An Examination of Block Motion Compensation Algorithms for MPEG-2 and prediction of bit rates from video sequence measurements

J. J Francis
Supervisor Prof G de Jager

21 Frebruary 1997

[1]Submitted to the University of Cape Town in fulfilment of the requirements for the degree of Master of science in Engineering. Cape Town, February 1997

Declaration

I declare that this dissertation is my own work. It is being submitted for the Degree of Master of Science in Engineering at the University of Cape Town. It has not been submitted before for any degree or examination at this or any other university.

Jerome.J Francis
(Signature of Candidate)

Acknowledgements

I would like to thank my supervisor, Professor Gerhard de Jager for his guidance and enthusiasm.
I would also like to thank:

- My family, for their support and encouragement.
- The Foundation for Research Development (FRD), for their financial assistance.
- Orbicom, for providing the data needed for this thesis.
- Greg Cox, Robert Crida, Karen Henry, Fred Hoare, Jia Liu, Fred Nicolls, Giovanni Pagliari, Andre le Roux, and Brendt Wohlberg, who helped create a stimulating environment.

Abstract

This dissertation examines the following two problems:

- Finding a block motion compensation algorithm which is optimum in performance and speed.
- Predicting the performance, for complex sequences, of an MPEG-2 encoder.

An optimum motion compensation algorithm can lead to optimum temporal compression. For fixed bit-rate encoders finding methods to predict the bit-rate from properties of the video sequence can lead to an optimum use of the transmission bandwidth.

The examination of motion compensation algorithms involved examining previous algorithms. Historically, one of three functions are used to evaluate a candidate motion vector, namely, Mean Square Error (MSE), Minimum Absolute Difference (MAD) and cross-correlation. The ideal motion vector being the one that minimises MAD and MSE, and maximises cross-correlation. Sub-sampling, hierarchical and feature domain methods were examined. Finally some new algorithms are proposed and further areas of research suggested. The new algorithms suggested perform close to optimum, particularly those algorithms searching feature space.

The relationship between the candidate motion vector evaluation functions are examined and conditions under which the MAD, MSE and the cross-correlation function perform similarly are examined. The MSE is shown to perform as well as cross-correlation. Provided the number of differences are small, MAD and MSE perform similarly. MAD is computationally faster than MSE and is thus recommended.

The effect of quantisation on the motion compensation process is examined. It is shown that motion compensation after quantisation introduces the possibility of a better match compared to motion compensation before quantisation. Modifications to the MSE and MAD functions are proposed to make these functions' performance quantisation invariant.

The relationship between complexity and machine estimations of quality is examined. It is proposed that the bit-rate is a view of the complexity of a sequence from the perspective of a given encoder. The complexity estimation problem then reduces to a bit-rate estimation problem.

Statistical prediction, prediction from bit-rates and finally model based prediction is examined. Model based prediction is chosen because of simplicity and desired performance. Certain objective variables are measured from the sequences (such as the mean and variance of the motion vectors and Sobel filtered data) and these used to generate the model. Linear and quadratic models are examined for predicting bit-rates. The models are generated for two different image qualities, good and excellent image quality, with the bit-rate for excellent exceeding that for good image quality as expected.

The effect of Jackknifing and Principal Components Analysis on the model generation process is examined. The decorrelating effect of Principal Components Analysis allows the use of a simple linear model. The Jackknifing process allows the validation of the linear model using at each stage as much of the data as possible. Jackknifing readily provides variance estimates for each model coefficient which makes calculation of the confidence intervals easy.

The validated simple linear model has a 30% prediction error compared to 37% for the quadratic model for good image quality bit-rates. The validated quadratic model outperforms the simple linear model for excellent quality bit-rates, 40% prediction error compared to 108%. The Jackknifed linear model after Principal Components Analysis performs particularly well with a peak prediction error of 39% and 50% for the excellent and good image quality bit-rates. The expected average prediction error per observation on the Jackknifed model is 15.29% and 15.4% for excellent and good image quality bit-rates respectively.

The Jackknifed linear model after Principal Components Analysis outperforms the simple linear and quadratic models for predicting the excellent image quality bit-rates. Given model complexity and performance the simple linear model performs as well as the quadratic or Jackknifed linear model and is more robust for predicting the good image quality bit-rates.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Digital encoding of video sequences offers a number of advantages which can include security (encryption of data for specific end users), better utilisation of bandwidth (compression) and tolerance of noise. Analogue television systems are susceptible to noise in the data channel and phase errors resulting from more than one transmission path from transmitter to receiver (e.g., reflections of the signal off buildings).

Consider an arbitrary colour sequence (with separate red, green and blue data) of 480 vertical by 640 horizontal pixels and 30 frames per second digitised with a precision of 8 bits. The number of bytes (one byte is eight bits) needed to encode such a sequence is $640 \times 480 \times 30 \times 3 = 27.65$Mbytes per second. The sequence described is typical of the resolution frame rate and digitising precision of digital video [44]. If one considers the storage required for an hours video ($60 \times 60 \times 27.65$Mbytes per second $= 99\,530$Mbytes) or the transmission channel capacity (27.65Mbytes per second) the need for compression of digitally encoded video becomes apparent. Conventional terrestrial television using the PAL (Phase Alternate Line) has a bandwidth of 6 MHz [51] which is far less than the bandwidth of the digital channel. The practical implementation of digital television then requires compression.

When compressing video sequences the temporal redundancies within the sequence can be exploited to maximise compression. Within MPEG-1, MPEG-2 and H.261 the method of choice is block motion compensation as opposed to Pixel Recursion or Optical Flow. Even taking into account recent advances in hardware [57], there remains scope for optimising and/or finding algorithms which minimise algorithmic complexity and search time while maximising compression. Several algorithms will be examined together with the theoretical background to motion compensation in general and block motion compensation in particular. Finally some new algorithms are presented and their performance compared to some standard algorithms.

Video sequence complexity has bearing on the compressibility of a sequence. It seems reasonable to suppose that a sequence with complex motion and spatial properties will not compress as well as a sequence with simple motion and image content. A clear distinction must be made between what a human observer will regard as complex and what a compression

algorithm will have difficulty compressing. An attempt will be made to model the resultant peak bit-rates of several MPEG-2 compressed sequences, in terms of their motion and image characteristics. Knowledge of the peak bit-rate of a video sequence allows budgeting of channel capacity for fixed bit-rate encoders.

The layout of this thesis is as follows:

- Chapter 2 presents a brief overview of the MPEG-2 standard and places the issues of motion compensation and video complexity in perspective.
- Part I is an examination of motion compensation.
  - Chapter 3 presents an overview of the theory of motion.
  - Chapter 4 presents a literature overview of current block motion compensation algorithms.
  - Chapter 5 presents proposals for several new block motion compensation algorithms.
  - Chapter 6 summarises the application of the algorithms on the Windmill test video sequence.
  - Chapter 7 concludes Part I and summarises the application of various algorithms on the test data, ranks the algorithms in order of effectiveness, attempts to determine the properties of successful algorithms, and proposes future work.
  - Chapter 8 is an aside to Part I and examines the effect of quantisation on the motion compensation process and examines attempts to minimise the effect of quantisation on motion compensation.
  - Chapter 9 is an aside to Part I and examines the performance of functions evaluating the suitability of a trial motion vector.
- Part II examines the complexity of video sequences.
  - Chapter 10 presents an overview of Part II.
  - Chapter 11 presents a literature overview of video sequence quality and relates quality, complexity and bit-rate.
  - Chapter 12 presents an overview of the process of measuring complexity.
  - Chapter 13 presents an overview of the process of modelling the bit-rate.
  - Chapter 14 presents the results of the model fitting process.
  - Chapter 15 examines the effect of Principal Components analysis and Jackknifing on the linear model.
  - Chapter 16 concludes Part II by comparing the proposed linear and quadratic models. Some weaknesses of the model fitting process are examined as well.

- Appendix A contains an overview of Sobel filtering and Hashing.
- Appendix B contains the the results of the application of several motion compensation algorithms on several video sequences.
- Appendix C contains source code listings.

The problem of motion compensation and video complexity arises from the need to maximise and predict the compression of a given sequence. The issues affecting the compressibility of an MPEG-2 video sequence are examined in detail in Chapter 2.

Chapter 2

An overview of MPEG-2

MPEG-2 is one of a family of video compression standards. Previous standards in this family include H.261 and H.320, which are optimised for low bit-rate (or low channel capacity) teleconferencing applications; and MPEG-1 which is intended for multimedia and video storage applications. MPEG-2 is intended for digital television applications including High Definition Television (HDTV) as well as normal video. A basic overview of the video portion of MPEG-2 follows [25], and finally some areas of possible optimisation are given.

MPEG-2 is an example of transform encoders. The spatial information is decorrelated (on an 8x8 subblock level) using the Discrete Cosine Transform. The temporal redundancies (redundancies between frames) are removed using motion compensation.

This chapter describes begins by describing the basic structure of an MPEG-2 encoder. This basic structure has similarities with MPEG-1. The Profiles and Levels which provide an MPEG-2 bitstream with the capacity to service end users with different services, for example, end users with HDTV and normal video, are described. The final section describes areas of possible optimisation of MPEG-2.

2.1 The basic structure of an MPEG-2 bitstream

This section examines the basic structure of the MPEG-2 bitstream. The differences between the coding of spatial and temporal data is examined. The lossy effect of the coding process is described.

Each MPEG-2 object consists of instances of the object below in the following hierarchy (as described in Fig. 2.1):

- The sequence itself consists of Groups of Pictures (GOP).
- Groups of Pictures consists of Pictures (Frames).
- Pictures (Frames) consist of Slices.

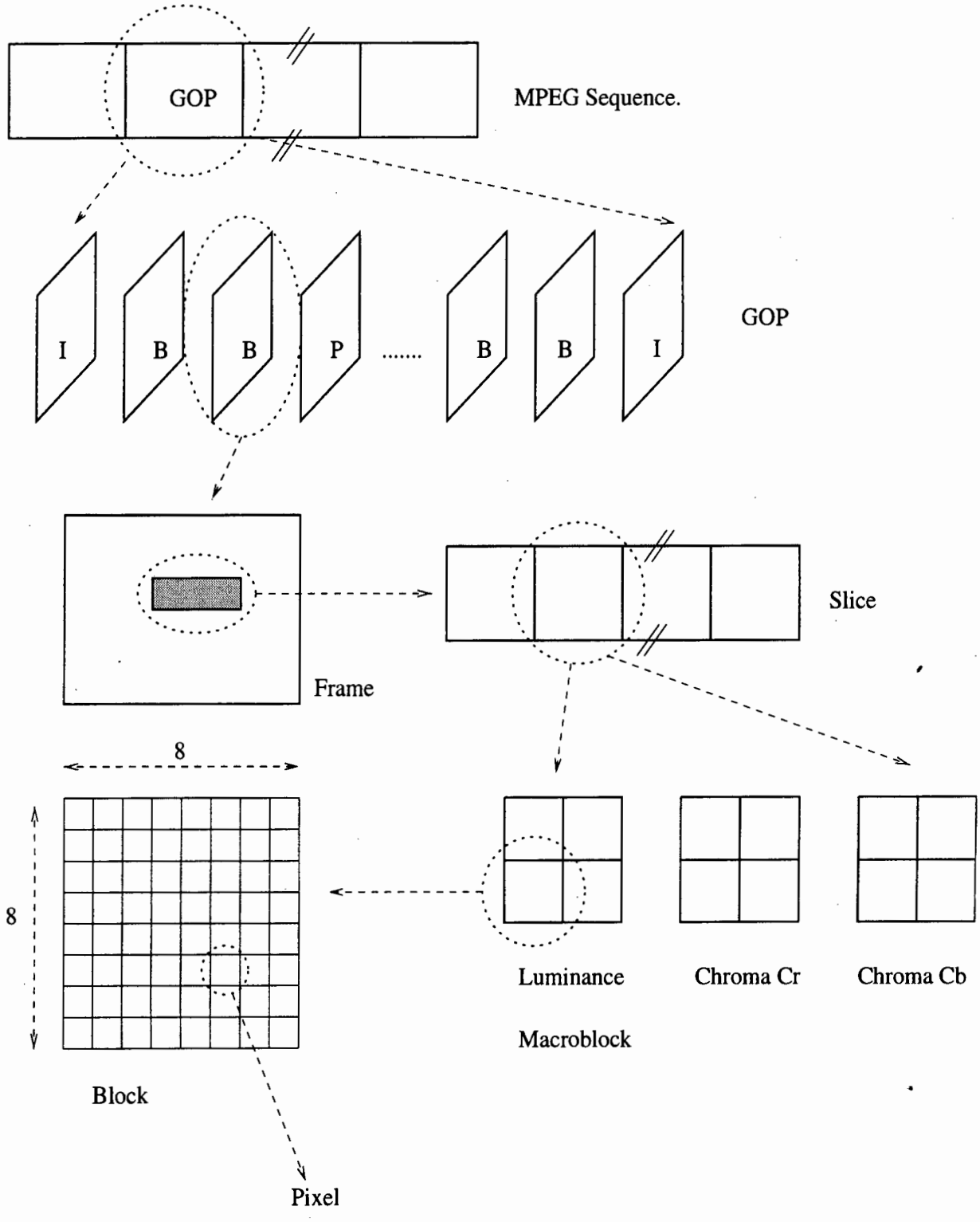


Figure 2.1: The MPEG-2 bitstream hierarchy.

- Slices consist of Macroblocks.
- Macroblocks consist of four luminance blocks and up to four blocks for each of the Cr and Cb colour components. The RGB space of the input video is transformed into the YCrCb colour space [25].
- Blocks are a array of 8x8 pixels.

The GOP consists of a series of I, P and B frames. Note that the motion compensation is performed on the Macroblocks.

Intra coded frames (I) No temporal relationship with any other frame.

Predicted Coded Frame (P) Motion compensation is performed on the Frame with the previous I or P frame used as the reference frame.

Bi-directionally Predicted Frames (B) Motion compensation is performed on the frame with both or one of the closest I or P frames used as the reference frames. If one frame is used then the reference frame is a past or future I/P frame and motion compensation proceeds as for the P frame case. If both reference frames are used then two motion vectors are generated and the best match block coded is the average of the best match blocks on the reference frames.

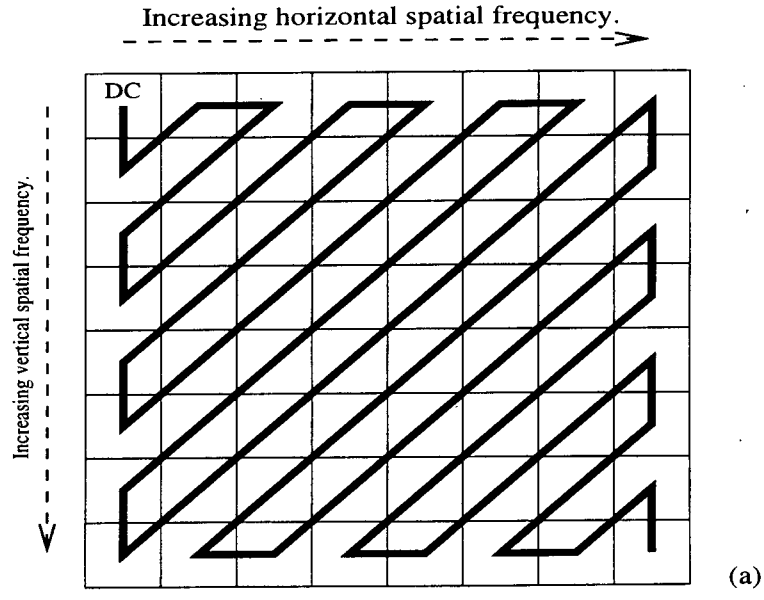
Intra coded blocks are coded as follows:

- Each block is Discrete Cosine Transformed (DCT).
- Each block is quantised. The Quantisation process first offsets by half the divisor and then divides each DCT coefficient by a frequency dependent scale factor.
- The block is then zig-zag encoded (starting at the DC coefficient) to form a one dimensional signal (Fig. 2.2 shows the two zig-zag patterns used; Fig. 2.2(b) is recommended for interlaced video). Run-length encoding then codes each non-zero DCT coefficient in terms of the coefficient itself, and the number of zero DCT coefficients before the non-zero coefficient on the zig-zag path.
- The one dimensional signal is Huffman Coded. Note the DC coefficient is coded differently; it is first scaled and then coded by a different Huffman Code Table.

Three types of motion compensated Macroblocks occur:

Forward predicted P coded.

Backward Predicted The reference Macroblock lies on a future frame.



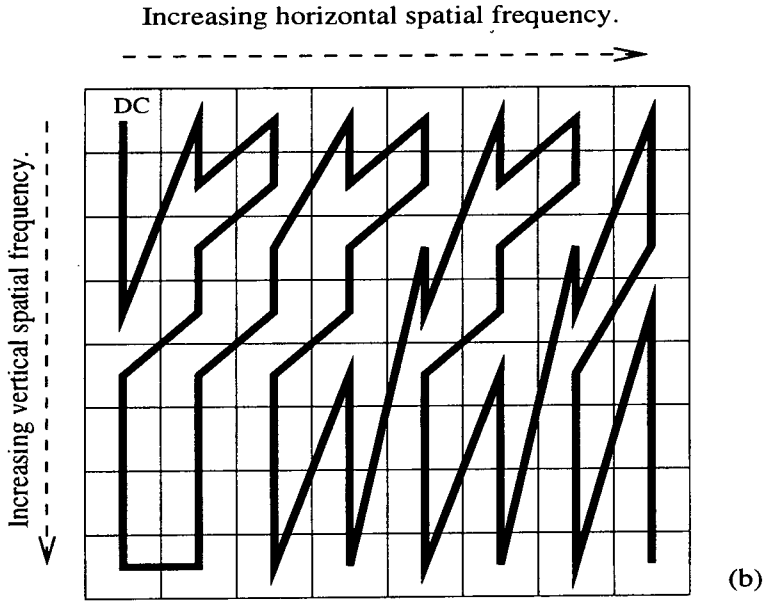


Figure 2.2: Zig-Zag encoding patterns.

8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

Table 2.1: Default Quantisation matrix for Intra blocks.

Average Bidirectionally coded Both motion vectors for the forward and backward predicted Macroblocks are encoded. The difference Macroblock is the difference between the search Macroblock and the average of the best fit Macroblocks on the future and past reference Macroblocks.

The motion compensated Macroblocks are coded the same as Intra coded blocks with the following exception: the Macroblock coded is the difference of the search Macroblock and the best fit Macroblock. The motion vector(s) in each Macroblock is coded differentially with respect to the motion information of the previous Macroblock in the Slice. Note that performing motion compensation on 16x8 pixel Macroblocks compensates for interlacing.

The quantisation matrices can be changed from the default values at the start of each sequence. Within each Macroblock the quantisation matrix can be scaled to provide coarser or finer quantisation as required. Intra and non-Intra blocks are quantised differently. Non-Intra blocks are motion blocks and since the Human Visual System cannot perceive high frequency structure (fine structure) together with motion (i.e., blurring seen in moving objects or motion blur); they can be more severely quantised. Since perception of quantisation errors in high spatial frequencies is lower than for low frequencies the high frequencies are more coarsely quantised (see Table 2.1 for the default quantisation matrix for Intra coded blocks).

The process of quantisation is as follows:

$$F_q(u,v) = floor(\frac{F(u,v) + \frac{Q(u,v)}{2}}{Q(u,v)})$$

where $floor(x)$ is x rounded down, $F(u,v)$ is the Discrete Cosine Transform of the block, and $Q(u,v)$ contains the quantisation coefficients.

The DC coefficient is separately coded and in differentially coded with respect to the previous luminance block for luminance blocks and the previous colour block for colour blocks in the Macroblock. Macroblocks are described as the ratio of luminance blocks to the red to the blue blocks, for example, a 4:4:4 Macroblock contains four luminance, four Cr blocks and four Cb blocks [25].

Level	Columns	Rows	Frame Rate	Max Sampling Rate (x10^6)	No. Layers
High	1920	1152	60	132.71	3
High-1440	1440	1152	60	99.53	3
Main	720	576	30	12.44	2 (No SNR)
Low	352	288	30	3.04	2 (No SNR)

Table 2.2: Comparison of Level Frame rates and allowed layers.

2.2 Profiles and Levels

MPEG-2 differs from MPEG-1 in that the syntax is divided into profiles and the profiles divided into levels (see Tables 2.2 and 2.3 for a comparison of different levels and profiles). The levels provide a defined constraint on the parameter values (e.g., a layer may set a maximum frame size). Profiles can be combined into hybrid scalable sequences with a base layer and up to two extra enhancement levels. Note that any profile or level will decode the one below it, e.g., Temporal Scalable will decode SNR and Spatial Scalable and High-1440 will decode Main and Low.

The uses of the profiles are as follows:

Spatial Scalable: Intended to cater for services where High Definition TV (HDTV) and ordinary TV co-exist.

SNR Scalable: Used where two layers of different video quality are needed. Similar to Spatial Scalable but with the following exception: this involves generating two layers of the same spatial resolution but different qualities. The base layer provides the basic quality and the enhancement layer the rest.

Temporal Scalable: Used for migration to higher temporal resolution (frame rate) systems. Again two layers are encoded with the base layer carrying the more important information. The enhancement layer is temporally predicted from the base layer.

Data Scalable: Used when two different channels are available. The bit-stream is partitioned such that the more important data (headers, motion vectors, DC coefficients) are transmitted over the channel with the better error performance and the rest of the bit-stream over the other channel.

2.3 Optimisation of MPEG-2

The MPEG-2 standard does *not* define methods for finding optimum motion vectors or optimum placement of frames but only specifies *how* these are to be coded. It is clear that in the

Profile	Comparison
Simple	No B-frames
Main	Not picture, sequence or temporal scalable
SNR	SNR scalable only
Spatial	SNR or Spatial scalable
High	4:2:0 or 4:2:2 chroma

Table 2.3: Comparison of different MPEG-2 Profiles.

areas where the implementation of the encoder has some discretion in the structuring of the bitstream, room for optimisation exist.

The areas of optimisation of MPEG-2 base layer bit-rate include the following [30]:

- Optimisation of the search for the best match forward and backward prediction motion vectors.
- Optimisation of the search for the best fit average coded (B coded) motion vectors.
- The effect of quantisation on the motion vector searches.
- Prediction of bit-rates from the sequence data for fixed bit-rate encoders.
- Prediction of the optimum placement of I, P and B frames in the GOP.

The issues involved in optimising the search for best match forward and backward prediction motion vectors are examined in detail in Part I. The issues involved in predicting bit-rates (and hence complexity of the video sequence) is examined in Part II.

Part I

Investigation into performance of Block Motion Estimation Algorithms

Chapter 3

A theory of motion

This chapter will examine some theory behind motion in general, block motion compensation in particular and pel recursive techniques.

Motion estimation is complicated by the aperture problem [26], which is really a special case of the correspondence problem. The correspondence problem is the difficulty in classifying a pixel in one frame as belonging to a particular object in another frame. The aperture problem is illustrated in Fig. 3.1. One cannot determine the difference in Fig. 3.1(a) between the stationary bar and one moving downward, if all that is visible is the portion of the bar within the ellipsoid (aperture). In Fig. 3.1(b) the end point of the bar gives sufficient information to find the motion of the bar. Note that where the aperture problem exists one can at best only find the motion normal to the edge(s) of the bar/object. To lessen the aperture problem one needs the edges of an object to be at some angle to each other, then the normal directions are at an angle to each other. A possible solution satisfying this is more difficult (for some blocks impossible) to find except at corners where the edges meet. This illustrates why corners are immune to the correspondence problem.

3.1 Pel recursive methods

Pel recursive methods were first described by Netravalli and Robbins [41]. These attempt to minimise some distortion function on a neighbourhood around a pixel. Given some original estimate of the motion of the pixel an updated estimate is found which reduces the distortion further. These algorithms are also known as *Recursive Displacement Estimation Algorithms* [42], [41]. These are essentially gradient descent algorithms (gradient descent algorithms are used by optical flow algorithms [14]). There is a good chance of finding a local minimum using these methods, or rather, not finding the actual displacements. The displacement is the vector from the start position in the previous frame to the current position in the current frame for a moving pixel.

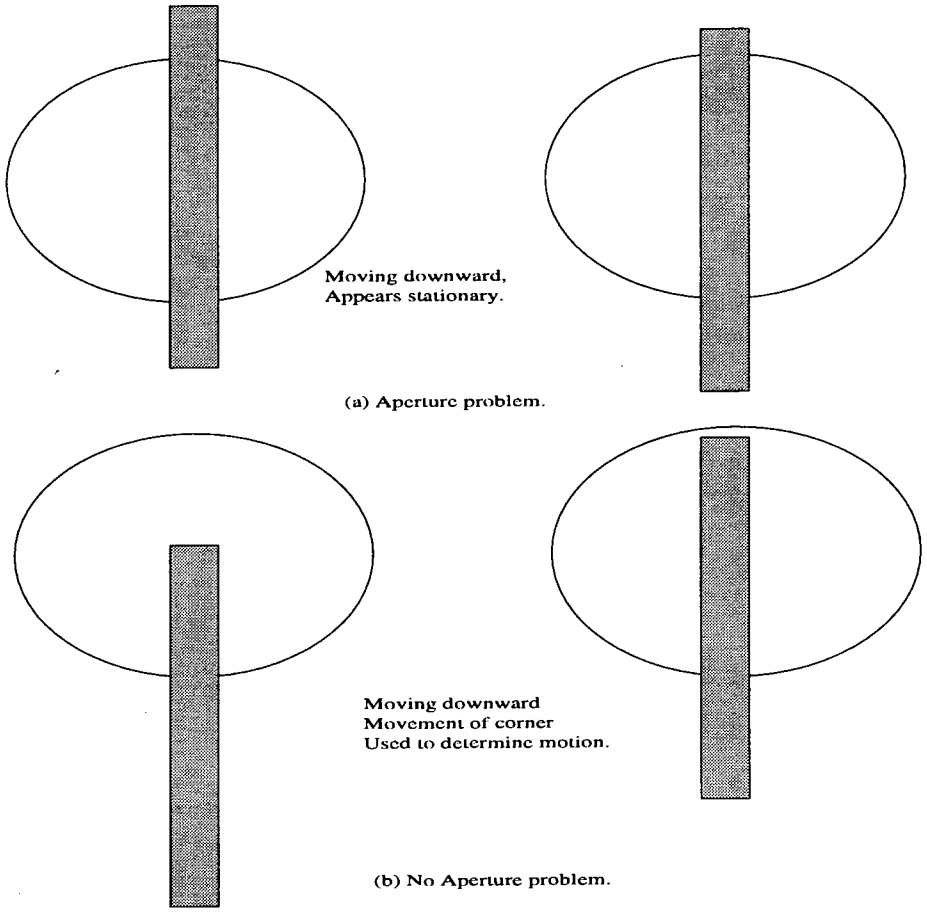


Figure 3.1: An illustration of the aperture problem.

