const unsigned BLOCK_SIZE = 8;

    cBmp = new MyBitMap(currFrameFile, BLOCK_SIZE, BLOCK_SIZE);
    pBmp = new MyBitMap(prevFrameFile, BLOCK_SIZE, BLOCK_SIZE);
    oBmp = new MyBitMap(outFrameSrcFile, BLOCK_SIZE, BLOCK_SIZE);
    oBmp->clear();

    blockList = new BlockElement [cBmp->getNumBlocks()];

    // Display the initial conditions.
    cout << "// Fixed Block Size Iterative Motion Estimation." << endl;
    cout << "// Creating GA with:" << endl;
    cout << "// # generations: " << NUM_GENERATIONS << endl;
    cout << "// population size: " << POPULATION_SIZE << endl;

    double totalFitness = 0.0;
    for (unsigned i=0 ; i<cBmp->getNumBlocks() ; i++)
    {
        totalFitness += gaFixedRegionPoint(NUM_GENERATIONS,
                                           POPULATION_SIZE, i);
    }
    oBmp->writeFile(outFrameFile);
    cout << "// totalFitness = " << totalFitness << endl;

    // Sort the blockList
    //qsort((void *) blockList, cBmp->getNumBlocks(), sizeof(blockList[0]),
    //      sortFunc);

    // We will now loop through all of the blocks and save them to a BMP
    // file using a different color for each motion vector.
    MyBitMap ooBmp(outFrameSrcFile, BLOCK_SIZE, BLOCK_SIZE);
    ooBmp.clear();
    Block motionBlock(BLOCK_SIZE, BLOCK_SIZE);
    unsigned c;
    for (unsigned i=0 ; i<ooBmp.getNumBlocks() ; i++)
    {
        c = ((blockList[i].xMotion & 0x0f) << 4)
           | (blockList[i].yMotion & 0x0f);
        motionBlock.fill((unsigned char)c);
    }
}

```

```

        ooBmp.myPutBlock(&motionBlock, blockList[i].index);
    }
    ooBmp.writeFile(outFrameFile2);

    delete [] blockList;
    blockList = 0;
    delete cBmp;
    delete pBmp;
    delete oBmp;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// VARIABLE BLOCK SIZE REGION MOTION ESTIMATION //////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//
// Variable block size region estimator.
//
void gpVariableRegion()
{
    currBmp = new BitMap(currFrameFile);
    prevBmp = new BitMap(prevFrameFile);
    outBmp = new BitMap(outFrameSrcFile);
    outBmp->clear();

    // The actual maximum number of nodes in a depth=6 tree is 1365
    const NUM_BLOCK_ELEMENTS = 1400;
    blockList = new BlockElement [NUM_BLOCK_ELEMENTS];

    const unsigned NUM_GENERATIONS = 50;
    const unsigned POPULATION_SIZE = 500;
    const unsigned MIN_TREE_HEIGHT = 2;
    const unsigned MAX_TREE_HEIGHT = 6;

    // Display the initial conditions.
    cout << "// Variable Block Size Motion Estimation." << endl;
    cout << "// Creating GP with:" << endl;
    cout << "// # generations: " << NUM_GENERATIONS << endl;
    cout << "// population size: " << POPULATION_SIZE << endl;
    cout << "// min tree height: " << MIN_TREE_HEIGHT << endl;
    cout << "// max tree height: " << MAX_TREE_HEIGHT << endl;

    // A height of 7 generates a tree with 4096 leaf nodes (5461 nodes
    // total), which covers a 64x64 image.
    MyGPPool pool(POPULATION_SIZE, MIN_TREE_HEIGHT, MAX_TREE_HEIGHT);
    pool.init();
    pool.work(NUM_GENERATIONS);
    GP * best = pool.getBestTree();
    cout << "The best found is:" << endl << *best << endl;
    cout << "in generation: " << pool.getBestGeneration() << endl;
    cout << "with fitness is: " << pool.getBestFitness() << endl;

    // Write the data to an output file.
    Node::useNode_ = true;
    best->fitness();
    Node::useNode_ = false;
    outBmp->writeFile(outFrameFile);

    // Sort the blockList
    //qsort((void *) blockList, cBmp->getNumBlocks(), sizeof(blockList[0]),
    // sortFunc);

    // We will now loop through all of the blocks and save them to a BMP
    // file using a different color for each motion vector.

```



```

BitMap ooBmp(outFrameSrcFile);
ooBmp.clear();
Block *motionBlock;
unsigned c;
for (unsigned i=0 ; i<NUM_BLOCK_ELEMENTS ; i++)
{
    if (blockList[i].used==false)
        continue;
    motionBlock = new Block(blockList[i].width, blockList[i].height);
    c = ((blockList[i].xMotion & 0x0f) << 4)
        | (blockList[i].yMotion & 0x0f);
    motionBlock->fill((unsigned char)c);
    ooBmp.putBlock(motionBlock, blockList[i].x, blockList[i].y);
    delete motionBlock;
}
ooBmp.writeFile(outFrameFile2);

delete [] blockList;
blockList = 0;
delete currBmp;
delete prevBmp;
delete outBmp;
}