Algorithm	$i-$ th Update term	
Bergmann	$-\frac{\frac{\partial}{\partial x}R_{f_k f_{k-1}}(x,y,\hat{mv}_i)}{\frac{1}{2}\left[\frac{\partial^2}{\partial x^2}R_{f_k f_{k-1}}(x,y,\hat{mv}_i)+\frac{\partial^2}{\partial x^2}R_{f_k f_k}(x,y,0)\right]}$	
Cafferio and Rocca	$\frac{\frac{\partial}{\partial x}R_{f_k f_{k-1}}(x,y,\hat{mv}_i)}{\lvert\frac{\partial^2}{\partial x^2}R_{f_k f_k}(x,y,0)\rvert+\eta^2}$	and $\eta^2 = 100$
Netravali and Robbins	$\epsilon\frac{\partial}{\partial x}R_{f_k f_{k-1}}(x,y,\hat{mv}_i)$	and $\epsilon = 1/1204$
Newton – Rapson	$-\frac{\frac{\partial}{\partial x}R_{f_k f_{k-1}}(x,y,\hat{mv}_i)}{\frac{\partial^2}{\partial x^2}R_{f_k f_{k-1}}(x,y,\hat{mv}_i)}$	
Where	f_j	Frame no j
	$R_{f_k f_{k-1}}(x,y,\hat{mv}_i)$	Cross – correlation of f_k and f_{k-1}
	$\hat{mv}_i$	Estimated displacement

Table 3.1: Update terms for various Pel Recursive Algorithms. [42]

The Discrete Frame Difference is defined as follows:

$$DFD = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} search\_block(i,j) - best\_match(i,j)$$

Pel recursive techniques minimise the Discrete Frame Difference (DFD), usually on a pixel by pixel basis.

An general form of these algorithms are:

$$U_{i+1} = U_i + \delta$$

Where δ is the update term (predicted motion) for each pixel, and U_i is the current estimate of the best match motion vector. Several proposals have been made for the update term (see Table 3.1). The update terms have been given in terms of the derivative of the cross-correlation function with respect to x, but this is readily extensible to the multi-dimensional case.

3.2 Block Motion Compensation

Block matching algorithms divide the current frame into a series of non-overlapping blocks (the search blocks) and attempts to find the best match (via some criterion) in some search region in the previous frame [41], [42]. The reference frame is divided into overlapping blocks (reference blocks) and one of these will be the bestmatch block. The final displacement is the vector from the position of the search block to the position of the best match block, where both are considered to lie on the same two dimensional image plane.

Some similarities between block motion compensation and pel recursive algorithms exist, e.g., the block motion algorithms Three step Search and Block Based Gradient Descent Search Algorithm are really gradient descent algorithms. Block Matching Algorithms usually rely on the final displacement being close to the position of the search block. Since most motion vectors are short this is a valid assumption [45].

Three common criteria are used for matching, namely, minimising the *Mean Square Error(MSE)*, minimising the *Mean Absolute Difference(MAD)* and finding the peak on the correlation surface. The definitions of the criteria are:

$$MSE(i,j) = \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N}[f_k(m,n) - f_{k-1}(m+i,n+j)]^2 \tag{3.1}$$

and

$$MAD(i,j) = \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N} \mid f_k(m,n) - f_{k-1}(m+i,n+j) \mid \tag{3.2}$$

and

$$correlation(i,j) = \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N} f_k(m,n) f_{k-1}(m+i,n+j) \tag{3.3}$$

where

- M, N are the dimensions of the block, usually $M = N = 8$.
- f_k current frame.
- f_{k-1} previous frame.

Note that *MAD* is easier to calculate than *MSE* since the sign test and inversion for a difference value can be performed faster than a multiplication. *MSE* is related to *correlation* (see Chapter 9).

Netravalli and Robbins [42] propose defining the Discrete Frame Difference (DFD) as a measure of the error of the motion compensation algorithm. The DFD is the sum of the differences of the pixels of the best match and the search blocks:

$$DFD = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} search\_block(i,j) - best\_match(i,j)$$

A difficulty with this definition is that the positive and negative differences partially cancel. A more realistic definition of the error involves the absolute values of the differences which weights the error per pixel uniformly and involves all pixels. The definition of DFD used in this thesis is:

$$DFD = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \mid search\_block(i,j) - best\_match(i,j) \mid$$

Block based motion compensation (BBMC) does not yield exact motion (correct motion), motion vectors since it does not attempt to deal with the aperture problem but only minimises the DFD and the length of the motion vector. BBMC algorithms tend to have small block sizes (usually 8x8, 16x8 or 16x16 for MPEG), and are especially susceptible to the aperture problem. Some motion vectors will be valid estimators of the motion field but only for blocks which do *not* suffer from the aperture problem. A further complication is that certain algorithms (notably Three Step Search and Two Dimensional Logarithmic Search) require a monotonically decreasing distortion error as one approaches the position of exact match (where one exists) from the start position. This is problematic and the search is likely to converge to a local minimum instead.

Other BBMC algorithms use full search or an optimised version of full search. Some practical implementations of BBMC algorithms are examined in Chapter 4.

Chapter 4

Overview of current block-matching algorithms

Several current algorithms are described in this chapter. The algorithms presented cover a cross-section of approaches from ad-hoc (e.g., Parallel Hierarchical One Dimensional Search) to systematic (e.g., Successive Elimination Algorithm). The algorithms can be divided into two basic types: those that examine the entire reference frame and those that examine only a subset of the reference frame. The algorithms are quantatively compared in the results chapter (Chapter 7).

Block matching algorithms [42], [41] attempt to minimise the distortion (MSE, MAD, correlation, etc.) between the search block on the reference frame and some/all possible blocks on the motion frame. This implies keeping the pixel values of the difference block (the difference of the search block and the final best block chosen) as small as possible. The motion vector is the displacement from the search block's location to the location of the matching block. Note that the algorithms described use 8x8 blocks, although they can be readily generalised to any block size.

4.1 A Block Based Gradient Descent Search Algorithm

This search (proposed by Liu and Feig [36]) depends on the assumption that the distortion monotonically decreases as one moves away from the centre of the search block. At each step nine points in a 3x3 checking block (see Fig. 4.1 for an example of a search) are examined. Note that for each successive step most points in the checking block will have been examined in the previous pass.

1. The search is initialised at the centre of the search window (usually 14x14 pixels).

2. The distortion function is evaluated at all nine points.

3. If the centre point of the checking block has the smallest distortion, then the search ends with the motion vector pointing to the centre pixel.

4. If the centre point is not the best, then centre the checking block on the point with the best distortion and repeat 3.

5. If the search reaches the boundary of the search window, then the search ends with the motion vector pointing to the point with best distortion in the checking block.

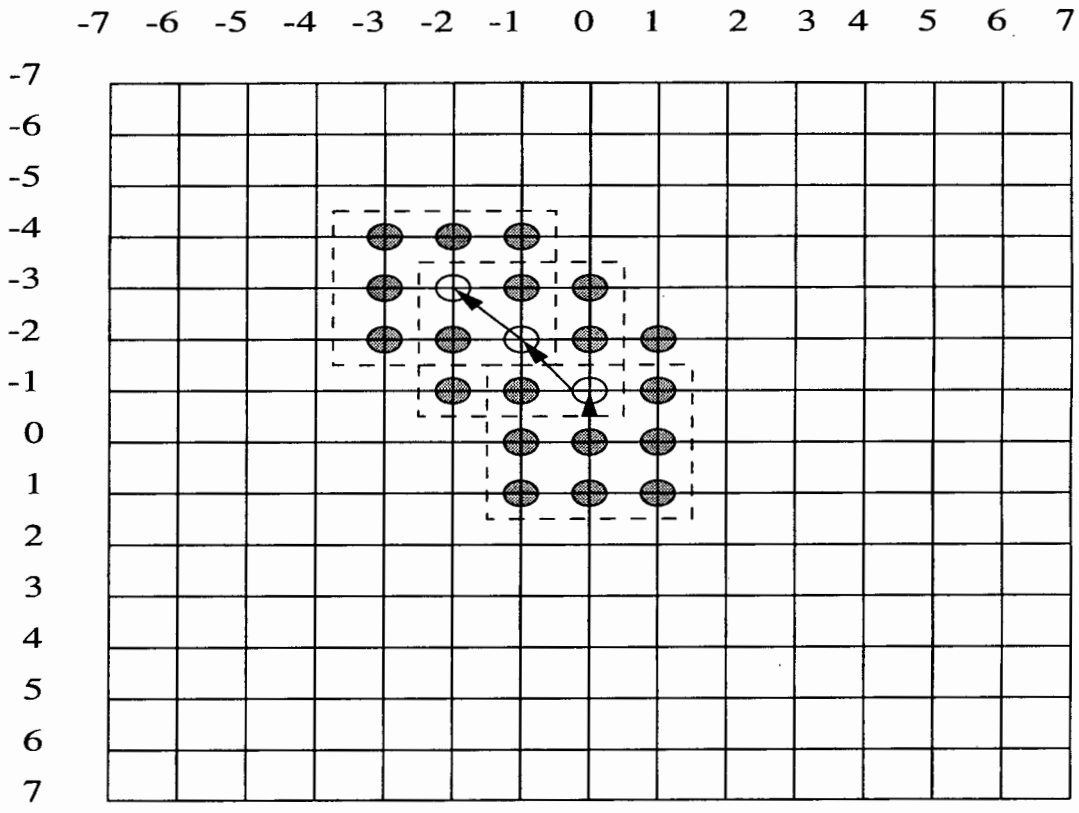


Figure 4.1: Example search points for the Block based Gradient Descent Algorithm.

4.2 The Conjugate Direction Search

This search (proposed by Srinivasan and Rao [50]) is a modification of the Conjugate Search (described in Section 4.3).

The search proceeds as follows (see Fig. 4.2 for an example search):

Step 1: Search the centre of the search block (0, 0) and one adjoining point to the left (-1, 0) and to the right (1, 0). If (-1, 0) or (1, 0) is a minimum repeat the search centred on

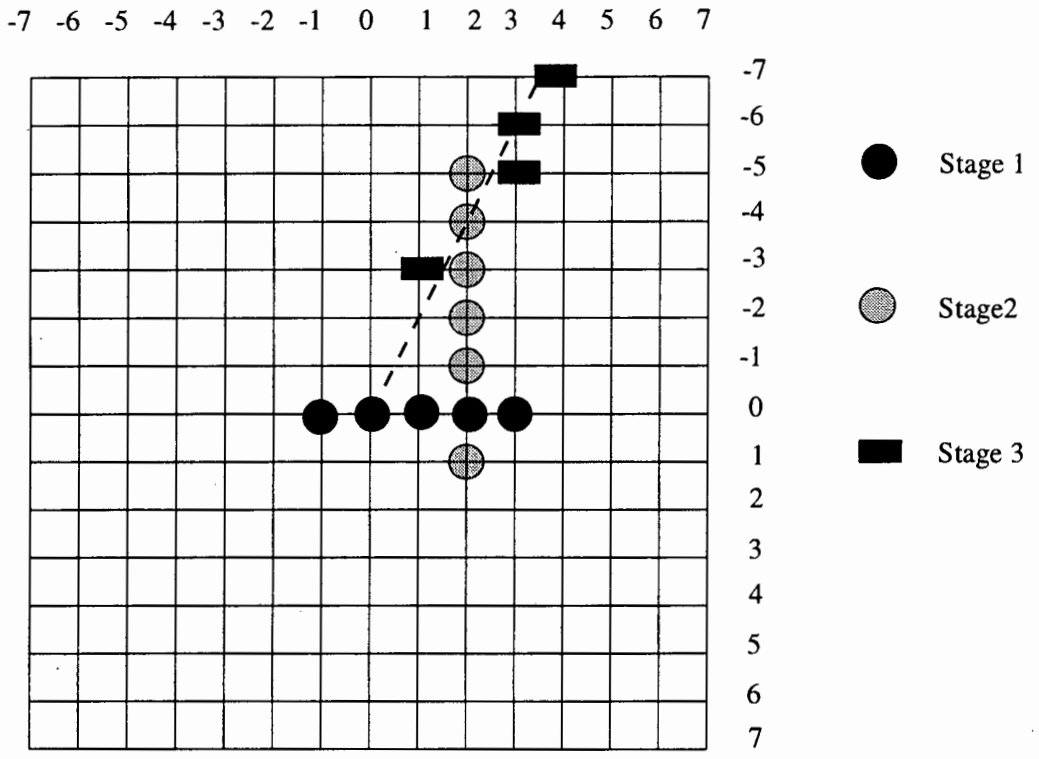


Figure 4.2: Example search points for the Conjugate Direction Search.

the minimum until either the boundaries of the search-block are reached or a minimum is found which lies between two larger values.

Step 2: Repeat the search (as in Step 1) in the Y-direction centred on the best match found in Step 1..

Step 3: Repeat the search (as in Step 1) along a vector from the origin of the search-block to the minimum as computed in Step 2. The search is repeated along this vector centred on the minimum found in Step 2. It should be noted that pixelation effects (i.e., attempting to represent a line by its pixel samples) leads to the final search line meandering along the search line.

4.3 The Conjugate Search (One-at-a-time-Search)

The first stage of the search (described by Musmann et al. [41]) (see Fig. 4.3 for an example search) involves searching the centre and the two points next to the centre (see Fig. 4.3) as in [50], [9] and [41]. If the smallest distortion (MAD or MSE) is one of the outer points, one continues the search horizontally in the direction of the lowest distortion until a point is found where the next point has a higher distortion, or the end of the search region is reached. The

next stage is the same as the previous stage except the search is vertical. A point above and a point below the new point is searched and so on. The displacement is the difference between the origin of the search area and the point of minimum distortion in the vertical direction.

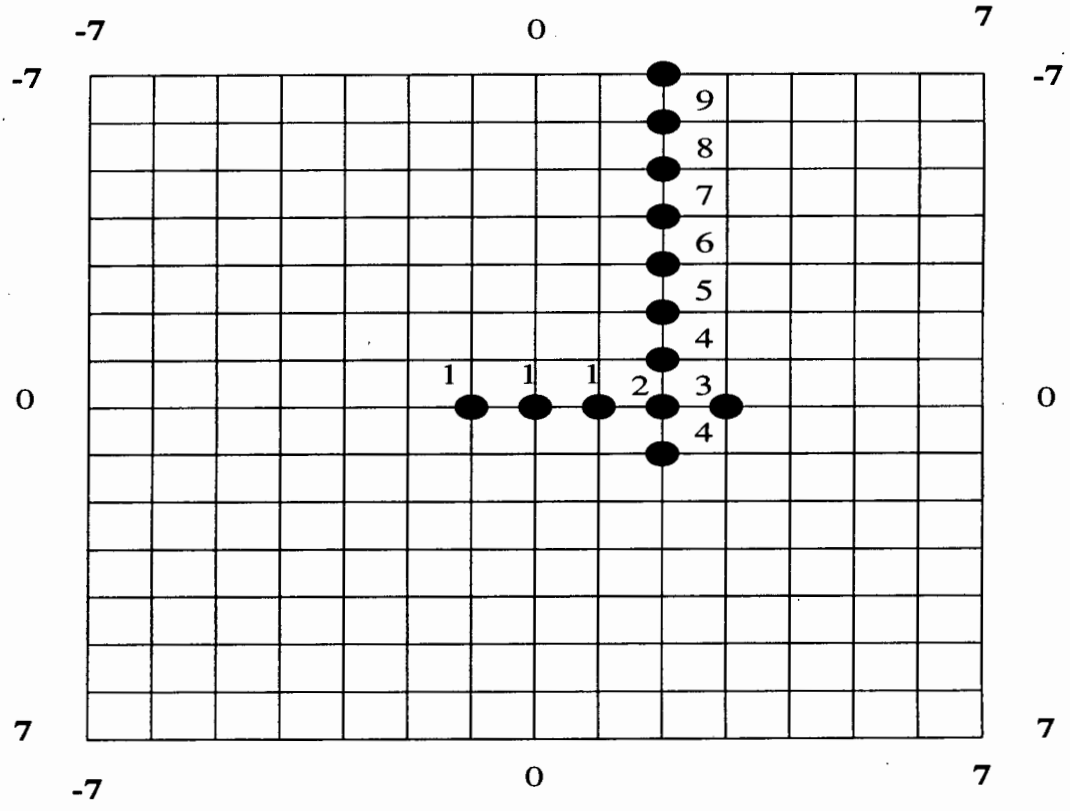


Figure 4.3: Example search points for the Conjugate Search.

4.4 The Cross Pattern Search (CPS)

This search (proposed by Fok et al. [2]) is similar to the Plus Pattern Search (Section 4.4) with the exception that the distortion measure used is:

$$MAD_{CPS} = \sum_{n=0}^{H} | f_k(x+n, y+n) - f_{k-1}(i+n, j+n) | + | f_k(x+H-n, y+n) - f_{k-1}(i+H-n, j+n) | \quad (4.1)$$

as illustrated in Fig. 4.4. Here $H = N - 1$.

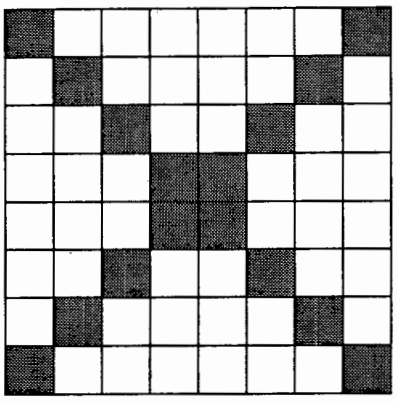


Figure 4.4: The feature for the Cross Pattern Search.

4.5 The Cross Search Algorithm

This is similar to the logarithmic search (described in Section 4.19), except only four points are searched at each stage and at the final stage either a St. Andrew's cross (X) or a normal (Greek) cross (+) is searched.

The algorithm (proposed by Ghanbari [19], illustrated in Fig. 4.5) is as follows:

1. If the distortion at the centre of the block is less than some threshold then the block is classified as unmoving and the motion vector is (0, 0).

2. Initialise the minimum position (m, n) to (0, 0) and set the step size to p = $\frac{w}{2}$; w is the maximum motion displacement, i.e., half search area width (or height).

3. Note the co-ordinates of the minimum position (m, n) by making (i, j) = (m, n).

4. Find the point of minimum distortion (m, n) of the following points: (i, j), (i - p, j - p), (i - p, j + p), (i + p, j - p), (i + p, j + p).

5. If p = 1 (end stage) go to step 6, otherwise halve p and go to step 3.

6. If the final minimum position is either (i, j), (i - 1, j - 1) or (i + 1, j + 1) go to step 7, otherwise step 8.

7. Search for the minimum position at (m, n), (m - 1, n), (m, n - 1), (m + 1, n) and (m, n + 1). A Greek Cross (+) is searched.

8. Search for the minimum position at (m, n), (m - 1, n - 1), (m - 1, n + 1), (m + 1, n - 1) and (m + 1, n + 1). The points on a St. Andrews Cross (X) is searched.

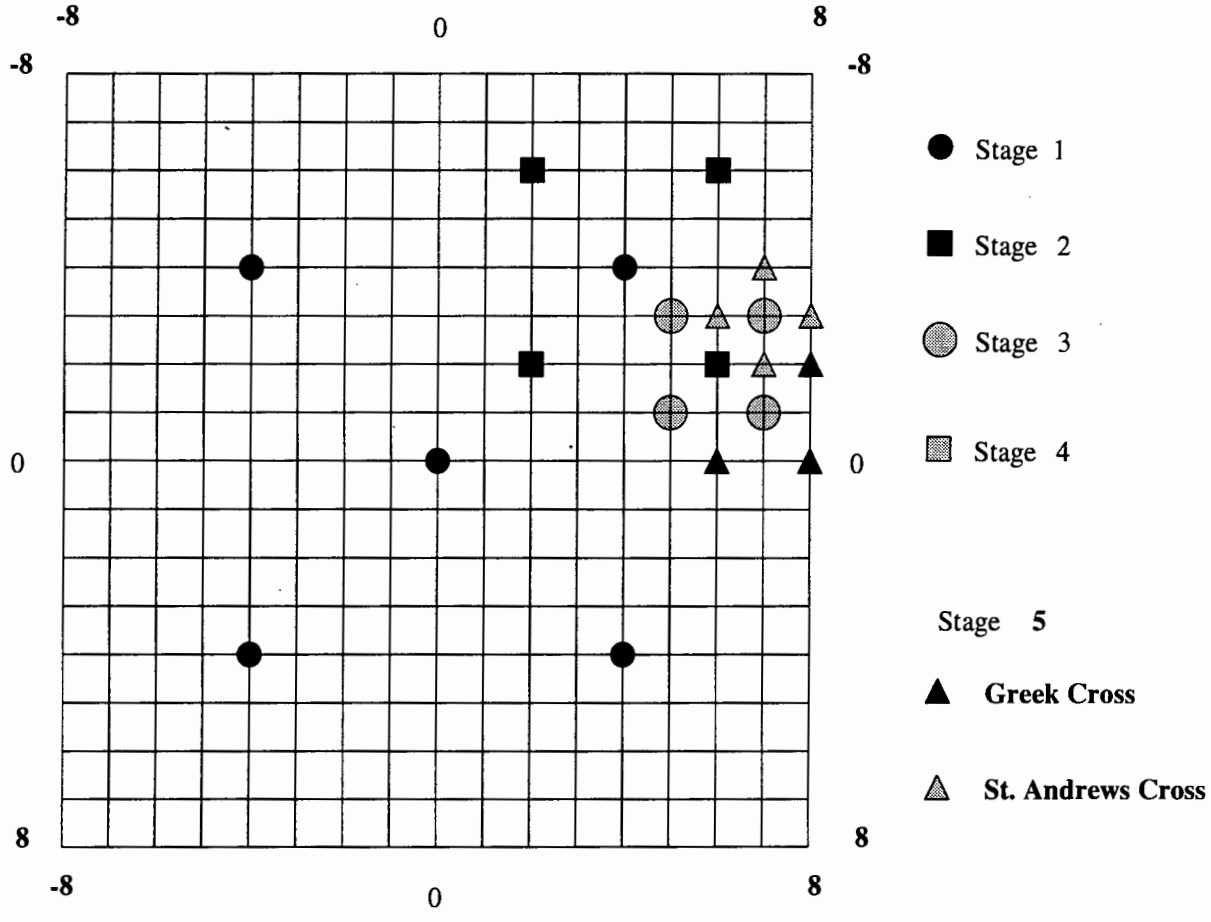


Figure 4.5: Example search points for Cross Search.

4.6 Dynamic Search-Window Adjustment and Interlaced Search Block Matching Algorithm

This search (proposed by Lee et al. [33]) adjusts the size of the search window for the next step based on both the goodness of the match and the second best match. The algorithm is essentially a gradient descent algorithm with varying convergence.

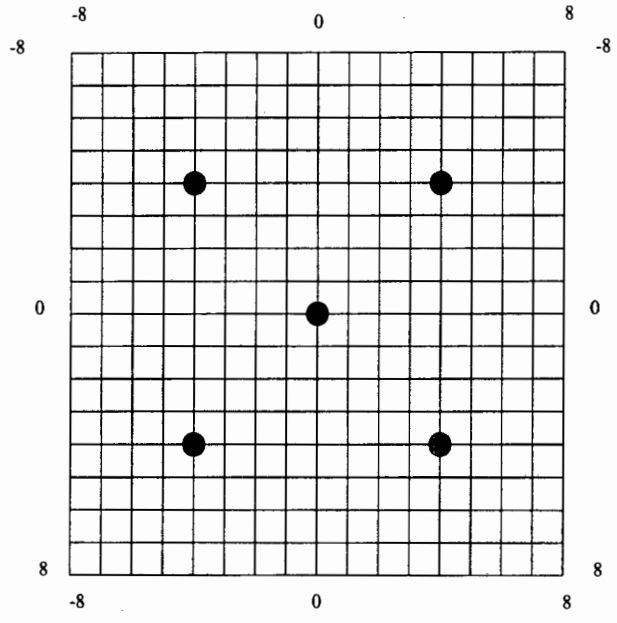


Figure 4.6: Example search points (X) for Dynamic Search-Window Adjustment and Interlaced Search Block Matching Algorithm.

The search is as follows for an initial 17x17 search window:

1. Search positions in the form of an X (see Fig. 4.6).
2. Adjust the search window to the proper size.
3. Search positions in the form of a + (see Fig. 4.7).
4. Adjust the search window to the proper size.
5. Return to Step 1 until only one point is left in the search window. Perform a full search of the nine points around the final point.

The adjustment to the window size proceeds as follows:

$$G = \frac{min2(MAD(P)) - min(MAD(P))}{min2(MAD(P))} \tag{4.2}$$

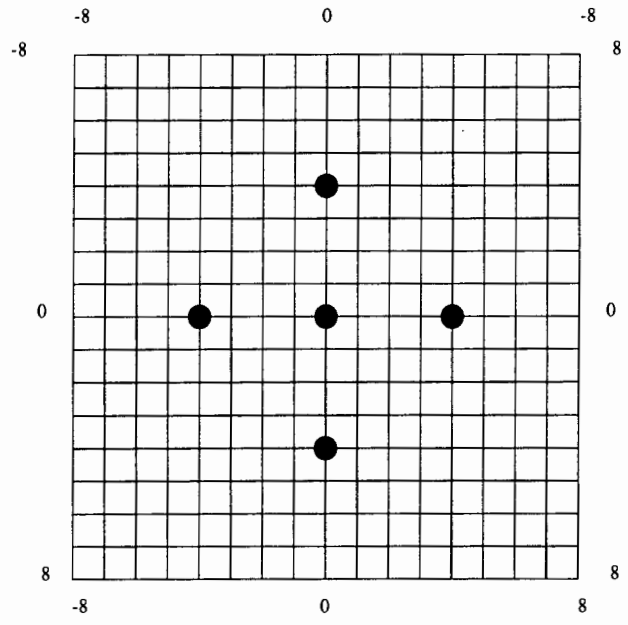


Figure 4.7: Example search points (+) for Dynamic Search-Window Adjustment and Interlaced Search Block Matching Algorithm.

where $min()$ and $min2()$ returns the smallest and second smallest values.

The window convergence is modified as follows:

- Fast mode: $R_w = \frac{1}{4} \times \frac{1}{4}$ if $G > T_H$
- Normal mode: $R_w = \frac{1}{2} \times \frac{1}{2}$ if $T_L < G < T_H$
- Slow mode: $R_w = \frac{3}{4} \times \frac{3}{4}$ if $G < T_L$,

where R_w denotes the dimensions of the search window with respect to the dimensions in the previous stage. The initial search window dimensions are 17x17.

Note that Lee et al. [33], propose using $T_H = 0.6$ and $T_L = 0.3$.

4.7 A Novel Four-Step Search Algorithm for fast Block Motion Estimation

This search (as proposed by Po and Ma [45]) relies on a centre biased motion vector distribution (an example search is shown in Fig. 4.8). The algorithm is as follows:

Step 1: A minimum is found from nine points on a 5x5 pattern as in Fig. 4.9(a). If the minimum is at the centre of the search pattern go to Step 4.

Step 2: The search window size is maintained as 5x5, but the pattern depends on the location of the minimum.

- If the minimum is at the corner of the search window five new positions are searched as in Fig. 4.9(b).
- If the minimum is located in the middle of the horizontal or vertical axes, three more points are searched as in Fig. 4.9(c).
- If the minimum is located at the centre of the search window go to Step 4 otherwise go to Step 3.

Step 3: The searching strategy is the same as Step 2.

Step 4: The search window is reduced to 3x3 as shown in Fig. 4.9(d). The resultant motion vector is regarded as the displacement from the origin of the search block to the minimum of these nine points.

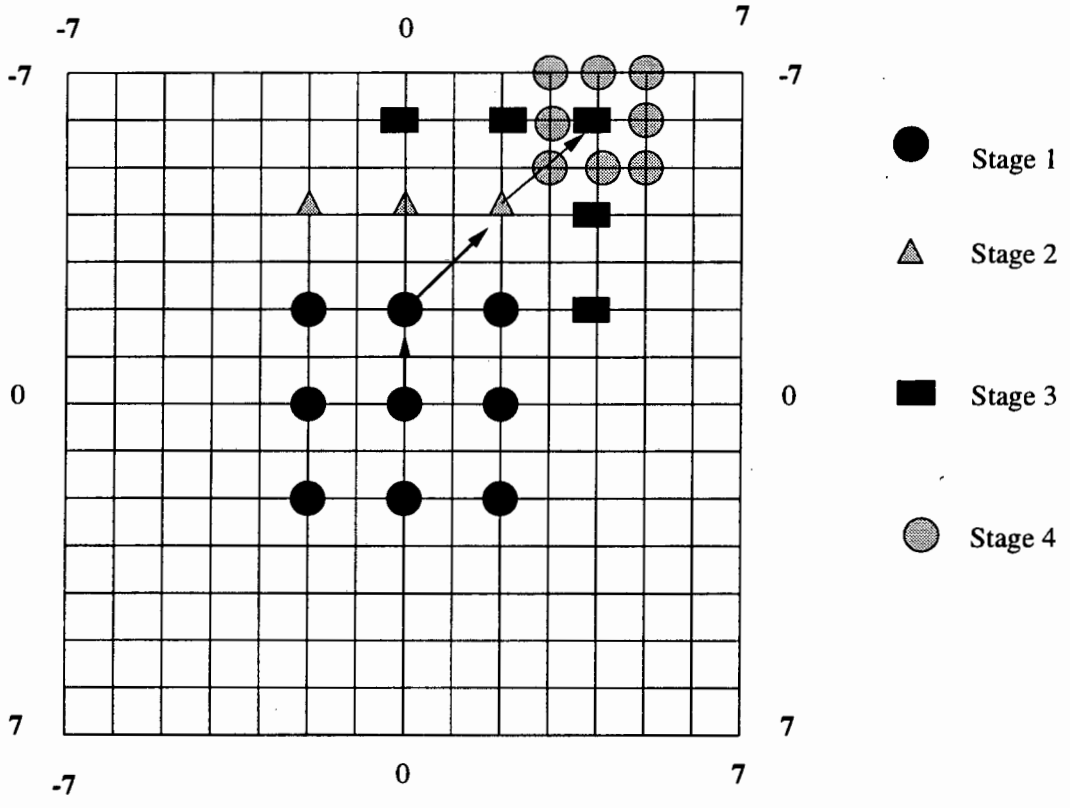


Figure 4.8: Example search points for the Four-Step Search.

4.8 The Full Search algorithm

This is an exhaustive search algorithm [9]. Every point in the search area is examined for the minimum distortion point. A variant which has the search area smaller than the frame/field

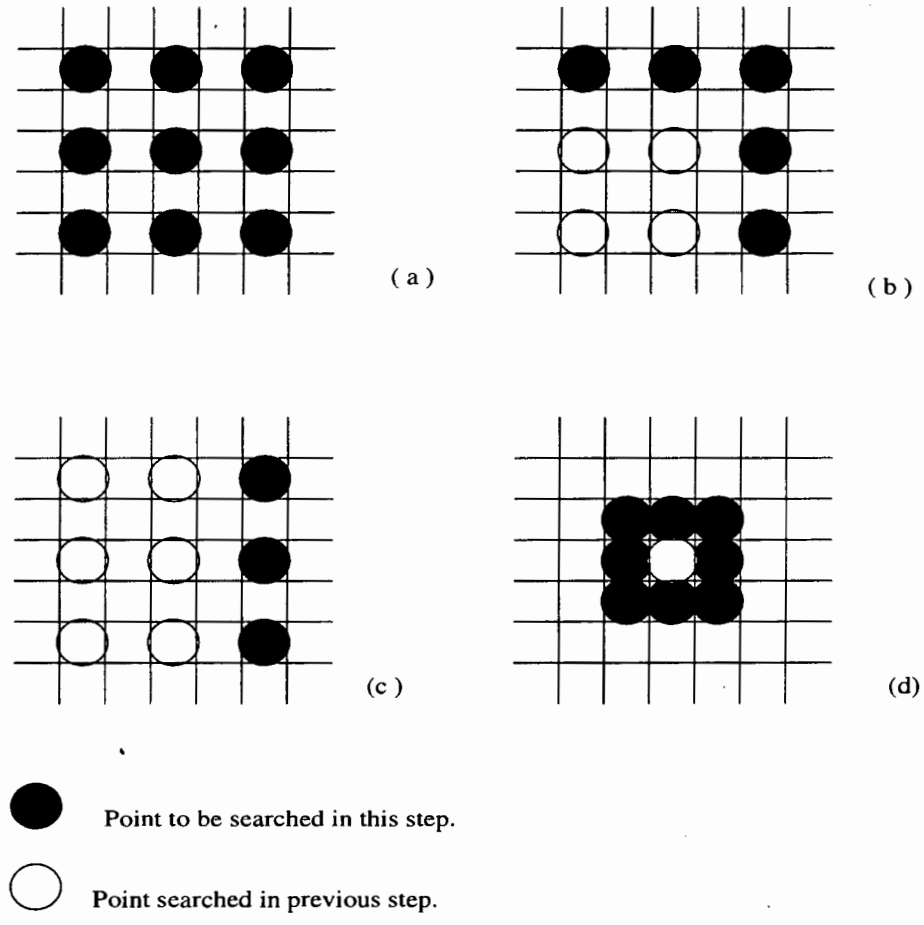


Figure 4.9: Example search patterns for the Four-Step Search. (a) First Step, (b) second, third step, (c) second, third step, and (d) fourth and final step.

should ignore points outside the frame for points on the border of the frame. If MAD is used as the distortion measure it can be sped up as in the Successive Elimination Algorithm (SEA) (described in section 4.15).

4.9 The Modified Three Step Search

This search (proposed by Li et al. [34]) is identical to the usual Three Step Search (section 4.17) except that the first points immediately adjacent to the centre of the search space are examined as well as the usual nine points.

- If the centre point has the lowest distortion the search ends with the motion vector (0, 0).
- If the point of lowest distortion is immediately adjacent to the centre point then the nine points around the point of lowest distortion are examined and the motion vector points to the new point of lowest distortion. Naturally only points that were unexamined by previous stages are examined.
- If the point of lowest distortion is one of the usual Three Step Search points then the search continues as Three Step Search.

4.10 The New Direction of Minimum Distortion Algorithm

This search (as proposed by Kappangantula and Rao, [29]) is derived from the Two Dimensional Logarithmic search (described in Section 4.19) and the Three Step Search (described in section 4.17). The algorithm attempts to perform fewer computations, on average, than the Three Step or Logarithmic Searches without sacrificing performance.

The algorithm is as follows (see Fig 4.10 for an example search):

1. If the centre point in the search space, corresponding to the position of the search block, has MAD = M and M < THRESHOLD, stop. The block is then classified as a stationary block. A THRESHOLD value of 32 is used in the simulation. Initially $i = 0$ and $j = 0$.
2. The following pixel positions are searched $(i-4, j)$, $(i, j-4)$, $(i+4, j)$ and $(i, j-4)$. If the minimum MAD form these search points is less than M, go to step 4. If the minimum < THRESHOLD, stop.
3. If the best match from the previous step is at $(i-4, j)$ the points to be searched are $(i-4, j-4)$ and $(i-4, j+4)$, for $(i+4, j)$ the points to be searched are $(i+4, j-4)$ and $(i+4, j+4)$.

If the best match from the previous step is at $(i, j-4)$ the points to be searched are $(i-4, j-4)$ and $(i+4, j-4)$, for $(i, j+4)$ the points to be searched are $(i+4, j+4)$ and $(i-4, j+4)$.

The tests for THRESHOLD and M are performed as in the previous step.

4. This is the same as for Step 2 except the spacing of the search positions is halved and (i, j) is set to the position of the current best match.

5. Similar to Step 3.

6. The spacing is halved again.

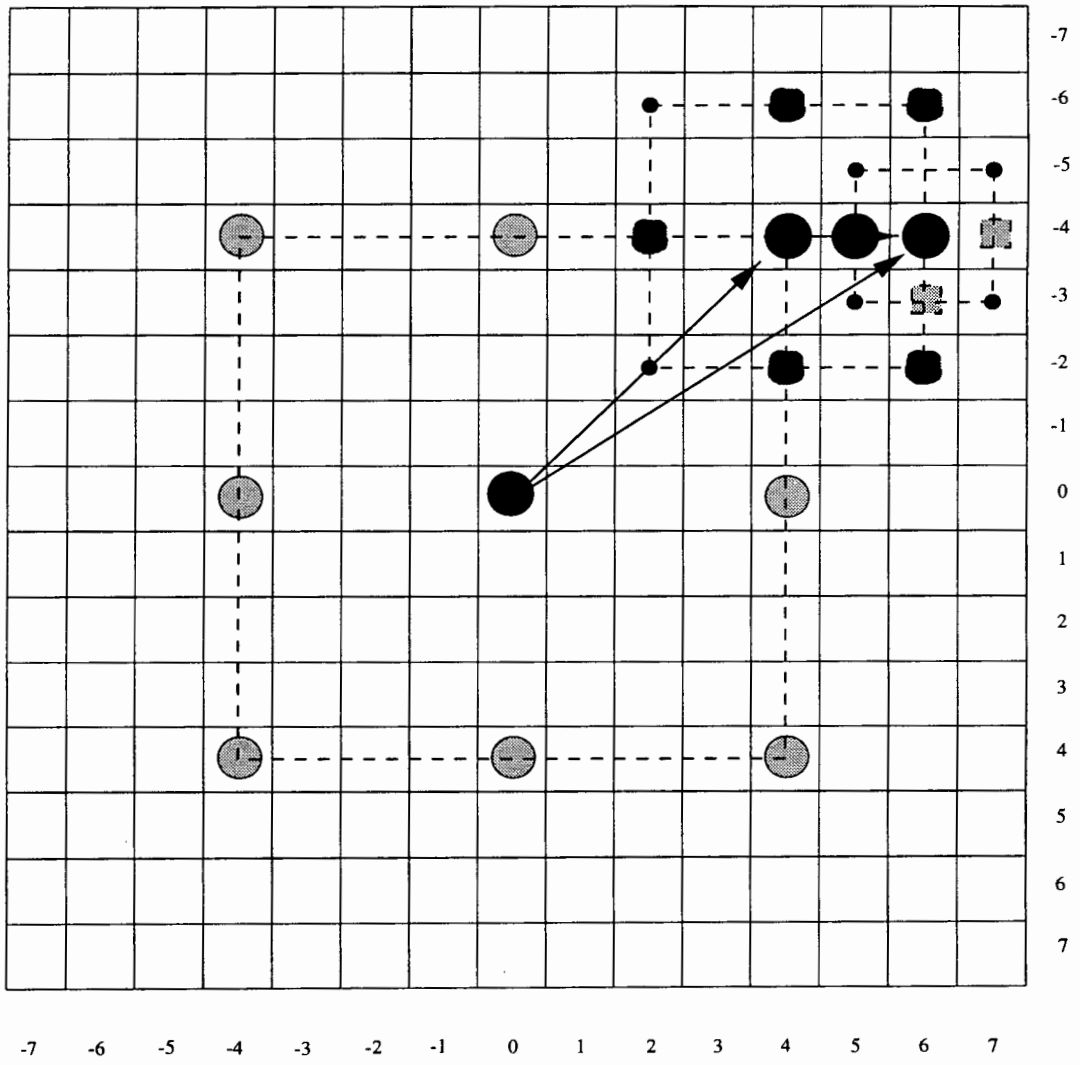


Figure 4.10: Example search patterns for the New Direction of Minimum Distortion.

4.11 The One Dimensional Full Search

This search algorithm (as proposed by Chen et al. [9], see Fig. 4.11) is related to the Conjugate search. The first stage searches the points on the horizontal line to the left and right (inclusive of the centre point) of the centre of the search area and finds the minimum distortion (MAD or MSE) of those points. The next stage searches all points vertically above and below the point of minimum horizontal distortion to find a new point of minimum distortion. The next stage searches horizontally around this new point searching half the points of the previous stages. The last stage searches vertically to find the point of least distortion.

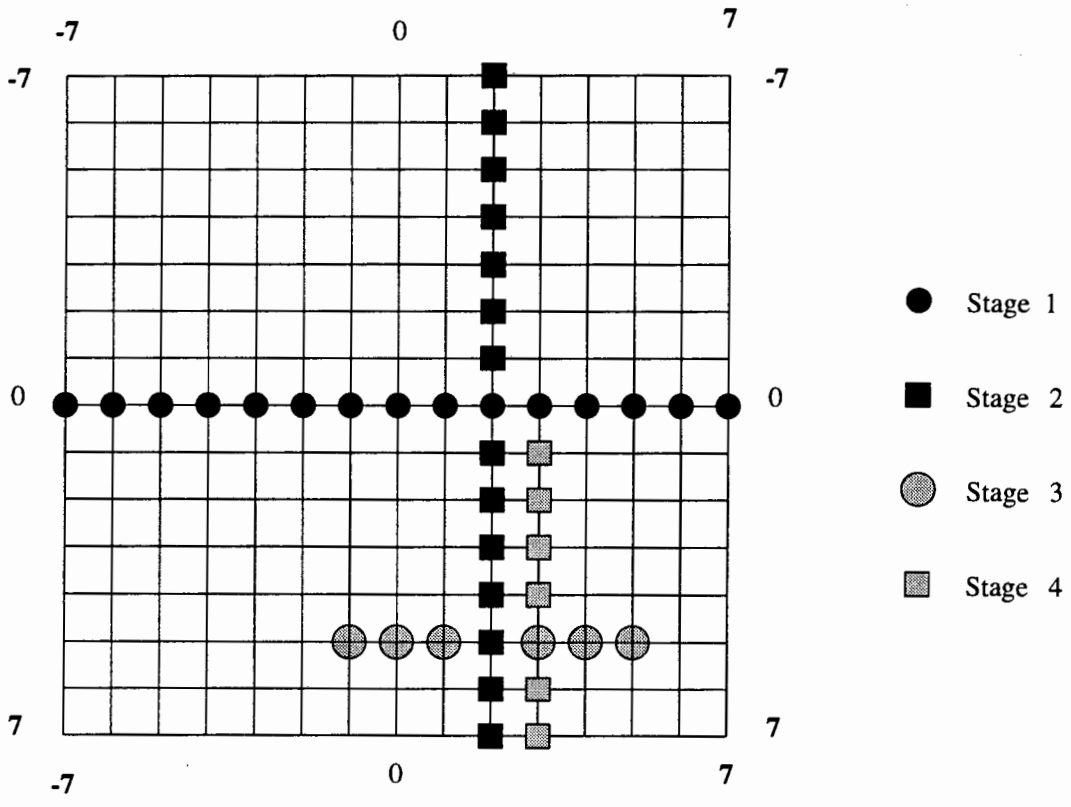


Figure 4.11: Example search points for One Dimensional Full Search.

4.12 Population Based Incremental Learning (PBIL)

The PBIL is derived from the genetic algorithm [3]. The basic method is as follows:

1. Code each trial point location as a binary vector of two numbers.

2. Initialise a prototype vector (P) of the same length as the binary vector and initialise the elements to 0.5. A better approach is to initialise the vector to the position of the

search block. If the element of the bit vector of the search block position is one then the element in the prototype vector is 0.75 (the choice is arbitrary but larger than 0.5) and 0.25 (less than 0.5) otherwise. This biases the search to the area around the position of the search block.

3. For each element in a population of size 30 and for 100 generations (these numbers are arbitrarily chosen).
   - Generate a random bit-string. The probability of a bit being 1 or 0 depends on the corresponding element of P.
   - Evaluate the distortion measure at the point specified by the bit-string.
   - If the new bit-string has a worse distortion measure than its predecessors then reject it.
4. Update each element (i) of P by adding $L(1 - P[i])$ if the bit in the surviving bit-string is 1 and $L(P[i] - 1)$ if 0. Here L is the learning rate, $0.005 - 0.05$ is suggested, and 0.1 used.
5. Occasionally randomly mutate P. The mutation value used is ± 0.01 with a probability 0.02.
6. After all generations are examined the final bit vector is generated as follows: if P[i] is less than 0.5 then the bit is zero, otherwise 1. The final point of displacement is thus found.

4.13 The Parallel Hierarchical One-Dimensional Search (PHODS)

With certain exceptions, notably Full Search and Successive Elimination, block motion estimation algorithms make the assumption that the distortion increases monotonically as one moves away from the direction/point of minimum distortion. This assumption is not always warranted and the algorithm may converge to a local minimum. To achieve a low bit-rate it is not always necessary to find the direction of minimum distortion; one can accept a direction with a small enough distortion.

PHODS (proposed by Chen et al. [7]) assumes that the contours of the distortion function increases monotonically along the axes. Instead of finding the two dimensional motion vector independently PHODS finds the lowest distortions along each axis. The estimate of the motion vector on the x-axis becomes the x-part of the final motion vector, the lowest distortion on the y-axis similarly becomes the y-part of the motion vector(see Fig. 4.12). The algorithm is as follows:

- Initialisation
  - $S = 2^{\lfloor log_2 p \rfloor}$
  - $\Delta x = 0$; $\Delta y = 0$;
- While($S < 0$) /* Loop executed $\lfloor log_2 p \rfloor + 1$ times. */
  - x axis (Δx, 0) $\leftarrow$ the location of min(D(Δx - S, 0), D(Δx, 0), D(Δx + S, 0));
  - y axis (Δy, 0) $\leftarrow$ the location of min(D(0, Δy - S), D(0, Δy), D(0, Δy + S));
  - S = S/2 ;
- end While_loop;
- The motion vector is (Δx, Δy).

where,

- Note step size S = 4 when $4 \leq p \leq 7$ and $p = \frac{(block\ width\ OR\ block\ height)}{2}$.
- $\lfloor \cdot \rfloor$ is a lower integer truncation function in this context.
- Where possible the minimum distortions along the axes are calculated in parallel.
- Distortion measure used is minimum absolute difference (MAD), although others could be used.

4.14 The Plus Pattern Search (PPS)

This search (proposed by Fok et al. [2]) is identical to the Subblock Full Search Algorithm except the distortion measure examines points in the shape of a plus (or cross) as illustrated in Fig. 4.13.

The distortion measure is as follows:

$$\begin{aligned} MAD_{PPS} &= \frac{1}{K} \sum_{n=0}^{N-1} \mid f_k(x+n, y+H) - f_{k-1}(i+n, j+H) \mid \\ &\quad + \sum_{n=0}^{H-1} \mid f_k(x+H, y+n) - f_{k-1}(i+H, j+n) \mid \end{aligned} \tag{4.3}$$

Where $f_k(x, y)$ is the pixel value at location (x, y) of frame k. $H = \frac{N}{2} - 1$ and $K = 2N - 1$.

1. For a NxN block in the k-th frame, define a search area of size $(2W + 1) \times (2W + 1)$
2. For each location in the search area measure the distortion using equation 4.3.
3. Re-examine the R best matches using the usual distortion measures, e.g., *MAD* or *MSE*.

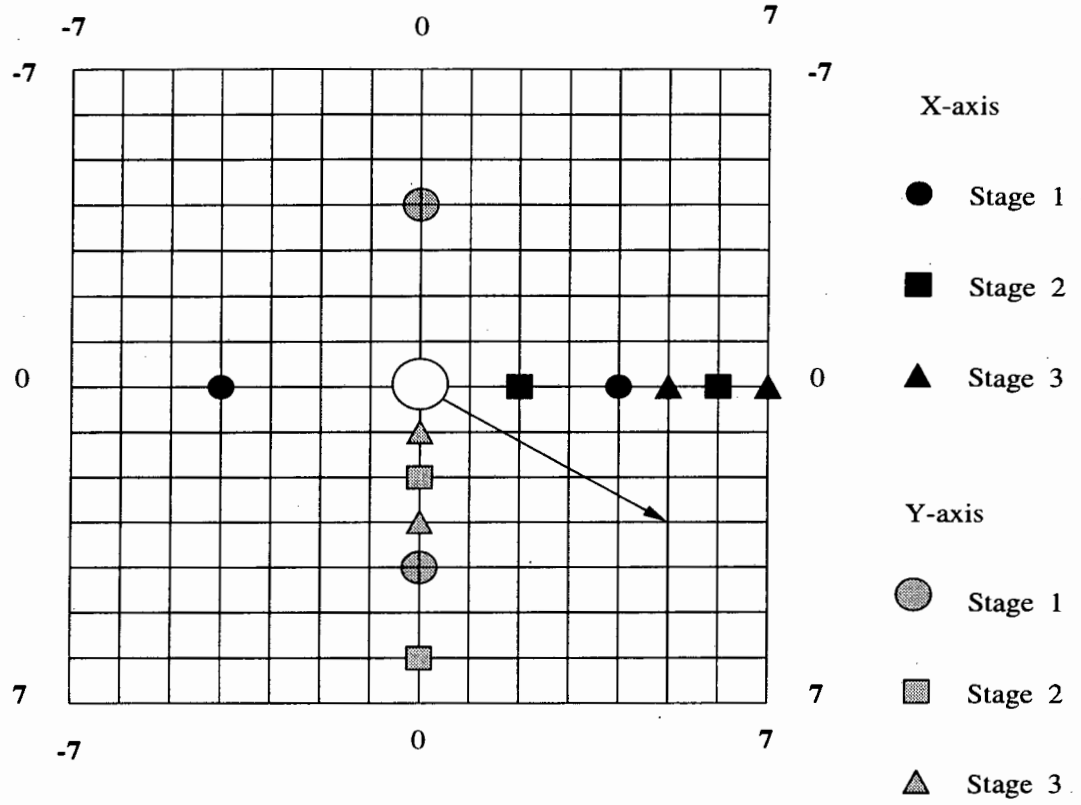


Figure 4.12: Example search points for Parallel Hierarchical One-Dimensional Search (PHODS).

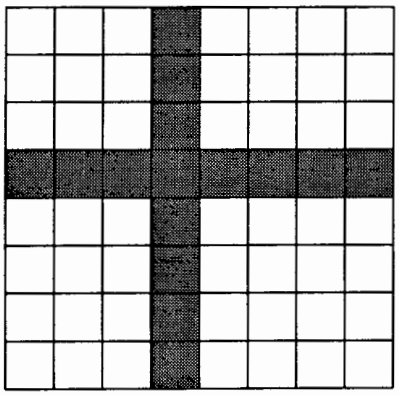


Figure 4.13: The feature for the Plus Pattern Search.

4.15 The Successive Elimination Algorithm (SEA)

This search (proposed by Li and Salari [35]) relies on MAD exclusively for a distortion measure. It is an attempt to speed up the full search by searching a subset of the points in the reference frame.

The SEA is an example of a feature based search algorithm. It classifies each block in terms of the sum of all pixels within the block or sumnorm. This section describes a method to rapidly calculate the the sumnorms and explains how the MAD is used to place upper and lower bounds on the sumnorm of blocks capable of improving the current best match.

4.15.1 Fast calculation of sumnorms

The operation of calculating the sumnorm at every point in the image (ignoring the points on the border of the image where this would be impossible) is essentially a convolution operation of the image and a square of the same dimensions as the search block. All the pixel values in the square is set to one. The convolution operation is thus separable. The convolution is thus of the image with two vectors with all elements one and with number of elements in each vector equal to the dimensions of the square respectively.

The calculation proceeds as follows for block size $N \times N$ (N is usually eight):

1. For the first element of each row calculate the sum of the first N points on the row ($\sum_{i=0}^{N-1} f(i,k)$, k is the row) and save as $C_{11}, C_{12}, ...$ For example C_{11} is the sum of the first eight pixels on row one. The second element of each row strip is calculated as follows: $C_{21} = C_{11} - f(1,1) + f(N+1,1)$ and $C_{2w} = C_{2w} - f(1,w) + f(N+1,w)$ and so on.

2. For the first element of each column calculate the sum of the first N points on the column ($\sum j = 0^{N-1} C_{jk}$, k is the row) and save as $D_{11}, D_{12}, ...$ The second element of each column strip is calculated as follows: $D_{21} = D_{11} - C_{11} + C_{(N+1)1}$ and $D_{2w} = D_{2w} - C_{1w} + C(N+1)w$ and so on. The values of D are the sumnorms.

The above reduces the number of operations in calculating the MAD. Without this method naively calculating sumnorms requires: (for a $W \times H$ image):

- $\frac{W.H}{N^2}$ blocks
- N^2 subtractions per block.
- $N^2 - 1$ additions per block.
- **total**: $\frac{1}{N^2} WH(2N^2 - 1)$ operations.

Whereas with the previous method this can be reduced to:

- for the row calculations: $W(N-1)+2W(H-N-1)$ operations.
- for the column calculations: $(H-N)((N-1)+2(W-N-1))$ operations.
- **total**: $4WH-(H-N)(N+3)-3W(N+1)$ operations.

4.15.2 Theoretical Background of Successive Elimination Algorithm

The following result allows tighter and tighter bounds to be placed on the sumnorms of blocks with potentially better matches. The algorithm relies on the following result, derived as follows:

- Block size is $N \times N$
- Search window size is $(2M+1) \times (2M+1)$
- f(i, j, t) represents a pixel intensity at (i, j) in frame t.

Now,

$$\| f(i,j,t) \| - \| f(i-x,j-y,t-1) \| \leq | f(i,j,t) - f(i-x,j-y,t-1) | \tag{4.4}$$

which breaks down to:

$$| f(i,j,t) | - | f(i-x,j-y,t-1) | \leq | f(i,j,t) - f(i-x,j-y,t-1) | \tag{4.5}$$

and,

$$| f(i-x,j-y,t-1) | - | f(i,j,t) | \leq | f(i,j,t) - f(i-x,j-y,t-1) | \tag{4.6}$$

Summing on both sides of of the above two equations and using the following:

$$\sum_{i=1}^{N}\sum_{j=1}^{N} f(i,j,t) = R \tag{4.7}$$

$$\sum_{i=1}^{N}\sum_{j=1}^{N} f(i-x,j-y,t-1) = M \tag{4.8}$$

$$\sum_{i=1}^{N}\sum_{j=1}^{N} | f(i,j,t) - f(i-x,j-y,t-1) | = MAD(x,y) \tag{4.9}$$

It is apparent that,

$$R - M(x,y) \leq MAD(x,y) \tag{4.10}$$

and,

$$M(x,y) - R \leq MAD(x,y) \tag{4.11}$$

Combining:

$$R - MAD(x,y) \leq M(x,y) \leq R + MAD(x,y) \tag{4.12}$$

This is the major result of the SEA algorithm. It implies that the block with the best match is constrained by equation 4.12. This result implies that the search should only be done on blocks satisfying 4.12. Ideally the number of blocks in the search area is less and at each stage the limits on the acceptable sumnorm become tighter, thus excluding more of the search space.