//////////////////////////////////////
////////// EXHAUSTIVE SEARCH CODE //////////
//////////////////////////////////////

//
// This method is for testing the exhaustive search method.
//
void exhaustiveSearch()
{
    const int VECTOR_RANGE = 16;
    const int MIN_VECTOR = (- VECTOR_RANGE / 2) + 1;
    const int MAX_VECTOR = VECTOR_RANGE / 2;
    const int BLOCK_SIZE = 8;

    MyBitMap *currBmp = new MyBitMap(currFrameFile, BLOCK_SIZE, BLOCK_SIZE);
    MyBitMap *prevBmp = new MyBitMap(prevFrameFile, BLOCK_SIZE, BLOCK_SIZE);
    MyBitMap *outBmp = new MyBitMap(outFrameSrcFile, BLOCK_SIZE,
                                    BLOCK_SIZE);

    Block *srcBlock;
    Block *destBlock;
    Block *copyBlock;
    int bestX;
    int bestY;
    double bestFitness;
    double fitness;
    unsigned blockCompares = 0;
    for (unsigned i=0 ; i<currBmp->getNumBlocks() ; i++)
    {
        cout << "(block " << i << ")";
        // Get the blocks to compare.
        srcBlock = currBmp->myGetBlock(i);

        bestFitness = 1.7e+308;
        for (int y=MIN_VECTOR ; y<MAX_VECTOR+1 ; y++)
        {
            for (int x=MIN_VECTOR ; x<MAX_VECTOR+1 ; x++)
            {
                destBlock = prevBmp->myGetBlock(i, x, y);
                // Compute the fitness of the vectors.
            }
        }
    }
}

```

```

        fitness = compareBlock(srcBlock, destBlock);
        blockCompares++;
        delete destBlock;
        if (fitness < bestFitness)
        {
            bestFitness = fitness;
            bestX = x;
            bestY = y;
        }
    }
}
delete srcBlock;

copyBlock = prevBmp->myGetBlock(i, bestX, bestY);
outBmp->myPutBlock(copyBlock, i);
delete copyBlock;

cout << "blk: " << i << ", "
     << " deltaX: " << bestX << ", deltaY: " << bestY
     << ", #compares: " << blockCompares << endl;

}
outBmp->writeFile(outFrameFile);

delete currBmp;
delete prevBmp;
delete outBmp;
}

//
// Main routine of the motion estimator. This is just a dispatcher
// which calls the appropriate method.
//
void main(int argc, char *argv[])
{
    char autoChoice = 0;
    if ( argc == 5 || argc == 6 )
    {
        autoChoice = argv[1][0];
        assert(autoChoice=='1' || autoChoice=='2' || autoChoice=='3' ||
              autoChoice=='4' || autoChoice=='5' || autoChoice=='6');

        currFrameFile = argv[2];
        prevFrameFile = argv[3];
        outFrameSrcFile = argv[3];
        outFrameFile = argv[4];
        if ( argc==6 )
            outFrameFile2 = argv[5];
    }
    else
    {
        currFrameFile = "test01.bmp";
        prevFrameFile = "test02.bmp";
        outFrameSrcFile = "test02.bmp";
        outFrameFile = "testout.bmp";
        outFrameFile2 = "testout2.bmp";
    }

    cout << "currFrameFile = " << currFrameFile << endl;
    cout << "prevFrameFile = " << prevFrameFile << endl;
    cout << "outFrameSrcFile = " << outFrameSrcFile << endl;
    cout << "outFrameFile = " << outFrameFile << endl;
    cout << "outFrameFile2 = " << outFrameFile2 << endl;

    char choice = 0;

```

```

while ( choice != '0')
{
    if ( autoChoice == 0 )
    {
        cout << endl
            << "Select the test to run:" << endl
            << " 1. Genetic Algorithm (fixed, iterative)" << endl
            << " 2. Genetic Algorithm (fixed, one shot)" << endl
            << " 3. Genetic Program (variable)" << endl
            << " 4. Genetic Algorithm (fixed, region)" << endl
            << " 5. Genetic Program (variable, region)" << endl
            << " 6. Exhaustive Search" << endl
            << " 0. Exit" << endl
            << "selection: " << flush;
        cin >> choice;
    }
    else
        choice = autoChoice;

    switch (choice)
    {
        case '1':
            cout << "Genetic Algorithm (fixed, iterative)" << endl;
            gaFixedIterative();
            break;
        case '2':
            cout << "Genetic Algorithm (fixed, one shot)" << endl;
            gaFixedOneShot();
            break;
        case '3':
            cout << "Genetic Program (variable)" << endl;
            gpVariable();
            break;
        case '4':
            cout << "Genetic Algorithm (fixed, region)" << endl;
            gaFixedRegion();
            break;
        case '5':
            cout << "Genetic Program (variable, region)" << endl;
            gpVariableRegion();
            break;
        case '6':
            cout << "Exhaustive Search" << endl;
            exhaustiveSearch();
            break;
        case '0':
            cout << "Exiting program..." << endl;
            break;
        default:
            cout << "Invalid selection, try again." << endl;
            break;
    }
    if ( autoChoice )
        choice = '0';
}

// eof

```