4.16 The Sub-block Full Search Algorithm

This is a variant of the Full Search Algorithm in which a neighbourhood of pixels around the search point is searched. The search window is usually a 15x15 subimage.

4.17 The Three Step Search

In this search (proposed by Koga et al. [41]) nine points are searched at each of three steps. At each step the distance between the search points is reduced. The new centre for the search pattern is the minimum distortion (MAD or MSE) of the search points in the current step. See Fig. 4.14 for an example search.

4.18 Accuracy Improvement and Cost Reduction of the Three Step Search Block Matching Algorithm

This search (proposed by Jong et al. [8]) essentially is four tiled Three Step Searches in each quarter of the search area with a final TSS in the centre of the search area (see Fig. 4.15).

The Three Step Searches are modified to centre each step (excluding the first) around not only the best match but a number of other close to best match candidates in order of goodness of match. Jong et al. [8] propose using the best match and the closest to best match point. The final minimum is the minimum of the of the four modified Three Step Searches. The motion vector is the displacement from the centre of the search area to the location of the minimum.

4.19 The Two dimensional Logarithmic Search

This search is also referred to as the Direction of Minimum Distortion (DMD) by Srinivasan and Rao [50]. This is based on the assumption that the distortion increases monotonically as one moves away from the direction of minimum distortion [41]. The direction of minimum distortion is defined such that the MSE or MAD is a minimum.

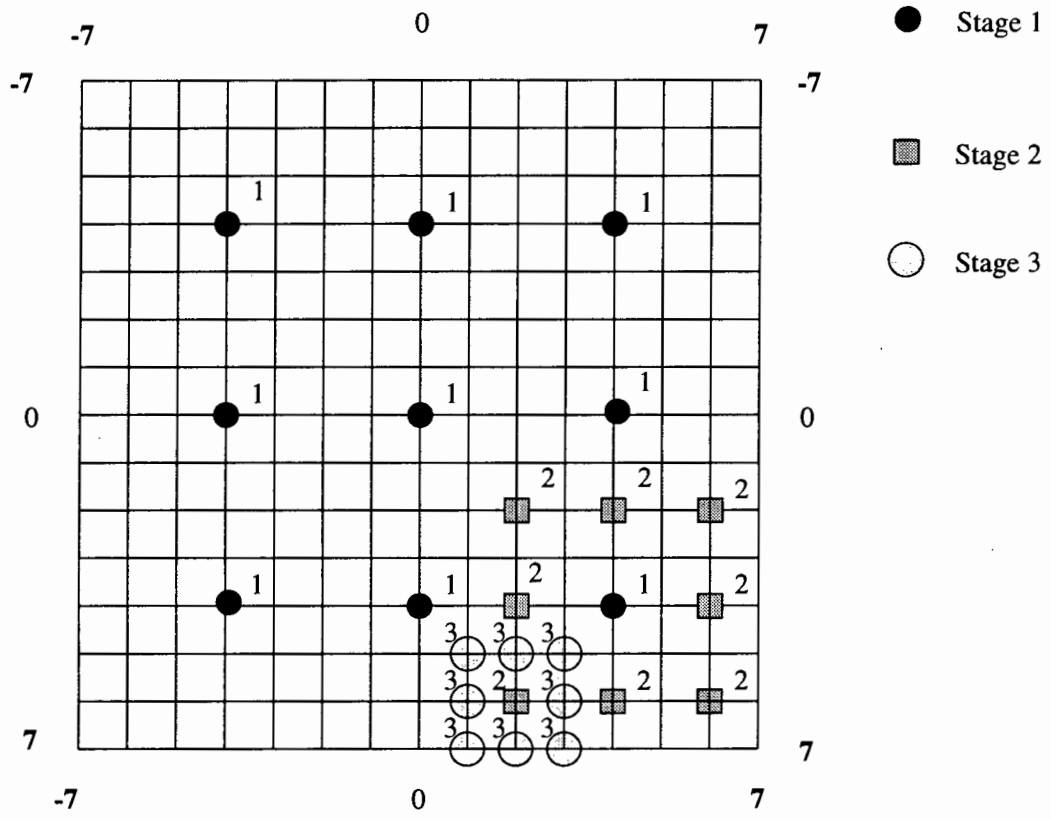


Figure 4.14: Example search points for the Three Step Search.

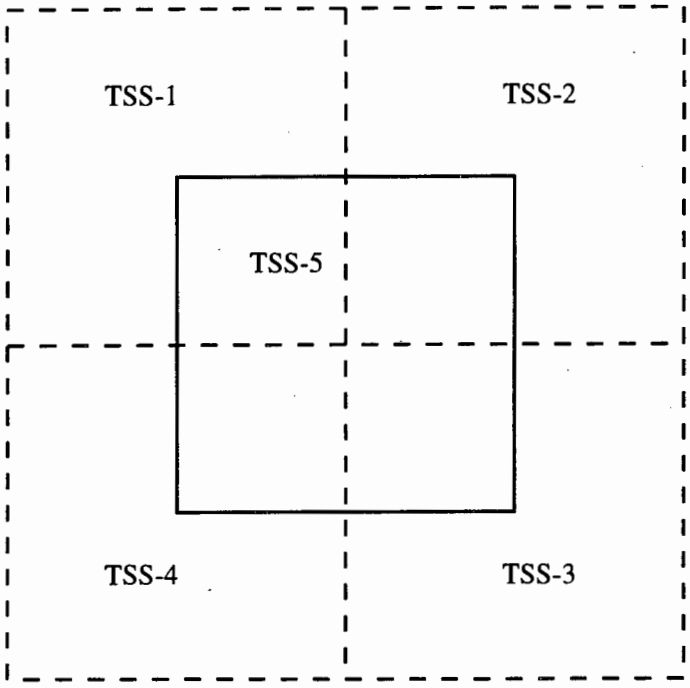


Figure 4.15: Example search layout for the Accuracy Improved Three Step Search.

In each step five points are searched. The distance between search points is reduced if the minimum is at the centre of the search points or if the boundary of the search area is encountered. Otherwise the search is repeated using the new minimum as the centre of the search. Points outside the search area are ignored. One continues the search until the distance between search points is one pixel. See Fig. 4.16 for an example implementation.

4.20 Summary

The algorithms presented cover a cross-section of approaches from ad-hoc (e.g., PHODS) to systematic (e.g., SEA). The algorithms can be divided into two basic types: those that examine the entire reference frame and those that examine only a subset of the reference frame.

Some proposed new algorithms are presented in Chapter 5.

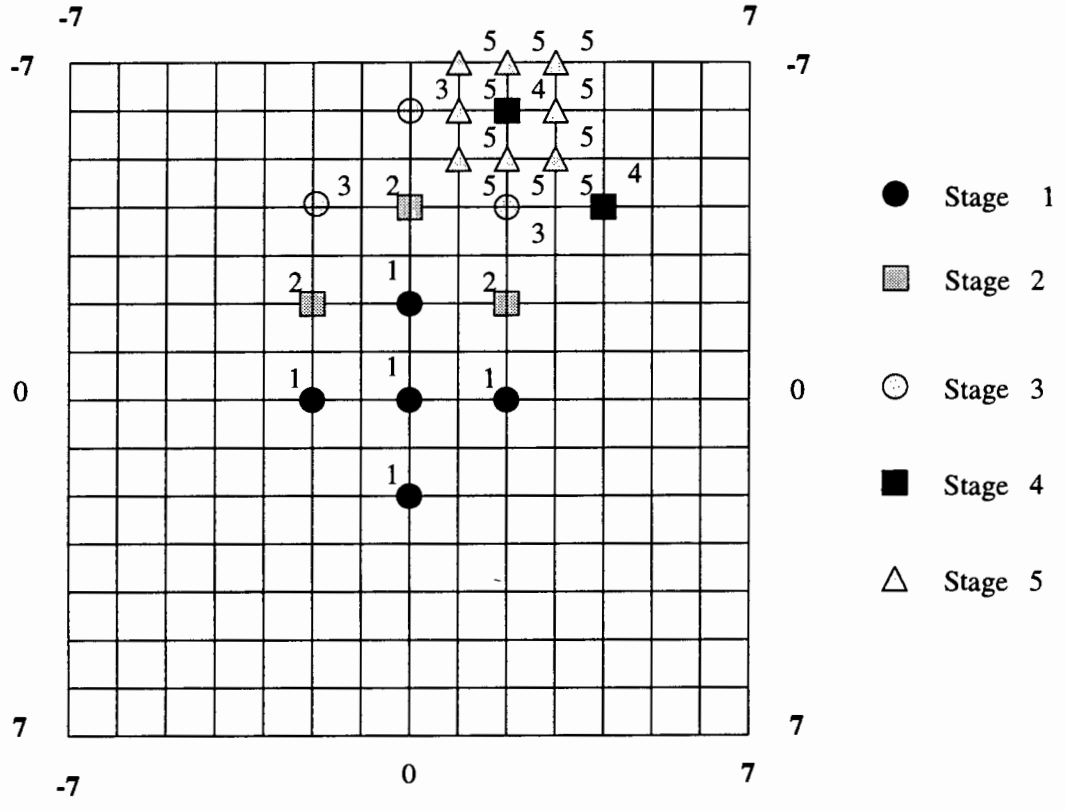


Figure 4.16: Example search points for the Two Dimensional Logarithmic Search.

Chapter 5

Some proposed new algorithms

The previous chapter examined some of the Block Motion Compensation Algorithms described in the literature. Some new algorithms are described here; based on insights gained from the previous chapter. The algorithms here attempt to find a balance between accuracy of the best match and speed of the search process. The proposed algorithms are quantatively compared to some current algorithms in the results chapter (Chapter 7).

The proposed algorithms are an attempt to speed up the Successive Elimination Algorithm by examining various features such as the energy, variance, monotony, modified monotony, and a few coefficients of the Hadamard Transform of the search block. The features are chosen for speed of calculation and for how well they describe the block. Sub-sampling the search space relies on the local correlation of pixels. The Tiled search attempts to combine the two approaches by examining subblocks of the reference frame one at a time (assuming a local motion vector) and by speeding up the search by using a feature (the mean as in SEA). The Projection Search attempts to minimise the MAD of a set of features by searching all possible points in the feature space with a monotonically increasing distortion until a point in the reference frame is found.

The following algorithms are proposed: some feature domain methods, an optimisation of the Successive Elimination Algorithm, the Projection Search, Sub-sampled and Filtered Sub-sampled Points Search.

5.1 Some Feature domain methods

The blocks on the search and reference frame can be described in terms of a set of carefully chosen features, e.g,. using some basis functions of Hadamard, DCT, etc.

Using a sub-set of the features available allows for faster searches for a matching block. The searches are performed using the basic Modified SEA (the Modified SEA is explained in Section 5.2) with the sumnorm pre-calculation stage replaced by the process of calculating one (or more) of the features to be described. For multiple features (eg. DCT coefficients)

the first feature fulfils the role of the SEA sumnorm and in finding the minimum of a set of points with identical sumnorms the MAD of the other features is minimised.

5.1.1 Using the DCT coefficients as features

The search is identical to the SEA with the exception that the initial MAD test is performed on a subset of the DCT coefficients. The final MAD is performed on points which pass the initial MAD test. The issues of the computational complexity of this search is examined and an attempt is made to speed up the calculation of DCT coefficients for overlapping blocks.

Using the DCT coefficients as features have the following advantages:

- A subset of the available coefficients can be used because of the energy compaction capability of the DCT.
- The quantisation of coefficients can be made part of the motion compensation process.

The disadvantage is computational complexity. A DCT needs to be evaluated for each overlapping block in the reference frame.

A method for the fast calculation of the DCT coefficients for overlapping blocks can be derived. This optimisation involves the row column decomposition of the DCT. Given the DCT of the row (column) in the previous block, the DCT of the row (column) of the new block is found as follows: subtract the DCT of the first pixel in the old row (column), left shift the DCT by one pixel followed by the addition of the DCT of the last pixel in the new row (column) of the new block. The Discrete Cosine Transform can be written as a row column decomposition as follows:

$$\begin{aligned} DCT(u,v) &= C(u)C(v)\sum_{x=0}^{N}\sum_{y=0}^{M} f(x,y)cos(\frac{(2x+1)u\pi}{2N})cos(\frac{(2y+1)v\pi}{2M}) \\ &= C(u)C(v)\sum_{x=0}^{N} cos(\frac{(2x+1)u\pi}{2N})\sum_{y=0}^{M} f(x,y)cos(\frac{(2y+1)v\pi}{2M}) \end{aligned}$$

$$\begin{aligned} where\ \ C(q) &= \frac{1}{\sqrt{2}} \cdots q = 0 \\ &= 1 \cdots \text{otherwise} \end{aligned}$$

Now,

$$DCT(u) = C(u)\sum_{x=0}^{N} f(x)cos(\frac{(2x+1)u\pi}{2N})$$

shifting by α,

$$\begin{aligned} DCT(u) &= C(u)\sum_{x=0}^{N} f(x+\alpha)cos(\frac{(2(x+\alpha)+1)u\pi}{2N}) \\ &= C(u)\sum_{x=0}^{N} f(x)cos(\frac{(2x+1)u\pi}{2N}+\frac{2\alpha u\pi}{2N}) \\ &= C(u)\sum_{x=0}^{N} f(x)cos(\frac{(2x+1)u\pi}{2N}+\frac{2\alpha u\pi}{2N}) \end{aligned}$$

Using the identity:

$$cos(\alpha+\beta) = (cos\alpha)(cos\beta) - (sin\alpha)(sin\beta)$$

the DCT can be written as:

$$\begin{aligned} DCT(u) &= C(u)\sum_{x=0}^{N} f(x)cos(\frac{(2x+1)u\pi}{2N})cos(\frac{2\alpha u\pi}{2N}) - \\ & C(u)\sum_{x=0}^{N} f(x)sin(\frac{(2x+1)u\pi}{2N})sin(\frac{2\alpha u\pi}{2N}) \end{aligned}$$

Now the Discrete Sine Transform can be decomposed in a similar fashion:

$$DST(u) = C(u)\sum_{x=0}^{N} f(x)sin(\frac{(2x+1)u\pi}{2N})$$

shifting by α,

$$\begin{aligned} DST(u) &= C(u)\sum_{x=0}^{N} f(x)sin(\frac{(2x+1)u\pi}{2N})cos(\frac{2\alpha u\pi}{2N}) - \\ & C(u)\sum_{x=0}^{N} f(x)cos(\frac{(2x+1)u\pi}{2N})sin(\frac{2\alpha u\pi}{2N}) \end{aligned}$$

Now, let:

- OST: The DST of eight pixels with the first pixel set to zero.
- OCT: The DCT of eight pixels with the first pixel set to zero.
- NST: The DST after the left shift of the pixels.
- NCT: The DCT after the left shift of the pixels.

$$
\begin{aligned}
OST &= (NCT)sin(\frac{2\alpha u\pi}{2N}) + (NST)cos(\frac{2\alpha u\pi}{2N}) \\
OCT &= (NCT)cos(\frac{2\alpha u\pi}{2N}) + (NST)sin(\frac{2\alpha u\pi}{2N}) \\
\\
\Rightarrow NST &= (OST)cos(\frac{2\alpha u\pi}{2N}) - (OCT)sin(\frac{2\alpha u\pi}{2N}) \\
\Rightarrow NCT &= (OST)sin(\frac{2\alpha u\pi}{2N}) - (OCT)cos(\frac{2\alpha u\pi}{2N})
\end{aligned}
$$

A difficulty with this method is that it requires the DST to be calculated in conjunction with the DCT which doubles the required storage. The algorithm is as follows for a two dimensional DCT:

1. Calculate the DCT and the DST for the first row of the frame.
2. For each point in the reference frame (except the ones with row or column greater than 8) perform a row DCT as follows
   - Subtract the DCT of the first element in the previous data.
   - Shift the transform of the data one pixel to the right, i.e., calculate the new DCT and DST with $\alpha = 1$.
   - Add in the DCT of the last element in the new data set.
3. Calculate the column DCT for the first row of the row transformed data.
4. Repeat 3 except using a column DCT.

Note that the subtraction of the first pixel and the addition of the last pixel in the data can be done in a single step.

The number of operations required for this algorithm is as follows:

- Subtraction of first pixel after left shift from previous DCT: n multiplications (scaling of transform of pixel value of unit value in the first position) and n subtractions: $2n$.
- Subtraction of first pixel from previous DST: $2n$.
- Calculation of NST: $3n$.
- Calculation of NCT: $3n$.
- Addition of last pixel after left shift to NST: $2n$.
- Addition of last pixel after left shift to NCT: $2n$.

The number of operations total $14n$ where n is the number of pixels. This operation is order n ($O(n)$) which grows slower than $nlog_2n$ (the order of calculations expected of a DCT/DST).

Chen et al. [10] proposed a fast DCT with:

- $\frac{3n}{2}(log_2n - 1) + 2$ additions,
- $nlog_2n - \frac{3n}{2} + 4$ multiplications,
- $\frac{5n}{2}log_2n - 3n + 6$ operations in total.

This implies this method is useful only for $n \geq 112$. Using a lookup table of all possible transform values for the DST and DCT of the first and last pixels respectively would reduce the number of addition and subtraction operations to $10n$. This implies $n > 37$ for the proposed method to require fewer operations than the fast DCT. Hardware implementation of this DCT calculation stage would reduce the time needed for this stage.

5.1.2 The monotony operator

The monotony operator is an attempt at a feature which is easy to calculate but also robust in terms of immunity to lighting changes. The monotony operator is defined [26] as the number of pixels with values smaller or equal to the centre pixel in a block. The block size used is 8x8 and the centre pixel used is at (4, 4) relative to the upper left-hand corner of the block. The search is identical to the SEA with the exception that only points with monotony values within a certain range centred on the search block's monotony are examined. A disadvantage is that this operator is vulnerable to noise. This feature is rotation invariant which is a disadvantage for BBMC.

5.1.3 A modification of the monotony operator

A modification to the monotony operator is proposed. The modification is identical to the above except for the calculation of the monotony. The modification is an attempt to remove the rotation invariance of the operator, and hence reduce the number of blocks to be examined during the search. The monotony is calculated as follows:

- Initially $monotony = 0$
- for each point which is less than or equal to the pixel value of the centre point ((4, 4) from the upper left hand corner of the block) in the block (8x8) the following is added to the monotony *column position* $+ 8 \times$ *row position*. Note that the centre pixel does *not* add to the monotony.

5.1.4 Using a subset of the Hadamard transform coefficients as features

The advantage of using the Hadamard Transform (see Appendix A) is that the coefficients can be rapidly calculated. Four coefficients are used. The search is identical to the SEA except that only the point(s) with the lowest MAD in the Hadamard Domain are examined using MAD in the spatial domain. The advantage is that in using only four transform coefficients the points can be examined and excluded faster (16 times) than MAD in the spatial domain with 64 pixel values. See Fig. 5.1 for the coefficients. Note that the dark areas are subtracted from the light areas within each coefficient.

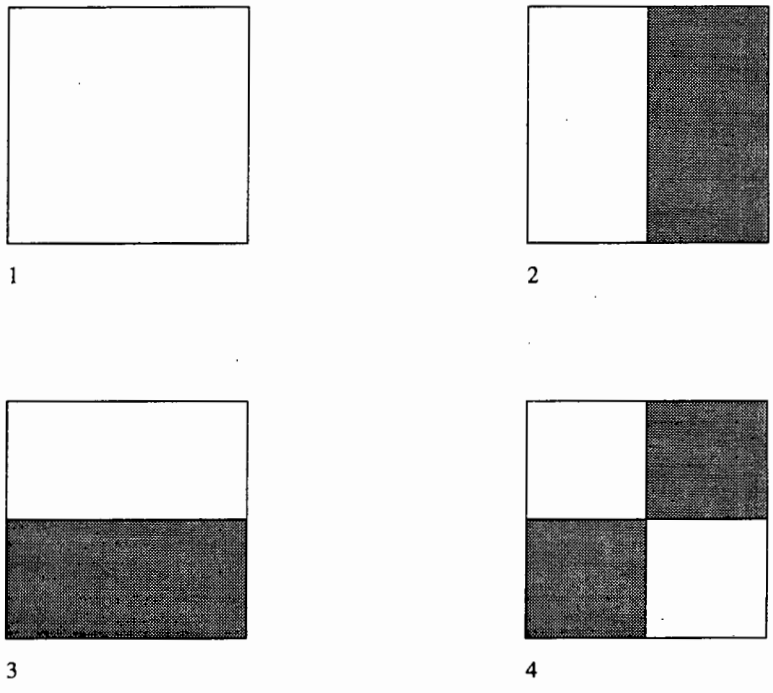


Figure 5.1: Hadamard coefficients used.

5.1.5 Variance

This is similar to the Monotony and Modified Monotony approaches. Variance is used in an attempt to find a more noise immune measure than the Monotony methods. The definition of Variance is as follows: $\sum_{i=1}^{N}(x_i - \overline{x})^2$. The search is identical to the SEA with the exception that only points with variance values in a certain range centred on the search block's variance are examined.

5.1.6 Energy

For good Signal to Noise Ratios energy should be a fairly noise immune feature. The energy in each block is computed and, similar to the approach in Monotony and Modified Monotony, the blocks with a particular mean are searched for the block with energy closest to the search block. The computationally expensive *MAD* is then performed on this best fit block. The energy is defined as follows:

$$Energy = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} x_{ij}^2$$

5.2 A Modified SEA

This algorithm is a modification of the Successive Elimination Algorithm. Each point is sorted in order of sumnorm and the search operates on this sorted list and not on the image of sumnorms (the low pass filtered image). The problem of finding the optimum motion vector in region matching methods can thus be reduced to a sorting problem. One firsts sorts to find points with the best distortion measure and then by distance to minimise the length motion vector.

The algorithm is a good basis on which to construct feature based searches (as in DCT and Monotony based searches), in that the sumnorm calculation stage can be replaced by the feature (or features) generation stage. All that is required is that the feature values can be ordered. If several features are used then several SEA constraint equations may be used; one for each feature. This section examines the algorithm itself and then proposes a modification.

5.2.1 The basic Modified SEA algorithm is as follows:

- Generate the reference frame in the same way as in the Successive Elimination Algorithm (SEA).
- Generate lists of all points in the reference frame with the same value of sumnorm.
- When given a search block, sum all pixel values and find the list with the closest value (sumnorm). It is possible to have two lists differing from the search value by the same amount.
- Search the list(s) for the point with best distortion match closest to the position of the search block.

It is interesting to note that generating the reference frame sumnorms is the same as convolving or correlating with a block of given size (here 8x8). This implies a low pass filtering of the reference image.

5.2.2 A Modification

The time consuming part of the search process is that each search with a new search block on the same frame, with the exception of the sumnorm/feature generation and ordering stage, starts fresh. This modification attempts to use the results generated in finding the previous best match for the current frame to speed to process of finding the new best match. The basic difference $| x - y |$ can be modified to allow the current MAD to be determined in terms of the MAD for a past search block. The upper and lower bounds of the new MAD is calculated in terms of the old (past) MAD. Now let $y = z + d$, where d is the difference between z and y:

$$
\begin{aligned}
& | x - (z + d) | \\
& | (x - z) - d | \\
|| x - z | - | d || \quad & \leq \quad | (x - z) - d | \\
| (x - z) - d | \quad & \geq \quad | x - z | + | d |
\end{aligned}
$$

The variables are as follows:

x: Current point on search frame

z: Previous (long past) search block.

d: Difference between the new search block and the previous search block.

Now,

$$
\begin{aligned}
\sum_i^N \sum_j^N | (x_{ij} - z_{ij}) - d_{ij}) | \quad & \leq \quad \sum_i^N \sum_j^N (| x_{ij} - z_{ij} | + | d_{ij} |) \\
\sum_i^N \sum_j^N | x_{ij} - z_{ij} | + \sum_i^N \sum_j^N | d_{ij} | \quad & \leq \quad old\_MAD - \sum_i^N \sum_j^N | d_{ij} |
\end{aligned}
$$

and

$$
\begin{aligned}
\sum_i^N \sum_j^N | (x_{ij} - z_{ij}) - d_{ij}) | \quad & \geq \quad \sum_i^N \sum_j^N (| x_{ij} - z_{ij} | + | d_{ij} |) \\
\sum_i^N \sum_j^N | x_{ij} - z_{ij} | + \sum_i^N \sum_j^N | d_{ij} | \quad & = \quad old\_MAD + \sum_i^N \sum_j^N | d_{ij} |
\end{aligned}
$$

Of interest is the lower bound. In general the following is true of lower bounds:

$$
\begin{aligned}
| a + b | \quad & \leq \quad | a | + | b | \quad \ldots \text{ Triangle Inequality} \\
\text{Let } a \quad & = \quad x + y
\end{aligned}
$$

$$
\begin{aligned}
\text{Let } b &= -x \\
|\, x+y-x \,| &\leq |\, x+y \,| + |\, -x \,| \\
|\, y \,| - |\, x \,| &\leq |\, x+y \,|
\end{aligned}
$$

$$
\begin{aligned}
\text{Since } MAD &= \sum_i^N \sum_j^N |\, (x_{ij} - z_{ij}) - d_{ij} \,|, \\
\sum_i^N \sum_j^N |\, (x_{ij} - z_{ij}) \,| - \sum_i^N \sum_j^N |\, d_{ij} \,| &\leq \sum_i^N \sum_j^N |\, (x_{ij} - z_{ij}) - d_{ij} \,| \\
\textit{Old } MAD - \sum_i^N \sum_j^N |\, d_{ij} \,| &\leq \sum_i^N \sum_j^N |\, (x_{ij} - z_{ij}) - d_{ij} \,|
\end{aligned}
$$

The form of the search for the best match is as follows:

1. A Full Search (or SEA) must be performed with at least one search block in order to construct a table of old MAD values versus pixel position.

2. If the new search block is identical to the first block then one can search for the closest point to the location of the reference block from the list of points of minimum MAD on the table of old MAD values.

3. For each new search block different from the first search block, calculate the lower bound at each pixel position. The term $\sum_i^N \sum_j^N |\, d_{ij} \,|$ can be pre-computed since this is the sum of the absolute value differences of the new and first search blocks.

4. Examine points with a smaller, lower bound on the new MAD than the current best match using the MAD measure. This ensures fewer points are searched with the expensive MAD calculation and that points with large new MADs are rapidly excluded from the search.

5.3 A Projection Search Algorithm

This algorithm approaches the search process from another perspective. Instead of examining points on the frame, it partitions the search space until a point in the search space is found which can be related to a point on the frame.

This algorithm relies on making an initial estimate of the distortion and only examining the points which could have this distortion. If the initial estimate proves unfounded a new larger estimate is generated and the search repeated. Each pass of the algorithm only examines a subset of the number of pixels available on the reference frame. One essentially tries to fit a hyper-cube/rectangle over a distortion shell/solid created by assuming a particular distortion value. One only closely examines points within the cube/rectangle. The dimensions of the

hyper-cube/rectangle correspond to the ranges of values for each feature which will allow for a distortion as given. Note that for points outside the distortion shell the distortion is necessarily greater. Only points with feature values within the cube/rectangle are examined and the ones with smallest distortion and closest to the search origin (as usual) are chosen. One should avoid the temptation to reduce the number of points examined by generating a better approximation to the distortion function since one here begins to perform a full search on a portion of the feature space, i.e., one examines more points. For a symmetric object such as a sphere (MSE distortion function in multiple dimensions) cube (MAD distortion function in multiple dimensions) the dimensions of the enclosing cube are simply the estimate of the distortion. For an ellipse (MSE distortion function taking quantisation into account) each dimension is a scaled version of the estimated distortion. The algorithm is as follows:

1. For each overlapping block in the reference frame pre-calculate the features of those blocks.
2. Sort the points into bins, using a hash function to sort the blocks.
3. For each search block generate the test values according to the hash function and the allowed distortion, and probe each bin for an acceptable block.
4. If no acceptable point is found increase the allowed distortion by one and repeat.

The following hash function is used: $\frac{(q_1*512 + q_2*256 + q_3*256 + q_1)}{16}$, where q_i refer to the first four Hadamard transform coefficients. Note that any features can be used provided they are ranked in order of energy compaction or how much of the data they explain. The data structure used is an open-ended hash table (each entry in the hash table can contain more than one entry/candidate point). Since most motion vectors are short it seems best to use 0 or so as the start distortion (if one uses *MAD*) and increment the allowed distortion until a point emerges with a distortion less than or equal to the allowed distortion.

As an optimisation one can ensure that with each trial of a particular search block each entry (slot) in the hash table is examined only once.

5.4 A Prefiltered Sampled points search

This search attempts to examine the effect of search area size in isolation to the effect of sub-sampling the search space (see the next section). The search proceeds as follows:

- Low pass filter the frame. Replace each pixel by the sum of the pixel values of a block NxN with one corner of the block corresponding to the pixel location. N is the subsample factor.
- Subsample the low pass filtered frame by a factor of N.

- Perform an SEA (or Modified SEA) on the samples and select the best match.
- Around the best match perform a full search on the original frame with a search window of dimension $N-1$ by $N-1$.
- The motion vector points to the pixel with lowest distortion.

5.5 A Sampled points search

This is an attempt to generalise searches like Three Step Search, Modified (New) Three Step search and the Two Dimensional Logarithmic search, etc. to cover the entire frame. Given that a pixel tends to be highly correlated with its immediate neighbours [42], it should not be necessary to examine every point during a full search. This search proceeds as follows:

- Sub-sample the images with a factor N.
- Use SEA or Modified SEA on the samples.
- On the best match found in the previous step perform a full search within some search window of dimension $N-1$ by $N-1$ centred on the best match.

5.6 The Tiled Search

The Tile Search attempts to exploit the fact that most motion vectors are short, typically less that 15 pixels [41], and that the positions of the search blocks are spaced eight pixels apart along the horizontal and vertical axes, while still providing the performance of a full search of the search space. The algorithm proceeds as follows:

- The reference frame is subdivided into a series of square 8x8 tiles centred (i.e., pixel $(4,4)$ from the upper left corner of the tile placed) on the position of the corresponding search block.
- The sumnorms of each pixel is calculated as in SEA (Section 4.15) (note that any feature or combination of features could be used instead).
- For each search block:

    1. Perform an SEA or modified SEA (or some feature search) on the pixels within the tile corresponding to the position of the search block.
    2. If the distortion is within an acceptable limit search the Tiles surrounding the search area and end the search.
    3. If the distortion is unacceptable include in the search area and search the Tiles bordering on the search area. Goto 2.

Note that Tiles outside the search area are excluded. The distortion regarded as acceptable used for *MAD* is 500. Since the motion vectors are likely to be short, examining the Tile corresponding to the position of the search block and its neighbours is likely to provide the desired minimum distortion. Note that there is a trade-off between speed and minimising distortion in adjusting the threshold of acceptable distortion. Searching the Tiles bordering on the search space after an acceptable motion vector is found serves two purposes, firstly, because of the shape of the Tiles a point with similar distortion may be closer to the origin but be in a Tile bordering the search area, and, secondly, since more points are examined the likelihood of finding a good match is improved. Clearly no point with equal distortion may be closer and no point with larger distortion is of interest. It is possible that a point with smaller distortion may be well away from the origin of the search block (but long motion vectors are not very likely) and reducing the threshold of acceptable distortion further, results in more Tiles examined.

5.7 Summary

The current algorithms which only examine a portion of the reference frame rely on a monotonically decreasing distortion along a path to the best match, and short final motion vectors. Full search algorithms rely on methods to speed up the examination of each point in preference to performing a computationally expensive *MAD* or *MSE*, an example of such an algorithm is the Successive Elimination Algorithm. The proposed algorithms are an attempt to speed up the Successive Elimination Algorithm by examining various features such as the energy, variance, monotony, modified monotony, and a few coefficients of the Hadamard Transform of the search block. The features are chosen for speed of calculation and for how well they describe the block. The Tiled search attempts to combine the two approaches by examining subblocks of the reference frame one at a time (assuming a local motion vector) and by speeding up the search by using a feature (the mean as in SEA). The Projection Search attempts to minimise the *MAD* of a set of features by searching all possible points in the feature space with a monotonically increasing distortion until a point in the reference frame is found. The algorithms are quantitatively compared in Chapter 6.

Chapter 6

The application of different Motion Compensation algorithms on some test video sequences

Chapters 4 and 5 presented algorithms from the literature and some new proposed algorithms respectively. The problem of how to assess the performance of these algorithms, in terms of both speed and accuracy of the best match, needs to be examined. First it is explained, in this chapter, how the results are generated, then an example of the application of the algorithms on the Windmill test video sequence is shown.

6.1 Generation of the data

This section describes the results measured from the data using the different BBMC algorithms. The results for a single test sequence (Windmill) are reproduced and examined.

Quantitative results are generated for the algorithms described in Chapters 4 and 5. The algorithms are compared as follows:

- Mean and Variance of the Motion Vectors.
- Mean and Variance of the Difference Frame.
- Mean and Variance of the absolute value of the Difference Frame. Denoted as *Di abs mean* and *Di abs var* in the figures respectively.
- The Peak Signal to Noise Ratio (PSNR) of the Difference Frame. The PSNR is defined as follows:

$$PSNR = -10log_{10}\frac{\sum_{x=0}^{x=P-1}\sum_{y=0}^{Q-1}[f(x,y)-g(x,y)]^2}{peak^2} \tag{6.1}$$

Where $peak = 255$ (8 bit resolution images), P is the number of columns, Q is the number of rows, $f(x,y)$ is the original frame and $g(x,y)$ is the estimated frame.

- The Discrete Frame Difference (DFD). The DFD is the sum of the absolute pixel values of all Difference Frames.

- The Entropy of the Motion Vectors, Difference Frame and combined Motion Vectors and Difference Frames. The entropy is defined as: [51]

$$H = \sum_i p_i log_2 p_i,$$

measured in bits. For the motion vectors, p_i is the ratio of the occurrence of each possible element of the motion vector to the number of elements in all motion vectors for all frames. For the difference frame, p_i is the ratio of the occurrence of each possible pixel value in the difference image to the number of pixels in the sequence of difference frames. For the cumulative difference, p_i is the ratio of the occurrence of each possible element of the motion vector and pixel value in the difference image to the sum of the number of elements in all motion vectors for all frames and the number of pixels in the sequence of difference frames.

Tables are generated comparing the performance of the algorithms on a particular video sequence with respect to Entropy, Difference Frame measures, Motion Vector measures, PSNR and DFD. Figures comparing the performance of each algorithm on a particular video sequence are generated for each measure. The video sequences were digitised from a selection of standard and non-standard sequences. The sequences are groups of 25 frames grabbed at the second, fifth, eighth, eleventh and fourteenth seconds. The following sequences were used: Animate1, Animate2, Boxing, CNN News, Crawls, Cross Country, Cycling, Disc3, Rotating Disc3, Table Tennis and Windmill. Each frame is 512x512 pixels greyscale grabbed by an EPIX framegrabber.

The data itself is located in Appendix B and enough data (of the Windmill sequence) to illustrate the performance of the algorithms is included in this chapter.

A good algorithm minimises the DFD or errors between the best match and search blocks; this implies the PSNR is maximised (see equation 6.1) and the mean of the absolute value of difference image is minimised. The pixel, motion vector and cumulative entropies represent the direct Huffman encoding of the data and must be minimised to achieve a minimum bit-rate. The mean of the motion vectors represents the average motion magnitude. The variance of the motion vector means is an indication of the non-uniformity of the motion. The variance of the absolute value of the difference image is an indication of the frequency content excluding the DC coefficient; ideally this should be minimised.

Table 6.1 relates the algorithms with the abbreviated algorithm names used in the figures

Abbreviated Algorithm	Expanded Name
2Dlog Search	Two Dimensional Logarithmic Search
Dynamic SW Adjustment	Dynamic Search Window Adjustment
Energy	Energy Feature based Search
Improved TSS	Improved Three Step Search
Modified Feature Search	A subset of the Hadamard Transform used
Modified Monotony	Modified Monotony based Feature Search
Modified SEA	Modified Successive Elimination Algorithms (SEA)
Modified SEA v2	Modified SEA version 2 uses the modification described in Section 5.2.2
Modified TSS	Modified Three Step Search
Monotony	Monotony based Feature Search
New DMD	New Direction Of Minimum Distortion
PBIL	Population Based Incremental Learning
Parallel Hierarchical One D Search	Parallel Hierarchical One Dimensional Search
Prefiltered Sampled Points k	Prefiltered Sampled Points with spacing of k pixels
Sampled Pts k	Sampled points search, spacing k pixels
TSS	Three Step Search
Variance	Variance based Feature Search

Table 6.1: Explanation of abbreviated Algorithm names.

and the tables. Tables 6.2, 6.3 and 6.4 contain the results of the application of all the algorithms listed on the Windmill test sequence.

Figure 6.1 is a bar graph comparing the DFD of the algorithms on the Windmill test sequence. Figure 6.2 compares the PSNR, Figure 6.3 compares the mean of the absolute value of the difference image. Note that algorithms with low DFD correspond to algorithms with low mean of the absolute value of the difference image, and with algorithms with a high PSNR. These three measures then appear to measure the goodness of the best match, or how close the search and best match blocks are. Figure 6.4 compares the variance of the absolute mean of the difference image.

Figure 6.5 compares the joint coding of difference images and motion vectors. Figure 6.6 compares the entropy of coding the motion vectors alone. Figure 6.7 compares the entropy of coding the difference images alone. These illustrate that coding and motion vector (and difference image) generation issues are different. An algorithm with bad performance may do well if coded carefully.

Figure 6.8 compares the mean of the motion vectors. Figure 6.9 compares the variance of the motion vectors. Algorithms examining the entire reference frame tend to have high motion vector means and variances.

The results for all test sequences are quantitatively compared in Chapter 7.

Algorithm	Difference block			
			Absolute	
	Mean	Variance	Mean	Variance
2Dlog Search	-0.09	229.8	7.3	175.9
Block Based Gradient Descent Search	-0.07	408.9	9.6	316.1
Conjugate Directional Search	-0.13	280.2	7.9	217.1
Conjugate Search	-0.14	278.1	7.9	215.1
Cross Pattern Search	-0.10	235.8	7.5	179.1
Cross Search	-0.30	565.6	12.0	421.1
Dynamic SW Adjustment	-0.12	229.5	7.4	175.0
Energy	-0.52	302.9	9.3	217.3
Four Step Search	-0.09	225.7	7.3	172.4
Improved TSS	-0.13	176.1	6.8	130.5
Modified Conjugate Search	-0.08	210.2	7.1	159.7
Modified Feature Search	-0.05	164.1	6.7	118.8
Modified Monotony	-0.26	306.8	9.4	217.9
Modified SEA	-0.06	130.7	6.2	92.5
Modified SEA v2	-0.22	567.6	13.6	381.7
Modified TSS	-0.10	211.9	7.2	160.5
Monotony	-0.26	305.1	9.4	217.0
New DMD	-0.37	393.0	10.3	287.2
One Dimensional Full Search	-0.07	1033.8	17.0	745.2
PBIL	-0.21	342.1	9.5	251.0
Parallel Hierarchical One D Search	0.00	1091.0	17.4	786.6
Plus Pattern Search	-0.10	271.3	7.9	209.5
Prefiltered Sampled Points 2	-1.84	596.0	12.3	448.8
Prefiltered Sampled Points 3	0.28	412.3	9.8	315.7
Prefiltered Sampled Points 4	0.30	360.9	9.0	279.4
Prefiltered Sampled Points 5	0.41	310.3	8.5	238.8
Projection Search	-0.10	743.5	15.0	517.5
Sampled Pts 2	-0.11	143.1	6.5	101.5
Sampled Pts 3	-0.27	265.3	8.5	192.4
Sampled Pts 4	-0.27	233.3	8.1	168.2
Sampled Pts 5	-0.35	323.8	9.5	234.5
Subblock Full Search	-0.09	191.4	6.8	144.6
TSS	-0.21	236.7	7.6	178.9
Tile search	-0.12	210.5	7.8	149.9
Variance	0.02	728.6	15.2	497.4

Table 6.2: Comparison of Algorithms for the sequence **windmill**

Algorithm	Motion Vector		PSNR	DFD
	Mean	Variance		(x 10^8)
2Dlog Search	2.422	2.351	24.52	2.386
Block Based Gradient Descent Search	1.524	1.996	22.01	3.133
Conjugate Directional Search	1.830	2.016	23.66	2.584
Conjugate Search	1.815	1.920	23.69	2.581
Cross Pattern Search	4.104	7.202	24.40	2.449
Cross Search	3.512	6.788	20.61	3.909
Dynamic SW Adjustment	2.483	1.942	24.52	2.400
Energy	134.534	17910	23.32	3.012
Four Step Search	2.713	3.029	24.60	2.373
Improved TSS	5.265	21.323	25.67	2.196
Modified Conjugate Search	3.287	5.316	24.90	2.310
Modified Feature Search	15.465	3277	25.98	2.186
Modified Monotony	160.194	21837	23.26	3.065
Modified SEA	70.883	11395	26.97	2.008
Modified SEA v2	218.814	20745	20.59	4.432
Modified TSS	2.703	3.838	24.87	2.331
Monotony	155.092	21030	23.29	3.054
New DMD	4.409	7.972	22.19	3.346
One Dimensional Full Search	0.690	4.283	17.99	5.523
PBIL	47.800	6882	22.79	3.103
Parallel Hierarchical One D Search	0.001	0.005	17.75	5.671
Plus Pattern Search	4.232	7.108	23.80	2.556
Prefiltered Sampled Points 2	117.689	27463	20.38	3.989
Prefiltered Sampled Points 3	13.903	3634	21.98	3.195
Prefiltered Sampled Points 4	18.840	4299	22.56	2.935
Prefiltered Sampled Points 5	20.135	4402	23.21	2.753
Projection Search	165.867	16285	19.42	4.886
Sampled Pts 2	78.342	12276	26.57	2.097
Sampled Pts 3	116.763	15396	23.89	2.776
Sampled Pts 4	112.233	15045	24.45	2.623
Sampled Pts 5	131.927	16145	23.03	3.074
Subblock Full Search	4.193	8.789	25.31	2.223
TSS	3.520	4.533	24.39	2.473
Tile search	33.202	7526	24.90	2.531
Variance	198.325	17903	19.51	4.943

Table 6.3: Comparison of Algorithms for the sequence **windmill**

Algorithm	Entropy		
	Pixel	Motion Vector	Cumulative
2Dlog Search	5.09	2.76	5.05
Block Based Gradient Descent Search	5.38	2.26	5.31
Conjugate Directional Search	5.18	2.39	5.13
Conjugate Search	5.18	2.38	5.13
Cross Pattern Search	5.14	3.51	5.11
Cross Search	5.64	3.49	5.60
Dynamic SW Adjustment	5.11	2.71	5.06
Energy	5.34	8.08	5.48
Four Step Search	5.09	2.92	5.04
Improved TSS	5.00	3.87	4.97
Modified Conjugate Search	5.06	3.05	5.02
Modified Feature Search	5.00	4.15	4.99
Modified Monotony	5.38	8.34	5.53
Modified SEA	4.88	6.28	4.96
Modified SEA v2	6.00	9.16	6.18
Modified TSS	5.07	2.92	5.03
Monotony	5.37	8.32	5.52
New DMD	5.53	3.71	5.49
One Dimensional Full Search	6.09	0.76	6.00
PBIL	5.39	4.53	5.40
Parallel Hierarchical One D Search	6.13	0.00	6.04
Plus Pattern Search	5.18	3.57	5.14
Prefiltered Sampled Points 2	5.66	5.63	5.72
Prefiltered Sampled Points 3	5.39	2.83	5.35
Prefiltered Sampled Points 4	5.29	3.49	5.26
Prefiltered Sampled Points 5	5.22	3.82	5.20
Projection Search	5.96	8.98	6.11
Sampled Pts 2	4.94	6.14	5.02
Sampled Pts 3	5.24	7.84	5.37
Sampled Pts 4	5.19	7.00	5.31
Sampled Pts 5	5.35	8.20	5.50
Subblock Full Search	5.00	3.49	4.97
TSS	5.14	3.25	5.10
Tile search	5.19	4.13	5.19
Variance	6.02	9.24	6.19

Table 6.4: Comparison of Algorithms for the sequence **windmill**

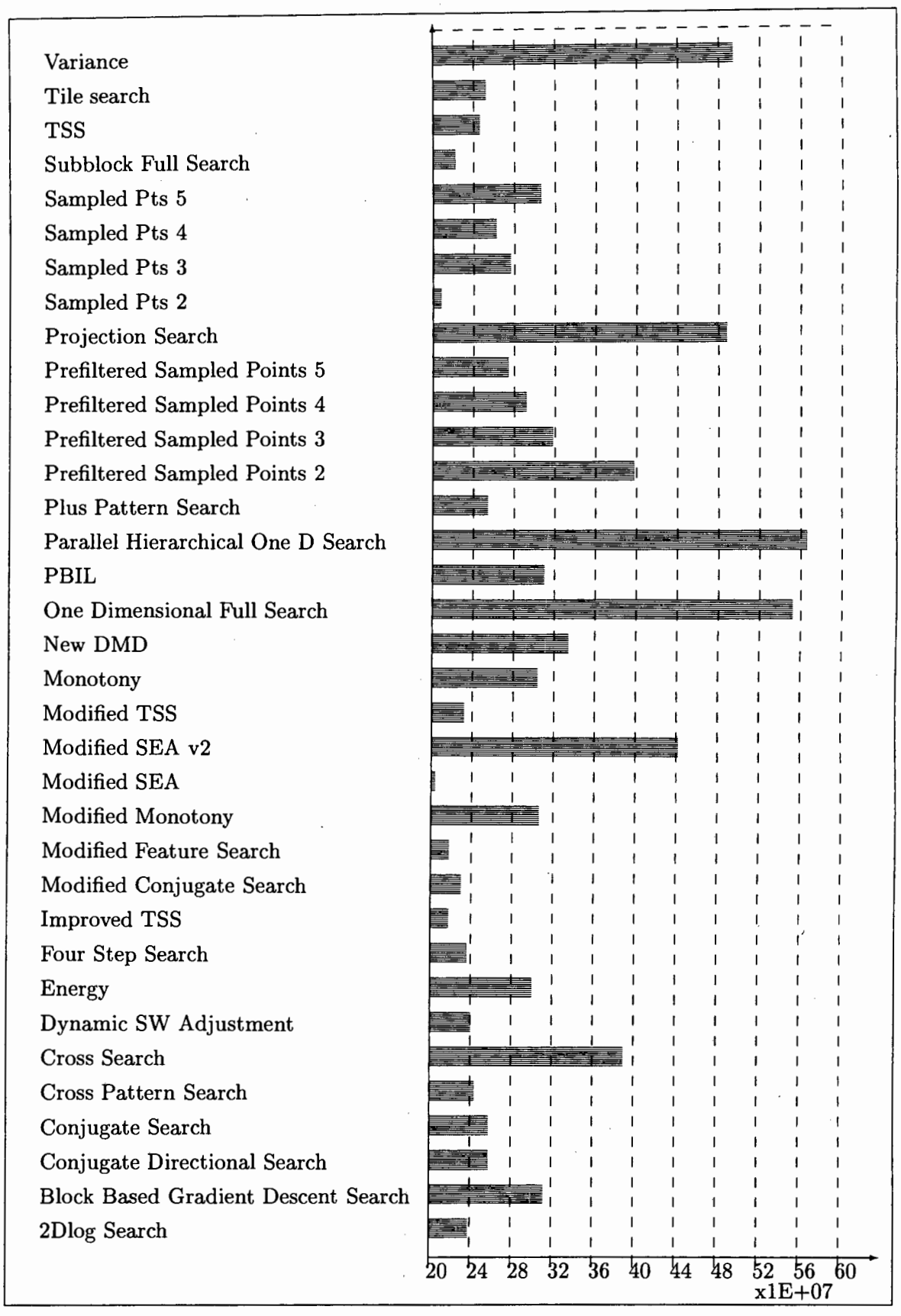


Figure 6.1: **DFD** for sequence **windmill**

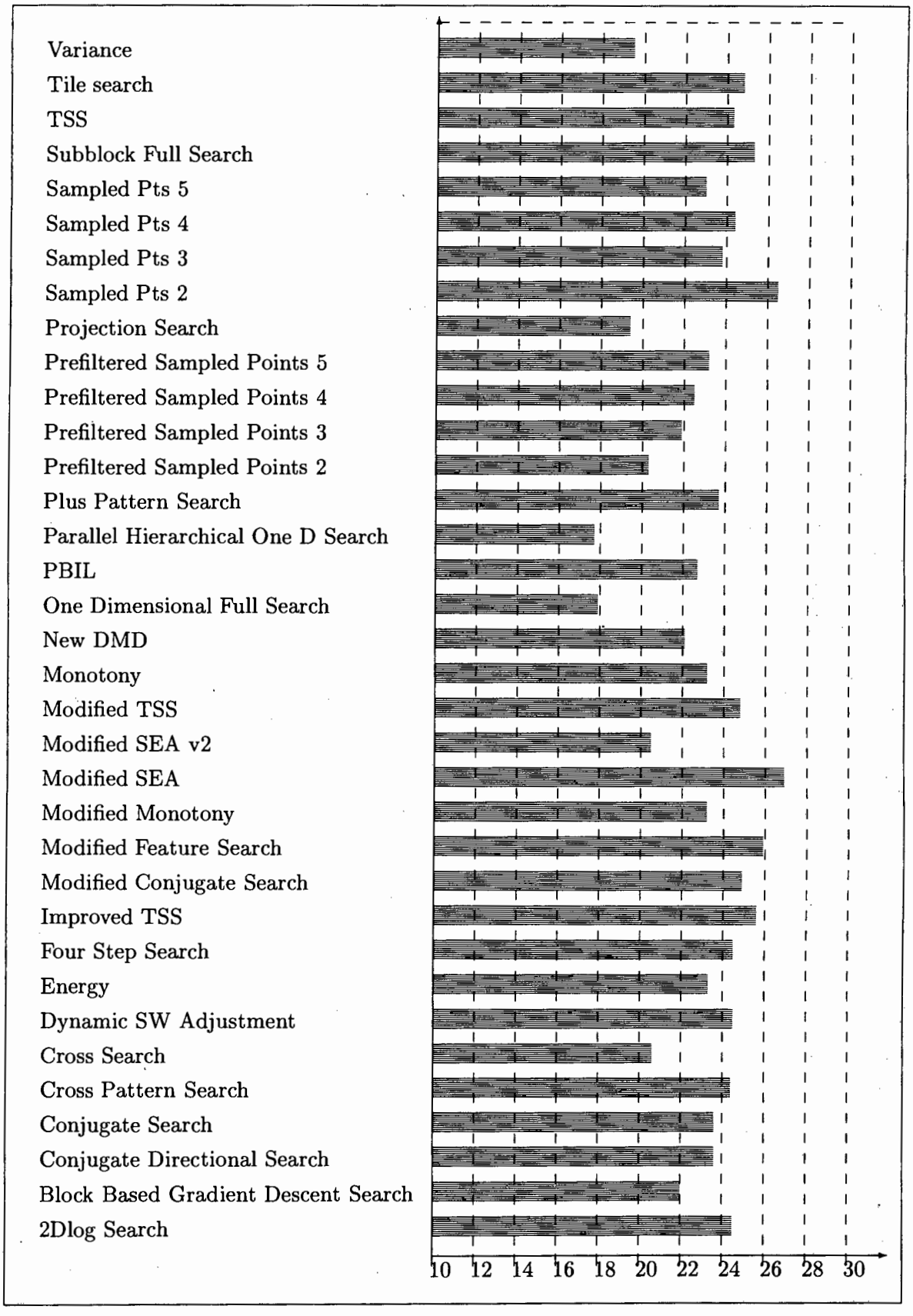


Figure 6.2: **PSNR** for sequence **windmill**

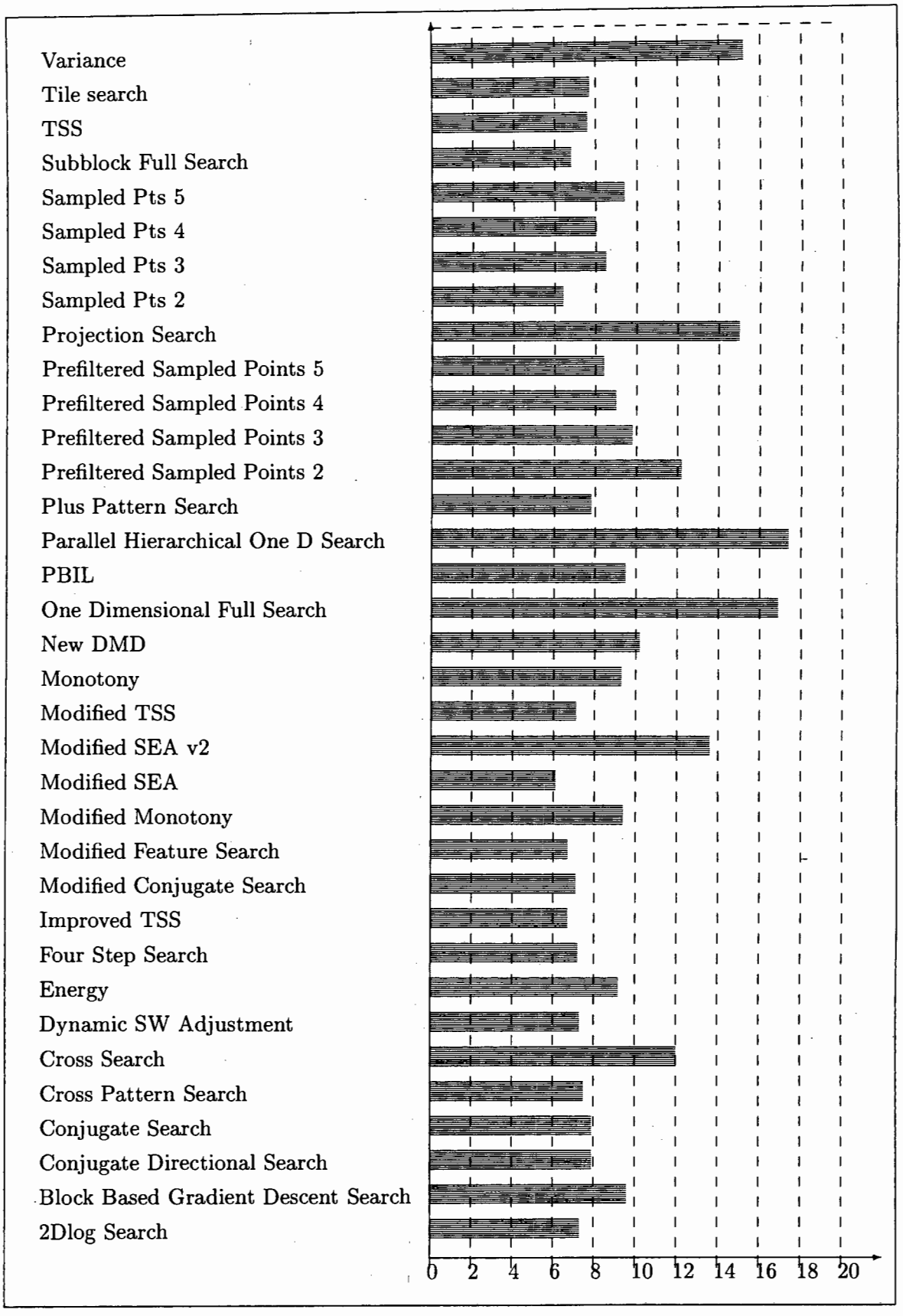


Figure 6.3: **Di abs mean** for sequence **windmill**

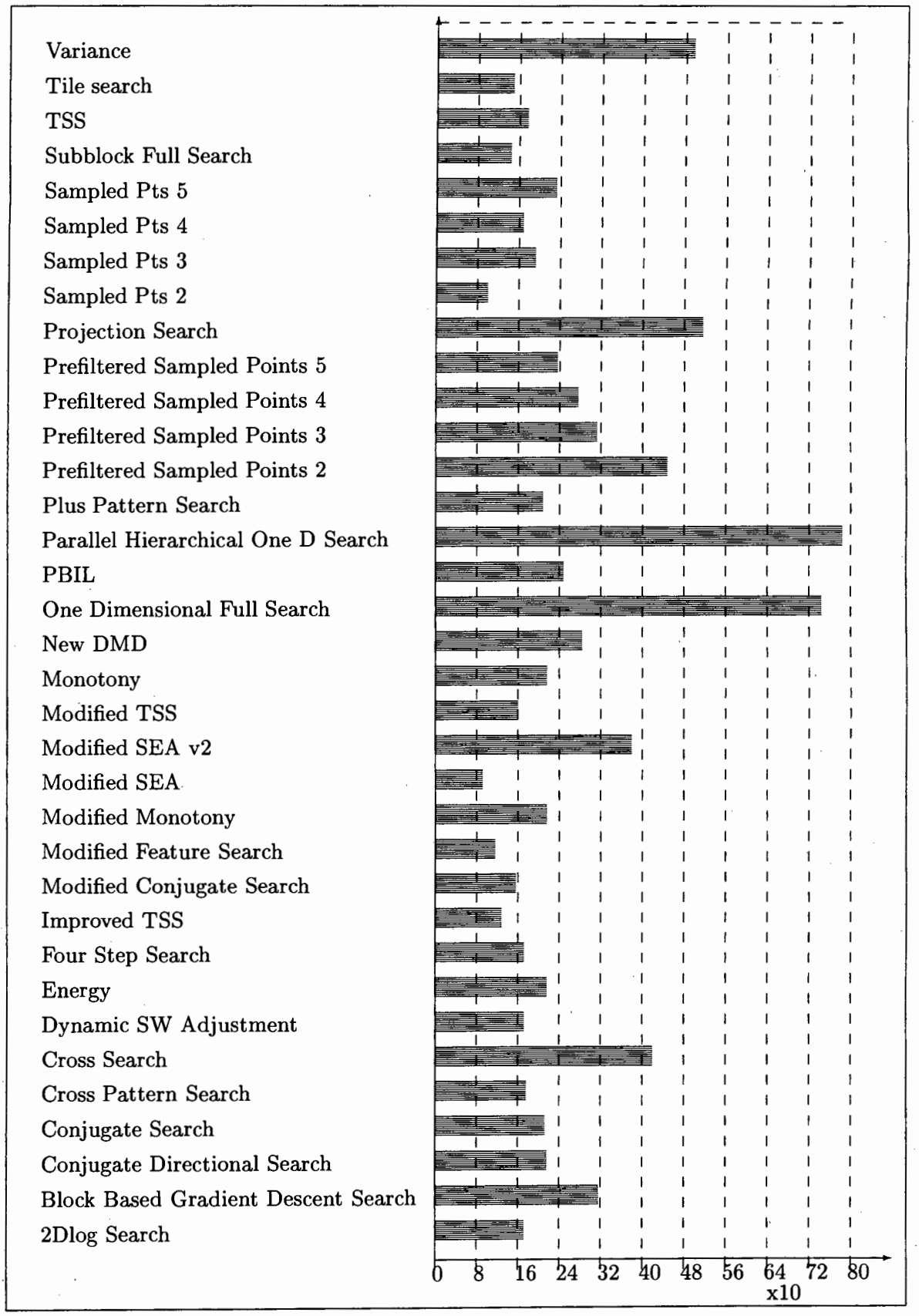


Figure 6.4: **Di abs var** for sequence **windmill**

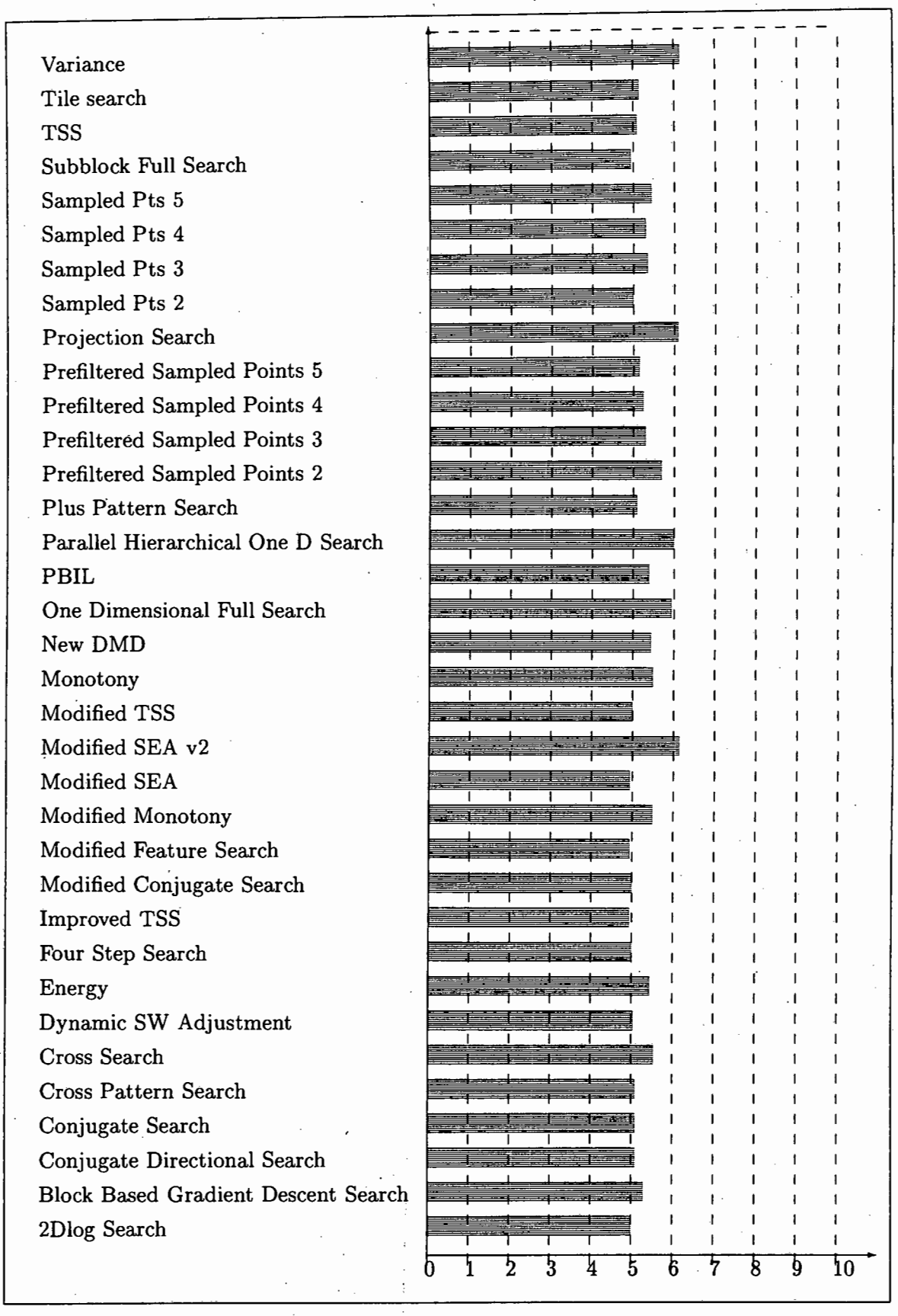


Figure 6.5: **Entropy cumulative** for sequence **windmill**

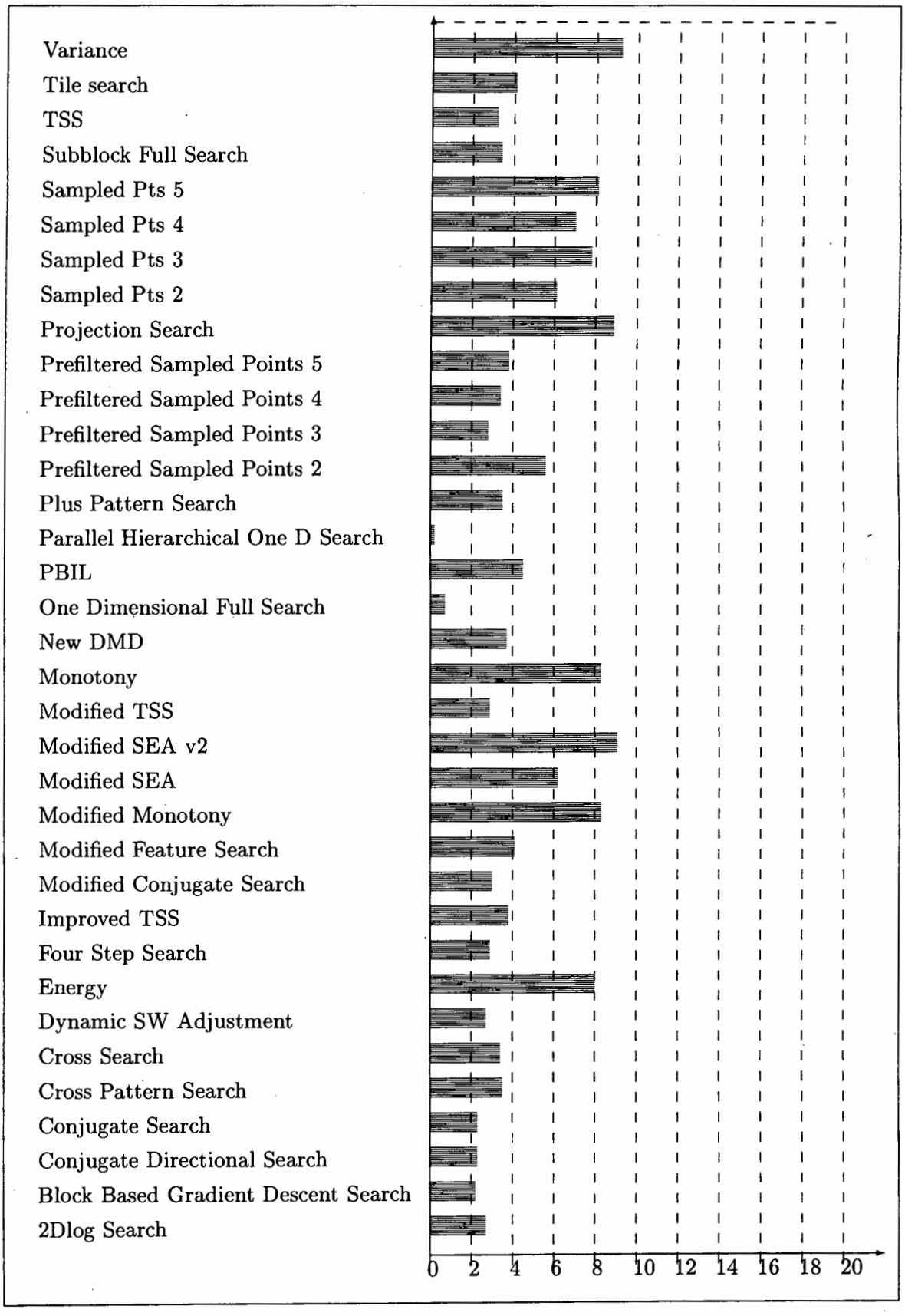


Figure 6.6: **Entropy mtn vec** for sequence **windmill**

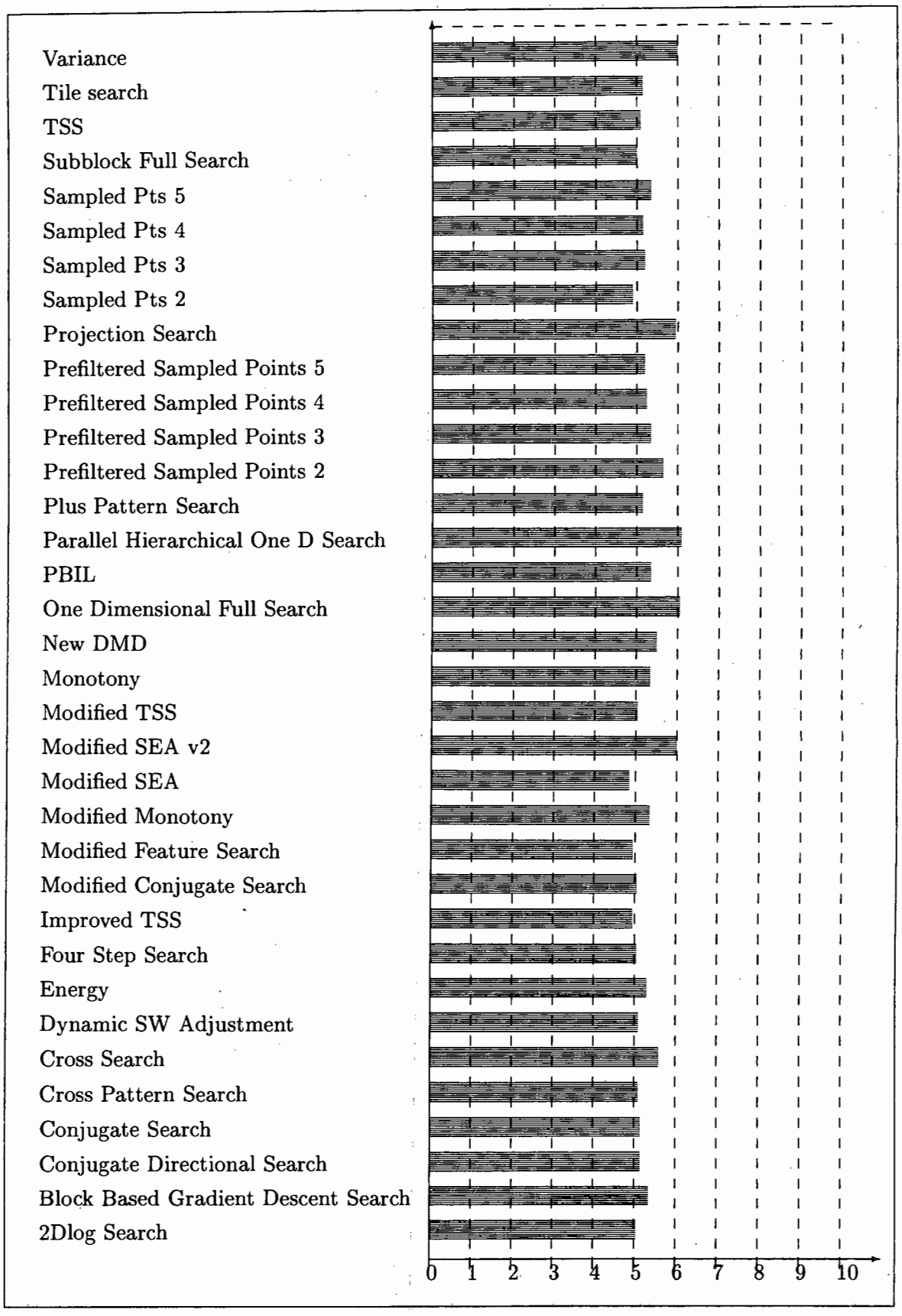


Figure 6.7: **Entropy pixel** for sequence **windmill**

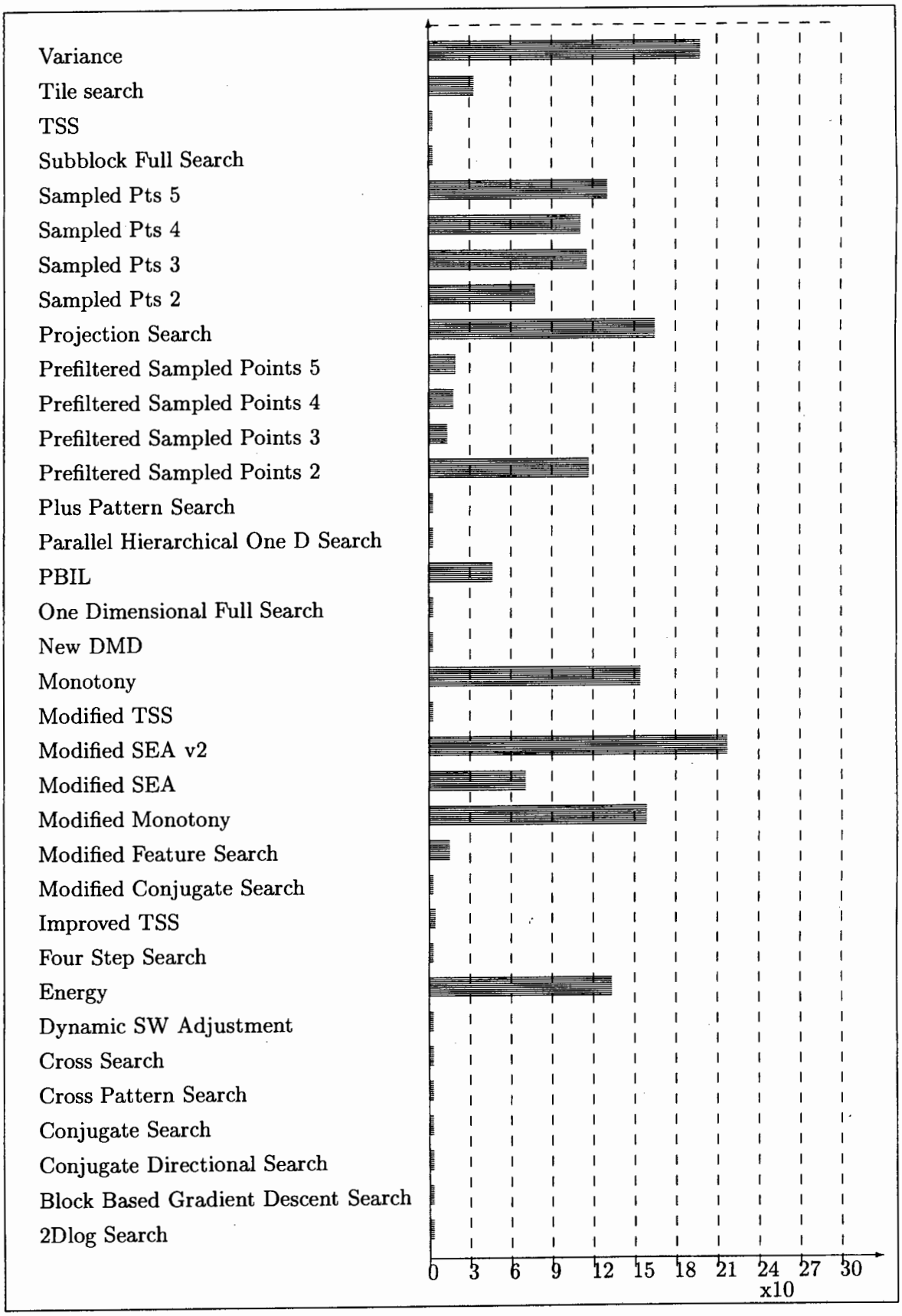


Figure 6.8: **Motion Vector Means** for sequence **windmill**

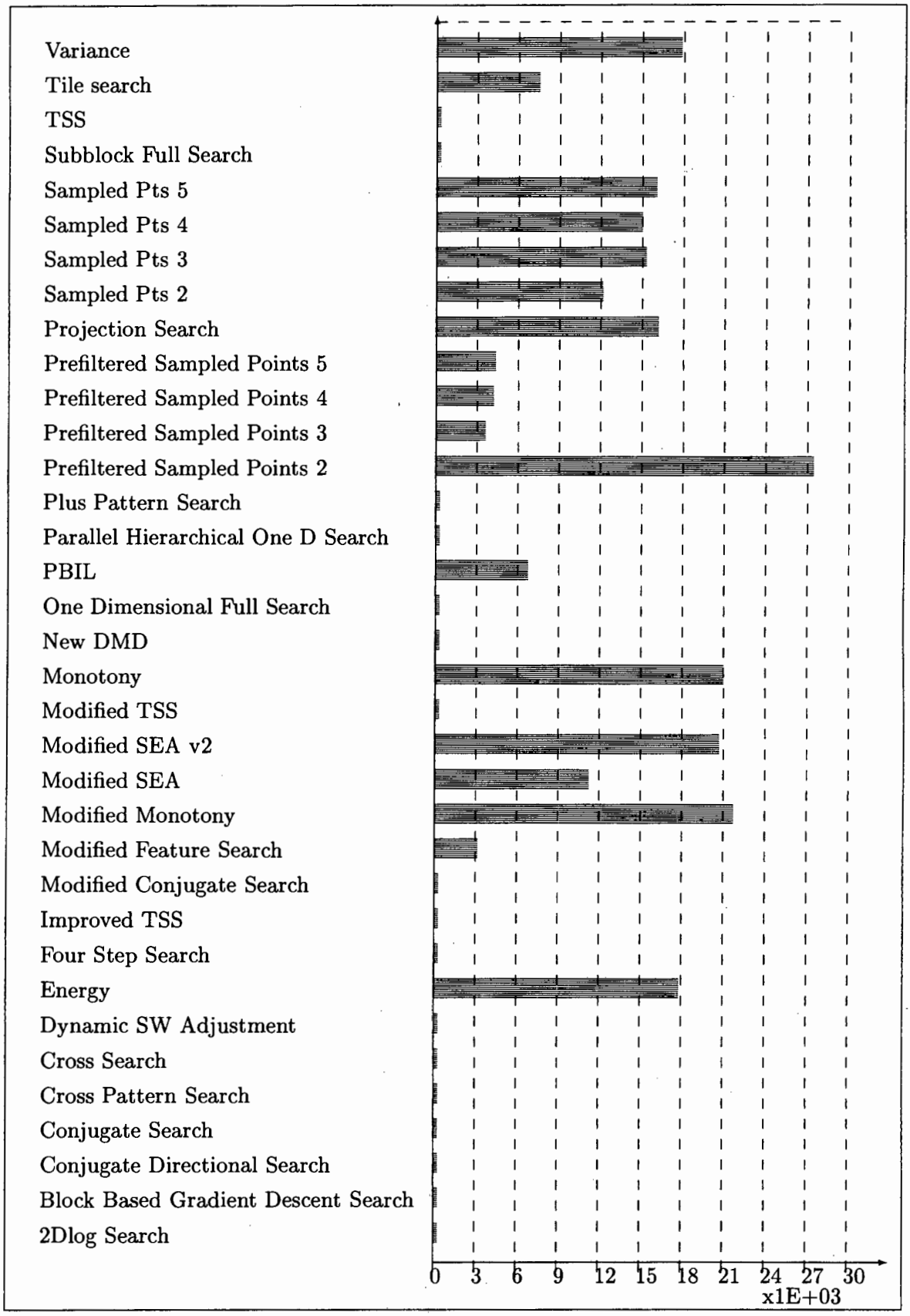


Figure 6.9: **Motion Vector Var** for sequence **windmill**

Chapter 7

Conclusion to Part I
Analysis of the performance of the motion vector algorithms

The previous chapter examined the process of applying the various motion compensation algorithms to the data at hand. This chapter presents an analysis of the results of this application. The performance of the algorithms is examined and the best algorithm found in terms of complexity and performance. The performance measures themselves are examined and the best found. Some conclusions on the results are drawn and finally future work is proposed.

7.1 A physical interpretation of the parameters measured

This section examines the parameters measured on the test sequences by the BBMC algorithms. The parameters are compared with respect to how well they measure the performance of the algorithm and the computational complexity of the calculation of the parameter.

The difference image has negative as well as positive pixel values. The mean of the difference image is therefore, small, and a better estimate of the difference between two frames is the absolute difference. An algorithm with a higher absolute difference than another clearly performs worse, since the algorithms attempt to minimise the difference between the reference and search block. Variance is a useful measure since it provides information about the variability of the difference image and hence the frequency content. Ideally a good algorithm minimises the variance of the difference frame.

The entropy (rounded up) is a measure of the number of bits needed to Huffman code the difference image and Motion Vectors. Ideally an algorithm will reduce the number of bits to code as compared to the original frames. Note that block matching algorithms are really a

modified form of DPCM (Differential Pulse Code Modulation). The combined entropy can be justified as a measure. The entropy for motion vectors depends on the search range of the algorithm and since there are far fewer Motion Vectors than pixels the effect of the motion vectors on the cumulative entropy is small compared to the pixels contribution. Note that the Cumulative and Pixel entropies are similar and thus there is no advantage in coding the pixels and Motion Vectors apart. The Motion Vector entropies are divided into two classes and are fairly similar within those classes. This excludes the entropy of the Motion Vectors and Cumulative entropy as a reasonable measure. The results indicate that the entropy of the pixels may be a reasonable measure and that even a poor motion compensation algorithm may compress data well.

The algorithms may be divided into two classes; those which examine a sub-window around the search origin and those which examine the entire frame. The mean of the Motion Vectors *cannot* then be used to compare these classes of algorithms since the means will be very different, similarly for the variance of the Motion Vectors.

The DFD and the mean of the absolute pixel values of the difference frame correlate (by definition). The PSNR, though expensive to compute, is a good measure and is related to the variance of the difference image. A good algorithm will minimise DFD and maximise PSNR.

As regards computational complexity and robustness, the DFD and variance of the difference image seem to be the best measures.

The algorithms are ranked in order of goodness as regards DFD and variance for algorithms operating over the entire frame and algorithms operating over a sub-region of the frame.

7.2 Comparing the algorithms

The previous section examined the different parameters measured on the test data by the various algorithms. It emerged that the DFD is a good yet simple measure for examining the accuracy of the algorithms. Table 7.2 ranks all algorithms in terms of the total DFD of all eleven test sequences. The percentage increase on the DFD result of the algorithms is calculated as follows:

$$\frac{100 * (DFD_2 - DFD_1)}{DFD_1}$$

The following conclusions can be drawn:

- Filtering a frame before subsampling does not improve the performance of the SEA algorithm.
- The Prefiltered Sampled Points Search separates the effect of the subsampling from the effect of the full search around the best match of the subsampled points. The search of a local neighbourhood of pixels surrounding the best match improves the performance of the Sampled Points Search.

Algorithm	DFD $\sum$ ($x10^8$)	DFD Mean ($x10^8$)	% Increase
Modified SEA	20.731	1.885	—
Sampled Pts 2	21.710	1.974	4.72
Sampled Pts 4	23.671	2.152	9.02
Sampled Pts 3	23.830	2.166	0.65
Energy	24.606	2.237	3.28
Tile search	25.072	2.279	1.88
Sampled Pts 5	25.326	2.302	1.00
Monotony	25.329	2.303	0.04
Modified Monotony	25.401	2.309	0.26
Modified Feature Search	27.930	2.539	0.10
Prefiltered Sampled Points 5	31.429	2.857	12.52
Prefiltered Sampled Points 3	31.434	2.858	0.04
Prefiltered Sampled Points 4	31.453	2.859	0.03
PBIL	31.776	2.889	1.05
Prefiltered Sampled Points 2	33.257	3.023	4.64
Improved TSS	33.878	3.080	1.89
Modified SEA v2	36.557	3.323	7.89
Subblock Full Search	37.443	3.404	2.44
Variance	38.171	3.470	1.94
Projection Search	42.716	3.883	11.9
Cross Pattern Search	43.063	3.915	0.82
TSS	43.521	3.956	1.05
Modified TSS	43.734	3.976	0.51
New DMD	44.140	4.013	0.93
Modified Conjugate Search	44.404	4.037	0.6
Four Step Search	44.837	4.076	0.97
Plus Pattern Search	45.071	4.097	0.52
2Dlog Search	47.546	4.322	5.49
Cross Search	47.711	4.337	0.35
Dynamic SW Adjustment	51.110	4.646	7.12
Conjugate Search	53.933	4.903	5.53
Conjugate Directional Search	53.990	4.908	0.1
Block Based Gradient Descent Search	58.018	5.274	7.46
One Dimensional Full Search	63.066	5.733	8.7
Parallel Hierarchical One D Search	83.370	7.579	32.2

Table 7.1: Comparison of algorithms ranked by DFD.

- The features examined can be ranked in order of performance as follows Energy, Monotony, Modified Monotony and Variance.
- The Tile search performs significantly better than the Projection search.
- With all algorithms there is an inverse relation between speed and performance.
- From the percentage increase data; Sampled Points 4 and Sampled Points 3 are similar in performance; Sampled Points 5, Monotony, Modified Monotony and Modified Feature Search are similar in performance; Prefiltered Sampled Points 5, 3 and 4 are similar in performance; Projection Search is similar in performance to Cross Pattern Search; TSS to Modified TSS; New DMD to Modified Conjugate Search; Four Step Search to Plus Pattern Search; 2Dlog Search to Cross Search; and Conjugate Search to Conjugate Directional Search.
- For a number of searches yielding similar results the computationally simpler search is preferred, e.g., Sampled Points 4 is preferable to Sampled Points 3.
- Algorithms which cover a large search area perform substantially better than localised search area algorithms, (e.g., Three Step Search) although they are slower.
- Given the tradeoff between algorithm complexity (and hence computational time) and performance it appears best to use algorithms examining a portion of the search space (e.g., Three Step Search) if the motion in the video is expected to be small (e.g., for teleconferencing) and to algorithms such as SEA when the motion expected is large (e.g., sports).
- It should be noted that BBMC may not yield motion vectors corresponding to the actual motion of the pixels. This is not only the result of the aperture problem, but in the case of the algorithms examining the entire search space (e.g., SEA), the best match block generated may be physically impossible for the search block to have reached.
- The good correlation between algorithms with low DFD and high PSNR (PSNR calculation uses MSE) implies experimental verification of similar performance of MAD and MSE reported in the literature [41].
- The similar cumulative and difference image entropies indicate that careful coding of the compressed data can compensate for poor performance motion compensation algorithms.

7.3 Proposed future work

Given the overview of the BBMC algorithms presented, and the performance of these algorithm examined in this and the previous chapter, some improvements to these algorithms can

be suggested. An optimum algorithm must examine all possible locations (or search blocks) for an optimum match. To speed up the search, the use of features with good discrimination is proposed. Pre-calculating the features of each block and cataloguing the reference blocks in terms of their features allows an algorithm to only examine the reference blocks which closest resemble the search block.

More discriminant features for the Tile and Projection searches need to be found. Examination of the hash function for Projection Search to optimise memory use and performance is required.

It should be noted that optimisation of the motion compensation process in a practical encoder occurs in conjunction with other bit-rate optimisation processes (e.g., quantisation), and joint vs. separate optimisation of these processes may yield different results. The effect of quantisation on motion vector searches is examined in Chapter 8, an appendix to Part I. The different algorithms for measuring distortion are compared in Chapter 9, an appendix to Part I. The issues of complexity are examined in Part II.

Chapter 8

The effect of quantisation on motion compensation

Chapter 2 examined the MPEG-2 standard and the quantisation process. The quantisation process provides part of the compression ability of MPEG-2 by discarding low order bits thought to be redundant, without impacting the image quality. This chapter demonstrates that quantisation could possibly change the best match found on the unquantised data by the MAD measure and proposes a modification of the MAD and MSE measures of distortion which will find a best match unchanged (or at least minimally affected) by quantisation.

It is computationally simple to perform motion compensation on unquantised data (the macroblocks in MPEG-2) since quantisation involves a transformation to the DCT domain (see Chapter 2). To achieve best possible motion compensation results, the reference frame and the frame under consideration for motion compensation should be quantised since the final end-user images are quantised. It is unclear how overlapping blocks are to be quantised without a forward and inverse DCT transform as part of the motion compensation process.

If quantisation changes the location of the best match, (i.e., a better match is found elsewhere after quantisation) then quantisation must be performed before motion compensation. If quantisation does not change the location of the best match, (i.e., find a new best match) then motion compensation can be performed without considering quantisation.

This chapter first demonstrates that quantisation possibly changes the best match, then proposes a modification to the MAD and MSE functions to provide some immunity to quantisation and finally concludes with a summary of the chapter.

8.1 Demonstration that quantisation possibly changes the best match

This section demonstrates that quantisation could change the best match from examination of the quantisation process and some geometry. Symmetry allows the problem to be reduced to a more manageable two dimensional problem.

Now th quantisation process is as follows:

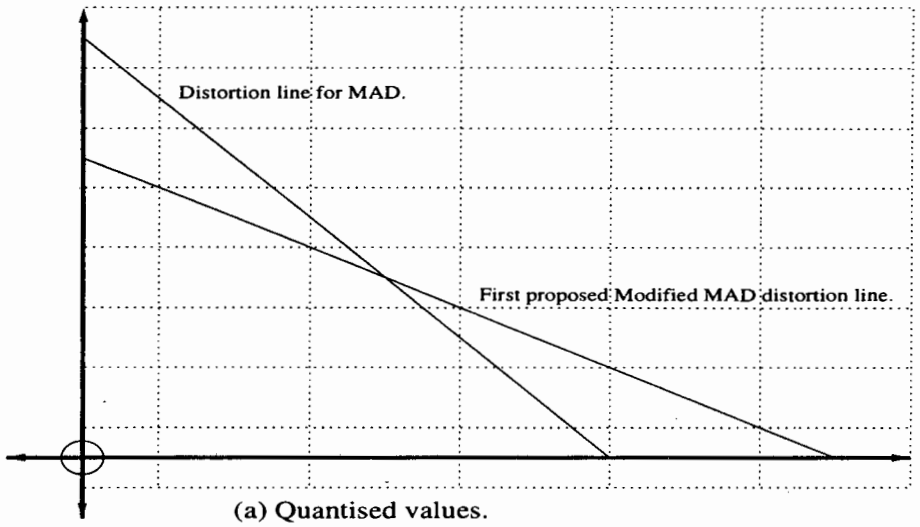
$$
\begin{aligned}
F_Q(u,v) &= floor(\frac{F(u,v) + \frac{Q(u,v)}{2}}{Q(u,v)}) \\
MAD &= \sum_{i=0}^{N-1} \mid a_i - b_i \mid \\
MAD &= \sum_{i=0}^{N-1} \mid a_i \mid \cdots \text{shifting origin.} \\
MAD &= \sum_{i=0}^{N-1} a_i \cdots \text{positive values only.} \\
\text{Let } R(u,v) &= F_Q(u,v)Q(u,v)
\end{aligned}
$$

Note $F(u,v)$ is the Discrete Cosine Transform (DCT) of a block, $Q(u,v)$ represents the quantisation coefficients, and $F_Q(u,v)$ is the quantised value of $F(u,v)$. It is convenient to consider the quantisation and inverse quantisation (QIQ) process as the mapping of $F(u,v)$ onto $R(u,v)$ denoted as $QIQ(F(u,v)) = R(u,v)$, with $R(u,v)$ the inverse quantisation process.

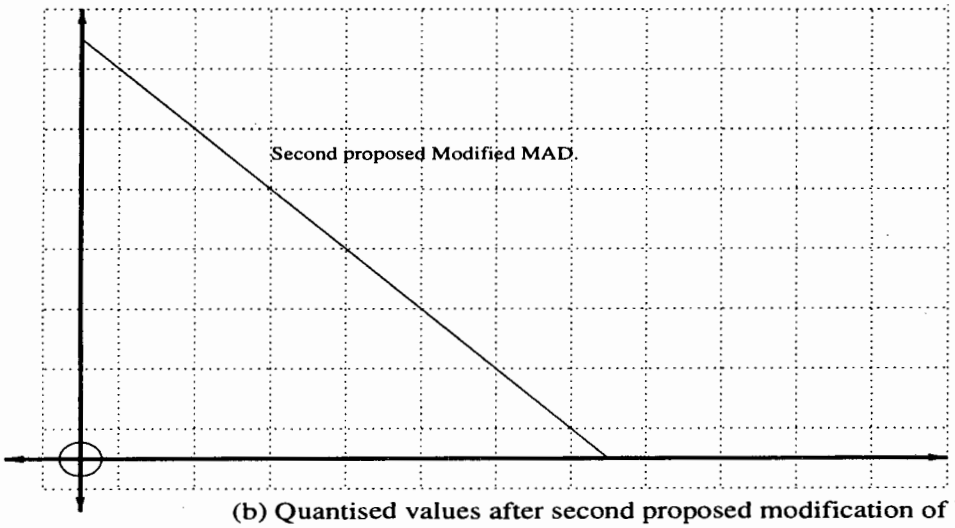
Symmetry considerations allow one to examine positive values for a_i and a_j only (with $i \neq j$). By positioning a plane parallel to one of the axes and located some distance (a multiple of the quantisation coefficient for that frequency) along a perpendicular axis, the MAD reduces to $MAD = X + a_i - a_j$, let $P = a_i - a_j$. Because the QIQ is different for each frequency, the spacing of the QIQ data along the one axis is different from the other. If quantisation/QIQ does not affect the best match for any slice of the distortion solid then quantisation/QIQ does not affect the best match for the distortion solid as a whole (symmetry).

If one considers simple QIQ of a single variable x, then after quantisation the values of x are multiples of the quantisation coefficient q. All values of x for $nq \leq x < (n+1)q$ collapse to nq. The usual quantisation adds $q/2$ to x which implies $nq - q/2 \leq x < (n+1)q - q/2$ collapses to nq. Geometrically a line segment collapses to its centre point.

QIQ can be thought of as collapsing a hyper-rectangle onto its centre for different possible QIQ values. In two dimensions this is simply a rectangle. The DCT is an orthogonal transform which implies a_i is perpendicular to a_j, and by inspection $P = a_i - a_j$ forms a line (the distortion line) in two dimensions, with a_i and a_j on the axes. A better match is possible if and only if for some QIQ values the rectangle protrudes above the distortion (MAD) line



(a) Quantised values.



(b) Quantised values after second proposed modification of MAD.

Figure 8.1: Quantisation effects with MAD and its modifications.

with the centre below the line. If a point (different from the best match), which before QIQ had a larger distortion (MAD/MSE) than the best match found without QIQ, moves after QIQ to the centre of its QIQ hyper-rectangle and this centre is closer to the search point than the unQIQ'ed best match (smaller distortion than the unquantised best match) then a better match is possible after QIQ (see Fig. 8.1).

If the quantisation coefficients are the same for both frequencies then the QIQ shape is a square. Clearly then the distortion line passes through both the centre and the upper left and lower right corners of squares on the line, and no square with centre below the MAD line can be found protruding above the line. This implies QIQ could change the best match provided each frequency is differently quantised, since the slope of a line passing through the centre and upper left and lower right corners of a rectangle cannot have a slope of -1.

8.2 Modification of MAD to prevent the possibility of a better match after quantisation

The quantisation process is an unavoidable part of MPEG-2. Making the motion compensation results QIQ invariant requires a change in the distortion measurement function, since this is under the control of the designed of the encoder. This section proposes a modification to the MAD function and justifies this modification.

Differently quantised frequencies form a QIQ hyper-rectangle around every possible QIQ value and allow for the possibility of a better match after QIQ. Hyper-squares, on the other hand, do not allow for the possibility of a better match after QIQ. Adjusting the distortion solid, then, could remove the possibility of a better match after the QIQ process.

It is proposed that the following be used:

$$New\_MAD = \sum_{i=0}^{n-1} a_i \mid x_i - y_i \mid$$

Modification 1 Where a_i is the product of the quantisation coefficients divided by the quantisation coefficient of that frequency. x_i and y_i are unquantised. This to an extent undoes some of the effect of quantisation. Each frequency is weighted the same and ensures that the distortion (MAD) line passes through the upper left and lower right corners, and centre of the distortion rectangle (see Fig. 8.1(a)). Effectively the old MAD definition line is tilted to pass through the upper left and lower right corners, and centre of the distortion rectangle.

Modification 2 Where a_i is the reciprocal of the quantisation coefficient and x_i and y_i are unquantised. It can be readily seen that in a two dimensional slice the QIQ shape is a square which implies the best match is QIQ independent (see Fig. 8.1(b)).

Closer examination of the two modifications shows they are essentially the same. Both scale the hyper-rectangles to hyper-cubes.

The modifications can be justified for other reasons:

- Application of the standard MAD function to motion compensation results in a point on the surface with minimum distortion. QIQ results in the distortion surface passing through points where each frequency is a multiple of the quantisation coefficient for that frequency, e.g., $(n_1 q_1 n_2 q_2 \cdots n_{k-1} q_{k-1} n_k q_k)$ with n_i some integer and q_i the quantisation coefficient for frequency i. The distortion surface then expands or contracts, i.e., the best match MAD (distortion) becomes larger or smaller. The modification of the MAD proposed ensures that after QIQ the best match does not change since the new distortion surface ensures that no quantisation hyper-rectangle with a centre *within* the distortion surface has any portion outside the distortion surface.

- *MAD modification 1* is less affected by finite arithmetic than *MAD modification 2* since it is a multiplication rather than a division.

- For *MAD modification 2* the differences $| x_i - y_i |$ are weighted in proportion to the energy per frequency; lower frequencies contain more energy and are less severely quantised.

- The modified version of MAD needs fewer computations per difference: three operations for the modified MAD compared to six for first QIQ and then finding the difference. The process of quantisation can be delayed until the difference blocks are coded.

8.3 Modification of MSE to prevent the possibility of a better match after quantisation

The quantisation process is an unavoidable part of MPEG-2. Making the motion compensation results QIQ invariant requires a change in the distortion measurement function. This section, similarly to the previous section, proposes a modification to the MSE function.

Quantisation results in the space of possible search and reference blocks being partitioned into hyper-rectangles. All points within the hyper-rectangle after QIQ take on the value of the point within the hyper-rectangle. For a distortion measurement function to be quantisation-invariant the surface of the distortion solid (the distortion measurement function in multiple dimensions) must pass through the centres of any block it passes through. This prevents the situation where a hyper-rectangle with centre within the distortion solid protrudes above the distortion surface. The implication i s that after quantisation any point within this hyper-rectangle but outside the distortion solid has a better distortion after quantisation.

It is proposed that the following be used:

$$New\_MSE = \sum_{i=0}^{n-1} a_i^2 (x_i - y_i)^2,$$

where a_i is the reciprocal of the quantisation coefficient and x_i and y_i are unquantised. It can be readily seen that in a two dimensional slice the QIQ shape is a square which implies the best match is QIQ-independent.

$$\begin{aligned} \text{Let } \frac{x^2}{\alpha^2} + \frac{y^2}{\beta^2} &= R^2 \\ y^2 &= \beta^2 (R^2 - \frac{x^2}{\alpha^2}) \end{aligned}$$

$$\begin{aligned} \text{Let } x &= q\alpha \\ y^2 &= \beta^2 (R^2 - q^2) \end{aligned}$$

y is then integer valued. Note that the proposed new MSE is an ellipsoid. Scaling each difference by the reciprocal of the quantisation coefficient ensures that the surface of the ellipsoid passes through the centres of any hyper-cube it passes through. The only provision is that $q \leq R$.

8.4 Summary

Although quantisation/QIQ may change the best match found by a search in the unquantised domain, a modification of the MAD/MSE distortion function can allow a search in the unquantised domain which is minimally affected by quantisation. The primary limitation is that the search must be carried out in the DCT domain. This has the advantage that for 8x8 blocks the DCT coefficients are close to perfectly decorrelated [21], i.e., the DCT coefficients can be used as decorrelated features and only a subset used for a computationally simpler distortion function.

Chapter 9 examines the distortion measurement functions used in motion compensation.

Chapter 9

A comparison of the distortion functions used for Motion Compensation

The literature of Motion Compensation describes three functions (the mean of the absolute differences (*MAD*), the mean of the squared differences (*MSE*), and the cross correlation) for measuring the match between the current macroblock (or search block) and the candidate best fit macroblock on the reference frame. Simulation results show *MAD* and *MSE* perform very closely [41], [50].

This chapter examines the three distortion functions and attempts to compare their performance. First the relationship between MSE and cross-correlation is examined and they are demonstrated to be closely related (cross-correlation is suggested on computational grounds). Next the relationship between MSE and MAD is examined and the conditions for similar performance described. The measures are:

$$
\begin{aligned}
MSE(i,j) &= \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N}[f_k(m,n) - f_{k-1}(m+i,n+j)]^2 \\
MAD(i,j) &= \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N} \mid f_k(m,n) - f_{k-1}(m+i,n+j) \mid \\
correlation(i,j) &= \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N} f_k(m,n) f_{k-1}(m+i,n+j)
\end{aligned}
$$

9.1 The relationship between *MSE* and cross-correlation

It will be shown that *MSE* and cross-correlation are related and that given one the other is computable. The computational complexity of cross-correlation and MSE is examined.

$$
\begin{aligned}
MSE(x,y) &= \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N}[\, f_k(m,n) - f_{k-1}(x+m,y+n)\,]^2 \\
&= \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N}[\, f_k^2(m,n) - 2f_k(m,n)f_{k-1}(x+m,y+n) + \\
&\qquad f_{k-1}^2(x+m,y+n)\,]
\end{aligned}
$$

Note now that,

- $\frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N} f_k^2(m,n)$ is the average energy contained in the reference block.
- $\frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N} f_{k-1}^2(x+m,y+n)$ is the average energy contained in the candidate block.
- $\frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N} f_k(m,n)f_{k-1}(x+m,y+n)$ is the cross-correlation of the search macroblock and the reference frame.

The relationship between the extrema of *MSE* and the cross-correlation of the reference and search (macro)blocks is examined.

9.1.1 The relationship between the maxima and minima of the *MSE* and cross-correlation functions

It will be shown that the maxima of the cross-correlation function and the minima of the MSE function coincide.

Let the following hold:

- $A = \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N} f_k^2(m,n)$.
- $B(x,y) = \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N} f_{k-1}^2(x+m,y+n)$.
- $CCorr(x,y) = \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N} f_k(m,n)f_{k-1}(x+m,y+n)$.

The maxima and minima of the *MSE* are located where the derivative with respect to x and y is zero:

$$
\begin{aligned}
MSE(x,y) &= A + B(x,y) - 2CCorr(x,y) \\
\frac{\partial}{\partial x}MSE(x,y) &= \frac{\partial}{\partial x}B(x,y) - 2\frac{\partial}{\partial x}CCorr(x,y) \\
&= 0 \\
\frac{\partial}{\partial y}MSE(x,y) &= \frac{\partial}{\partial y}B(x,y) - 2\frac{\partial}{\partial y}CCorr(x,y) \\
&= 0 \\
\frac{\partial}{\partial x}CCorr(x,y) &= \frac{1}{2}\frac{\partial}{\partial x}B(x,y)
\end{aligned}
$$

$$\frac{\partial}{\partial y}CCorr(x,y) = \frac{1}{2}\frac{\partial}{\partial y}B(x,y)$$

The same is true for the cross-correlation function, so:

$$\begin{aligned}
MSE(x,y) &= A + B(x,y) - 2CCorr(x,y) \\
CCorr(x,y) &= \frac{1}{2}[A + B(x,y) - MSE(x,y)] \\
\frac{\partial}{\partial x}CCorr(x,y) &= \frac{1}{2}[\frac{\partial}{\partial x}B(x,y) - \frac{\partial}{\partial x}MSE(x,y)] \\
&= 0 \\
\frac{\partial}{\partial y}CCorr(x,y) &= \frac{1}{2}[\frac{\partial}{\partial y}B(x,y) - \frac{\partial}{\partial y}MSE(x,y)] \\
&= 0 \\
\frac{\partial}{\partial x}MSE(x,y) &= \frac{\partial}{\partial x}B(x,y) \\
\frac{\partial}{\partial y}MSE(x,y) &= \frac{\partial}{\partial y}B(x,y)
\end{aligned}$$

The zero-crossings of the derivatives (or positions of extrema of the original) of both *MSE* and cross-correlation are in the same positions, since the derivatives depend on the same function $B(x,y)$ (note that scaling a function does not change its zero-crossings). Clearly then, *MSE* and cross-correlation functions potentially do yield the same results, i.e., find the reference block most similar to the search block. Maximising the cross-correlation minimises the *MSE*. This explains why the *MSE* function is used in the literature more than the cross-correlation.

Examining the computational complexity of *MSE* and cross-correlation leads to:

- For the *MSE* function per pair of corresponding pixels on the search and reference blocks:

  1. Operand fetch. The two pixel values must be fetched from memory.
  2. One Subtraction operation.
  3. One Multiplication operation.

- For the cross-correlation function per pair of corresponding pixels on the search and reference blocks:

  1. Operand fetch. The two pixel values must be fetched from memory.
  2. One Multiplication operation.

The cross-correlation function has one fewer instruction and is thus easier to compute than the *MSE* distortion measurement function.

9.2 The relationship between *MAD* and *MSE*

The relationship between MSE and MAD is examined and conditions under which the results are similar is examined. The definitions of *MSE* and *MAD* show that the *MSE* function is a hyper-sphere centred on some point and *MAD* a hyper-cube centred on some point. From symmetry considerations only the positive valued variables need be considered (in two dimensions this corresponds to the positive quadrant).

Considering each pixel within a block as a variable, and regarding each pixel difference as a variable the definitions for *MAD* and *MSE* can be rewritten as:

$$\begin{aligned} MSE &= \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N}[y_{(m,n)}]^2 \\ MAD &= \frac{1}{MN}\sum_{m=1}^{M}\sum_{n=1}^{N}y_{(m,n)} \end{aligned}$$

Without loss of generality the scale factor $\frac{1}{MN}$ can be ignored. Any *MAD* value has the following range of *MSE* values associated with it:

- The largest value of *MSE* is for the *MAD* hyper-cube contained inside the *MSE* hyper-sphere. The largest *MSE* value is $MSE = MAD^2$.

- The smallest value of *MSE* is for the *MSE* hyper-sphere contained inside the *MAD* hyper-cube. The *MAD* plane at the points of contact is tangential to the sphere. This implies the line from the origin to the point of contact is orthogonal to the *MAD* plane.

  Any plane in n-dimensions can be described by a vector normal (orthogonal) to the plane and a vector on the plane [52], i.e., $v - u$ where v is an arbitrary point and u is some reference point on the vector. Note $a \cdot b$ represents the inner (dot) product of a and b. The normal and the vector are orthogonal by definition. A plane can be then described as follows $n \cdot (v - u) = 0$ or $n \cdot v = n \cdot u$, since n and u must be known to describe the plane, $n \cdot v = A$. This is essentially the equation for the *MAD* distortion function at the point of contact of the MAD and MSE functions. Note than n is a vector of ones $(1, 1, \ldots, 1, 1)$.

  The line segment from the origin to the point of contact of the hyper sphere and the *MAD* plane (this is the radius of the sphere) is parallel to the normal of the *MAD* plane. Any line can be described as $y = \lambda n + p$ [52] where p is any point on the line, λ an arbitrary scale factor and n a vector parallel to the line, i.e., perpendicular to the direction in which the line runs. For the radius $p = 0$:

$$\begin{aligned} MSE &= R^2 \\ &= (\lambda n)^2 \end{aligned}$$

$$= \lambda^2(n \cdot n)$$

$$\begin{aligned} MAD &= n \cdot x \\ &= n \cdot \lambda n \\ &= \lambda(n \cdot n) \\ \lambda &= \frac{MAD}{n \cdot n} \end{aligned}$$

$$\begin{aligned} MSE &= (\frac{MAD}{n \cdot n})^2(n \cdot n) \\ &= \frac{A^2}{n \cdot n} \end{aligned}$$

Now $(n \cdot n)$ is a scalar.

The upper and lower bounds on the *MSE* are as follows:

$$\begin{aligned} Lower\ bound\ MSE &= (\frac{MAD}{n \cdot n})^2(n \cdot n) \\ &= \frac{A^2}{n \cdot n} \\ Upper\ Bound\ MSE &= A^2 \\ Thus \quad \frac{A^2}{n \cdot n} \leq\ MSE\ &\leq A^2 \end{aligned}$$

where A is the *MAD*, and n is a vector of ones$(1, 1, \ldots, 1, 1)$. Ensuring that the *MSE* and *MAD* values are similar implies minimising $n \cdot n$ or the number of variables (differences) used, since $(n \cdot n) = \sum_{i=1}^{q} 1 = q$. Minimising the number of variables used implies the use of a set of features accounting for most of the variability of the original data. Provided the differences are sufficiently small, *MSE* and *MAD* should perform similarly.

Part II

Objective measures of the complexity of an image sequence

Chapter 10

An overview of Part II

Successful encoding of a video sequence depends on delivering an acceptable image quality to the end user. This requirement is balanced by the need to minimise the bit-rate to maximise the utilisation of the available bandwidth. The bit-rate is constrained by the quality and the complexity of the video sequence. Complexity in this context can be reasonably defined as a measure of complicated motion or temporal complexity (i.e., many objects moving in different direction with different types of motion such as rotation and linear displacement) and complex image content or spatial complexity. An image of a clear sky can be considered to have low spatial complexity as compared to an aerial photograph of a city which can be considered to have a high spatial complexity. Spatial complexity can thus be related to the detail in the image. The total complexity of a sequence is then a combination of the spatial and temporal complexities, since the origins of the two complexities are different they are not likely to be correlated.

Acceptable video quality relies on the spatial and motion information being reproduced at an acceptable level (or the errors/distortions of the spatial and temporal information being small enough or non-noticeable). Both these requirements depend on the Human Visual System, for human end-users, and machine requirements for computer/robot vision requirements. These requirements translate to retaining sufficient information from the original (assumed to be of a higher quality than the reproduced sequence) in the reproduced image, which in turn implies a bit-rate (or number of bits) sufficient to encode this information. Quality and bit-rate are then related.

Complexity is related to the amount of information required to reproduce the same or similar spatial and temporal characteristics. This implies that complexity and quality are closely related (see Figure 10.1 for an illustration of this). High bit-rates are required to match the complexity and quality of a compressed video sequence to the original video sequence.

This chapter presents an overview of Part II. Part II examines the relationship between complexity, quality and the final bit-rate of an image or image sequence. It is proposed that the bit-rate (peak or average) is a measure of the complexity of an image sequence from the

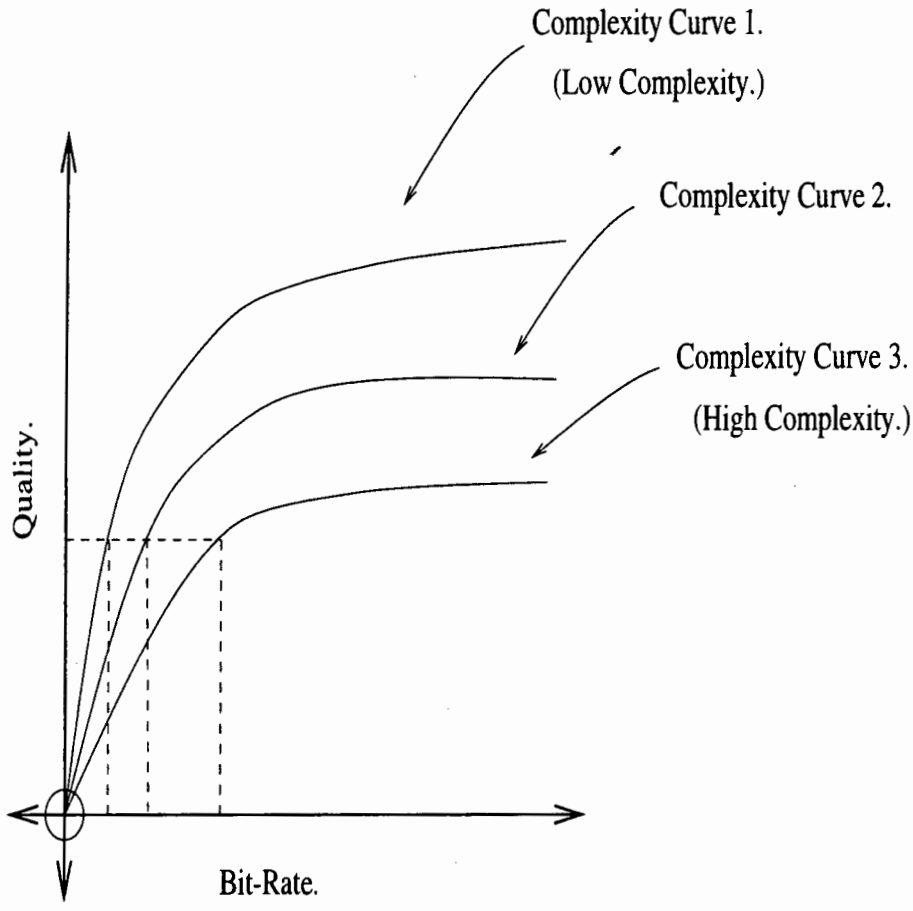


Figure 10.1: An illustration of the relationship between quality and bit-rate for different complexity values.

perspective of the MPEG-2 encoder. It seems reasonable to propose that a complex sequence requires more bits (a higher bit-rate) to encode than a simple image sequence for a given image sequence quality. Similarly a high quality image sequence requires more bits than a low quality sequence of the same complexity.

Video sequence complexity has bearing on the compressibility of a sequence. It seems reasonable to suppose that a sequence with complex motion and spatial properties will not compress as well as a sequence with simple motion and image content. A clear distinction must be made between what a human observer will regard as complex and what a compression algorithm will have difficulty compressing. An attempt will be made to find a model for the resultant peak bit-rates in terms of the motion and image characteristics of several sequences compressed by MPEG-2.

The chapters in Part II examine the following:

- Chapter 11 presents a literature overview of video sequence quality measurement, and relates quality to complexity and hence bit-rate.
- Chapter 12 presents an overview of the process of measuring complexity or variables which correlate with complexity.
- Chapter 13 presents an overview of the process of modelling the bit-rate.
- Chapter 14 presents the results of the model fitting process.
- Chapter 15 examines the effect of Principal Components Analysis and Jackknifing on the linear model.
- Chapter 16 concludes Part II by comparing the proposed linear and quadratic models. Some weaknesses of the model fitting approach are examined as well.

Chapter 11

An overview of current approaches to Complexity

This chapter provides a definition of complexity and examines some current literature on the relationship between subjective and objective image and image sequence quality. The similarities between the complexity and quality is examined.

A current focus of study is the prediction of human estimation and perception of video sequence quality from certain objective variables within the sequence. Such models are useful for maximising the perceived quality of received video while minimising the compressed bit-rate [24]. The loss of quality as a function of impairment is related to the complexity of the sequence, since the effect of impairment is more noticeable for complex sequences.

Complexity is a function of the temporal and spatial activity within a sequence. Clearly a sequence with different motion in magnitude and direction is more complex than simple panning of a static scene. Similarly, a static scene of a street map is more complicated than a static image of a brick wall. High spatial and temporal frequency magnitudes relate to high complexity.

Two approaches from the literature are examined: model based prediction of quality by Voran and Wolf [53], and Human Visual System based modelling by Davies et al. [13]. The relationship between quality and complexity is then explored.

11.1 Model based prediction of quality

A model of the relationship between subjective and objective image quality is attempted (proposed by Voran and Wolf [53]). Of particular interest is the features used.

The test data was corrupted by distorting the video amplitude and timing. Measurements of certain composite features (described below, see equations 11.2, 11.3 and 11.4) follow. The human scores for a particular video sequence are averaged and forms the dependent variable. The tests involved 64 video sequences of 9 seconds duration each. The following objective

variables are measured:

$$m_1 = RMS_{time}\left\{5.78 \mid \frac{std_{space}(S(O_n)) - std_{space}(S(D_n))}{std_{space}(S(O_n))} \mid\right\} \quad (11.1)$$

$$m_2 = f_{time}\{0.0934 \max\{[RMS_{space}(\Delta O) - RMS_{space}(\Delta D)], 0\}\} \quad (11.2)$$

$$and\ f_{time}(\{x_t\}) = std_{time}(high\ pass\ filter(\{x_t\})) \quad (11.3)$$

$$m_3 = = \max_{time}\left\{4.2522 \log\left(\frac{std_{space}(\Delta D)}{std_{space}(\Delta O)}\right)\right\} \quad (11.4)$$

The symbols used above are as follows:

- $S(\cdot)$ is the Sobel filtering operation. See Appendix A for details of Sobel filtering.
- O_n is the n^{th} frame of the original sequence.
- D_n is the n^{th} frame of the degraded (noise corrupted) sequence.
- *std* is the standard deviation of the pixel values.
- $\Delta F = F_n - F_{n-1}$.
- The high pass filter in equation (11.4) is not described by Voran and Wolf.

The linear model is fitted by least squares and is of the form $\hat{s} = c_0 + \sum_{i=1}^{3} c_i m_i$ where c_i's are the model coefficients and $\hat{s}$ is the estimated quality. The model accounts for 84.2% of the variance of the human scores and the predicted values correlate with the human scores with a correlation coefficient of 0.918. The standard deviation of the errors (mismatch between the predicted and the actual human estimated quality) is 0.53 impairment units.

11.2 Human Visual System (HVS) based modelling

Davies et al. [13] proposed the incorporation of a model of the Human Visual System (HVS) in automatic quality assessment. The Human Visual System (HVS) is modelled from psychophysical and neurophysical studies.

The setup is as follows: a CCD camera is used to examine the contents of 56% of a TV screen. Because of hardware considerations of the camera, the magenta stimuli were used to test the colour saturation.

HVS considerations led to the spatial domain being divided into eighteen bands evenly divided from 15 to -15 Cycles/Degree vertically and 0 to 15 Cycles/Degree horizontally in the Fourier Power Spectrum. The temporal domain is divided into five bands (spanning 0 to 12.5 Hz).

The measured parameters were mean, variance, and range for spatial and temporal bands and the saturation of the magenta image. The parameters were measured after subtraction of the original from the impaired image.

The impairments are caused by a part of a close in frequency co-channel (usually the audio associated with the broadcast), contaminating the video channel, i.e., the spectra of the two channels overlap. The types of impairment are as follows:

A: Unmodulated co-channel, causing panning vertical bars to appear.

B: Unmodulated co-channel, causing fine panning vertical bars.

C: Co-channel carrier with complex narrow-band modulation (e.g., co-channel audio).

D: Co-channel carrier with complex wide-band modulation (e.g., FM audio).

E: Adjacent channel carrier with complex wide-band modulation (e.g., adjacent channel wide-band FM).

F: Unmodulated channel, casing a flickering structured pattern with a "wood-grain" appearance.

G: Pulse modulated image channel carrier (e.g., digital audio), with a visual effect similar to F, but with slowly moving horizontal bars.

The parameters are processed by a three layer neural network, with forty hidden layers, to generate a CCIR impairment grade. The network was trained on 20000 iterations of stimuli varying in CCIR impairment grade from 1 to 5. The impaired images were generated by expert viewers adjusting the impairment grade from 1 to 5.

The error or mismatch between the predicted and actual impairment is quoted as 0.5 on the CCIR impairment grade scale, which implies the correct CCIR impairment grade is produced.

11.3 The relation of quality and complexity

Quality and complexity are shown to be related in this section. The same methods used for predicting video sequence quality can be used for measuring complexity. The two models described both examine temporal and spatial high and low frequency magnitudes. This is visible from the following approaches:

- Voran and Wolf generate variables on Sobel filtered frames and examine differences between consecutive frames (temporal processing).
- Davies et al. partition the spatial and temporal frequency domain and measure mean and variance on these.

Note that Sobel filtering is a band-pass operation.

Measurements of complexity involve both temporal and spatial activity. It is clear then that both quality and complexity estimation involve measurements on spatial and temporal activity and are thus related. Chapter 12 examines the issues in measuring the complexity of a video sequence.

Chapter 12

Measuring complexity

The previous chapter examined the relationship between quality and complexity and showed certain similarities between them. This chapter examines the issues involved in measuring the complexity of a video sequence. Complexity is difficult to quantify; what a human observer would regard as complex a machine/compression algorithm might not, and vice versa. Likely measures for machine orientated complexity include the number of bits encoded, the average bit-rate, and the peak bit-rate. In order to measure the exact complexity of a video sequence it is necessary to examine the entire sequence. This may not be practical for long sequences (e.g., a full length feature film) and an estimate of the complexity may have to be made.

Two basic approaches to the complexity problem are examined:

1. Estimate the average bit-rate (average number of bits encoded) or the likelihood of the bit-rate exceeding a particular threshold from the samples of the bit-rate taken. This is described in this chapter.

2. Model the bit-rate from samples of the input data and estimate the above. This is described in Chapter 13 and the results of the application of this approach in Chapter 14.

This chapter examines the issues involved in statistical sampling of data. An attempt is made to use the Chebychev inequality to derive probabilities of the bit-rate exceeding certain limits. The issues involved in reconstructing the bit-rate function from its samples are examined and finally the chapter is concluded.

12.1 Analysis of samples of the bit-rate

Statistical analysis of the complexity require that the mean and variance of the samples be accurately estimated. The issues involved in generating the samples require examination. A brief overview of random and systematic sampling is first presented then two approaches to analysis of these samples are examined. Two methods of processing the samples are:

1. The Chebychev inequality.

2. Attempting to reconstruct or interpolate between samples.

12.2 An overview of sampling

Sampling attempts to estimate parameters of the population from samples of the population. Two kinds of sampling are examined: Random and Systematic Sampling. One cannot measure the population parameters exactly and directly; one replaces these population parameters by the sample estimates, e.g., population mean is replaced by sample mean.

12.2.1 An overview of random sampling

In random sampling an attempt is made to ensure that each element in a population has an equal chance of being chosen. The mean of the samples taken is a consistent unbiased estimator (see [37]) of the mean of the population, and similarly for the variance of the samples. For large enough sample sizes the population mean and variance can be accurately estimated from the sample mean and variance. Two kinds of sampling are used:

- Replacement sampling (RR): The set of available samples does not change. The possibility of drawing the same sample more than once exists.
- Non-replacement sampling (RNR): A sample can be drawn once only.

The estimates of the mean and variance of these is as follows:

- Replacement sampling:

$$\begin{aligned} E\{\overline{y}\} &= \overline{Y} \\ E\{s^2\} &= \frac{N-1}{N}S^2 \\ Variance\{\overline{y}\} &= \frac{N-1}{N}\frac{S^2}{n} \end{aligned}$$

- Non-replacement sampling:

$$\begin{aligned} E\{\overline{y}\} &= \overline{Y} \\ E\{s^2\} &= S^2 \\ Variance\{\overline{y}\} &= \frac{N-n}{N}\frac{S^2}{n}, \end{aligned}$$

where

- $E\{\cdot\}$ denotes the mathematical expectation.

- $\overline{y}$ is the sample mean.
- s^2 is the sample variance.
- $\overline{Y}$ is the population mean.
- S^2 is the population variance.
- N is the population size.
- n is the number of samples taken.

12.2.2 An overview of simple systematic sampling

In Systematic Sampling, elements a particular distance d apart are selected with the first element's position $0 \cdots d-1$ determined randomly. Each element in the population thus has an equal chance of being selected. Certain difficulties with systematic sampling exist:

- Since the samples are constantly spaced apart the method is vulnerable to periodic trends in the data.
- Simple systematic sampling (i.e., only using one start position) cannot provide information on the variance between different clusters. A cluster can be defined as the samples associated with a certain start position.

The mean and variance of the population can be estimated as follows:

$$E\{\overline{y}\} = \overline{Y}$$

The simple systematic sampling variance of the samples is impossible to calculate using only one pass [49], but in practice the variance estimators for simple random sampling without replacement may be used [49]. Note that the cluster mean and internal variance (as for RNR sampling) are unbiased estimators of the population parameters [49]. Note that it is possible to estimate the sample (and hence population) variance but it requires two passes of sampling, i.e., two different start positions.

12.3 The Chebychev inequality

The mean and variance obtained from the samples can be used to obtain probabilities that the bit-rate exceeds certain values. This gives an idea of the upper and lower bounds on the bit-rate. The Chebychev inequality is independent of the underlying distribution of the data [51]: $P(\mid x - m_x \mid > k\sigma_x) < \frac{1}{k^2}$, the probability of x being within k standard deviations

from the mean is less than $\frac{1}{k^2}$. If the distribution of the data is known then a more accurate estimate of the probability of the data exceeding a certain threshold is possible. Use of the Chebychev inequality requires that the population variance and mean be estimated as accurately as possible.

12.4 Estimation from samples of the bit-rate

Given a particular sampling method one can estimate the mean and the variance of the population. In particular one can generate confidence intervals for the mean from the t statistic. The limitation of this method is that the average bit-rate and the average number of bits needed to encode the sequence are all that can be estimated without knowledge of the distribution of the data.

The case of simple systematic sampling of the data is similar to the situation of sampling a function at some rate. Nyquist predicts that if the sampling rate exceeds twice the highest frequency within the function, the function can be perfectly reconstructed from the samples [51]. One cannot guarantee that the bit-rate function is sampled at a high enough rate and aliasing is bound to occur.

This section looks at the issues of correcting for aliasing and examines the effect of the sampling functions on aliasing. Some sampling functions are proposed along with grabbing several consecutive bit-rate samples at a time.

12.4.1 A proposal to reconstruct bit-rates from the bit-rate samples

This approach attempts to reconstruct the bit-rate function from its samples. Given that one cannot estimate the bandwidth of this function it is necessary to expect aliasing and attempt to correct for it. This section first examines sampling in the discrete domain and then describes the effect of aliasing at each frequency after taking the Fourier Transform (FT).

Sampling results in the following:

$$\begin{aligned}
f_s(t) &= f(t)h_\Pi(t) \\
\Rightarrow F_s(w) &= F(w) * H(w) \\
\text{now } h_\Pi(t) &= h(t) * \sum_{n-\infty}^{\infty} \delta(t - nT) \\
\text{now } H(w) &= G(w) * \sum_{n-\infty}^{\infty} \delta(w - n\omega_0) \cdots \omega_0 = \frac{2\pi}{T} \\
\Rightarrow H(w) &= \frac{1}{T_0} \sum_n G(n\omega_0)\delta(w - n\omega_0)
\end{aligned}$$

$$\Rightarrow F_s(w) = \frac{1}{T_0}[F(w) * \sum_n G(n\omega_0)\delta(w - n\omega_0)]$$
$$F_s(w) = \frac{1}{T_0}\sum_n G(n\omega_0)F(w - n\omega_0)$$

and,

$$\sum_{n-\infty}^{\infty} \delta(t - nT) = \sum_{n-\infty}^{\infty} F_n e^{\jmath n\omega_0 t}$$
$$\text{and } F_n = \frac{1}{T_0}\int_{\frac{-T_0}{2}}^{\frac{T_0}{2}} \delta(t)e^{-\jmath n\omega_0 t}dt$$
$$= \frac{1}{T_0}$$
$$\text{and } FT\{\sum_{n-\infty}^{\infty} \frac{1}{T_0}e^{\jmath n\omega_0 t}\} = \frac{1}{T_0}\sum_n \delta(w - n\omega_0)$$

Given a discrete function one can define the following:

$$\hat{f}(t) = f(t) \cdots 0 \leq t \leq MT$$
$$= 0 \quad otherwise$$
$$\text{now } \hat{F}(w) = \int_0^{MT} f(t)e^{-\jmath wt}dt$$
$$\approx \sum_{k=0}^{MT} f(kt)e^{-\jmath nk\omega_0 T}T$$

$$\text{note } dt = T \quad t = kT \quad w = n\omega_0$$
$$\hat{F}(w)\mid_{w=n\omega_0} \approx TF_D(n\omega_0)$$
$$= \sum_{k=0}^{M-1} f(kt)e^{-\jmath nk\omega_0 T}$$
$$\text{Usually written as } F_D(n) = \sum_k f(k)e^{\frac{-\jmath 2\pi nk}{M}} \cdots \omega_0 = \frac{2\pi}{TM}$$

Without loss of generality $T = 1$ so, $F_D(m\Omega) \approx \sum_n G(n\omega_0)F(m\Omega - n\omega_0)$

The effect of aliasing on any particular point in the frequency domain can be described as follows, provided the magnitude of the sampling function decreases with frequency: consider the effect of aliasing on the DC component. For each alias term at some multiple of ω_0 the aliasing can be determined; the positive multiples of ω_0 contribute negative frequency terms and the negative multiples of ω_0 contribute positive frequency terms. In each case the spacing of the alias spectra is ω_0 and the scale factor of the alias term is a function of its position in the transform of the sampling function. All other positions are zero except for the DC position

which is one. One can see that the next positive frequency aliasing is found by shifting the DC aliasing one position to the right and for the next negative frequency one term to the left. Note that terms shifted over the allowed range of frequencies disappear and zeros are introduced at the appropriate edges.

A matrix can be formed with each row the alias terms and the element on the main diagonal set to one (the element on the main diagonal is the actual unaliased value). The elements on the main diagonal run from the upper left corner of the matrix to the lower right corner for square matrices.

Note that this is a diagonal matrix, i.e., all non-zero terms lie on a left to right diagonal, with the off diagonal terms less than the terms on the main diagonal. The matrix thus formed can be Gauss reduced (solved by some numerical method) to yield the unaliased spectrum. It may be better to retain more frequencies initially and later reduce the number of frequencies (used to increase the accuracy of the upper frequencies especially).

12.4.2 The sampling function

The shape and width of the sampling function affect aliasing for subsampled data. Two issues affect the reconstruction between samples:

The Sampling rate: Since no prior knowledge of the bandwidth of the bit-rate exists one cannot apply Nyquist's sampling theorem. This implies that to reduce aliasing one needs to make the sampling rate as high as practicable.

The width of the sampling function: The wider the sampling function the narrower the function in the frequency domain, i.e., the narrower the spacing between the $-3db$ points.

Suggested sampling functions:

$$\begin{aligned} rect(t/\tau) &= 1 \quad -\frac{\tau}{2} \leq t \leq \frac{\tau}{2} \\ &= 0 \text{ otherwise} \\ \text{Fourier Transform}\{rect(t/\tau)\} &= \tau Sa(\frac{w\tau}{2}) \end{aligned}$$

$$\begin{aligned} \Lambda(t/\tau) &= t \quad \cdots \quad -\tau \leq t \leq 0 \\ &= 1-t \quad \cdots \quad 0 \leq t \leq \tau \\ &= 0 \text{ otherwise} \\ \text{Fourier Transform}\{\Lambda(t/\tau)\} &= \tau Sa(\frac{w\tau^2}{2}), \end{aligned}$$

where,

- τ is the width of the *rect* and Λ functions.

- The rect function is a rectangular pulse (top hat) function.
- Λ is the integral of the *rect* function.
- $Sa(\alpha) = \frac{\sin(\alpha)}{\alpha}$

Note that the Fourier Transforms are real and the Fourier Transform of $\Lambda(t/\tau)$ dies away faster than the $rect(t/\tau)$ function. A sampling function with a sharper roll-off, or narrower Fourier Transform, can be constructed by integrating the given sampling functions. Note that the resulting function has a higher magnitude than before and possibly has discontinuities.

It is possible to create sampling functions with smaller magnitudes past the $-3db$ frequencies, i.e., sharper roll offs. This can be done by cascading filters with the *rect* or Λ frequency responses which corresponds to the following: $f_n(t) = f_{n-1}(t) \star f_{n-1}(t)$ or each new filter is the convolution of the previous filter time response with itself. The frequency response is the Fourier Transform (FT) of $f_n(t)$.

Alternatively the following is possible:

$$\begin{aligned} f_n(t) &= \int (f_{n-1}(t-t_0) - f_{n-1}(t+t_0))dt \\ f_0(t) &= rect(t/\tau) \\ \text{or } f_1(t) &= \Lambda(t) \end{aligned}$$

$$\begin{aligned} \text{Note } FT\{\int h(t)dt\} &= \frac{1}{jw}H(w) + \pi H(0)\delta(w) \\ \text{And } FT\{h(t-t_0)\} &= H(\omega)e^{j\omega t_0} \end{aligned}$$

$$\begin{aligned} \Rightarrow FT\{f(t)\} &= \frac{1}{jw}[FT\{g(t)\}e^{jwt_0} + jwF(0)\delta(w)e^{jwt_0} - \\ &\quad (FT\{g(t)\}e^{-jwt_0} + jwF(0)\delta(w)e^{-jwt_0})] \end{aligned}$$

$$\text{and } jwF(0)\delta(w)e^{jwt_0} = jwF(0)\delta(w)e^{-jwt_0}$$

$$\begin{aligned} FT\{f(t)\} &= \frac{1}{jw}[FT\{g(t)\}(e^{jwt_0} - e^{-jwt_0})] \\ &\quad \cdots sin\alpha = \frac{1}{2j}(e^{j\alpha} - e^{-j\alpha}) \\ &= FT\{g(t)\}2t_0Sa(wt_0) \quad \cdots \quad t_0 = \frac{\tau}{2} \\ &= FT\{g(t)\}\tau Sa(\frac{w\tau}{2}) \end{aligned}$$

Note that t_0 could be any value desired, although maintaining the same value through each stage results in the same frequency response as cascading a series of filters with the

$FT\{rect(t/\tau)\}$ frequency response.

12.4.3 An alternative approach - Modification of the Sampling Function

The previous subsection examined the effect of sampling functions on aliasing. Here the problem is considered in the frequency domain and a modification proposed.

In the frequency domain after sub-sampling the alias terms occur spaced by ω_0. One approach to removing alias terms is to multiply the Fourier Transform of the sampling function by a function with appropriately spaced zero crossings. One needs a function that zeros the alias terms but not the signal terms. This requires that the function's DC component must be non-zero.

- The rect function and functions based on the rect function are unsuitable since the width of the function is the spacing of the samples: $rect(t/\tau) = Sa(\omega\tau/2)$. This implies for zero crossings at multiples of ω_0, $n\omega_0\tau/2 = n\pi$ or $\tau = 2\pi/\omega_0 = T$, where T is the sampling period.

- An alternative approach is the cosine function. The desired DC component is preserved. The disadvantage is that it only removes half of the aliasing at any one time. This requires the use of more than one cosine to achieve good reduction of aliasing. In the time domain one convolves with spaced Dirac deltas which really correspond to adding two shifted versions of the sampling function. For each sampling rate one needs to recalculate the composite sampling function. For the zero-crossings:

$$\begin{aligned} \alpha n\omega_0 &= \pi/2 \\ \alpha &= \pi/2n\omega_0 \\ and\ \omega_0 &= 2\pi/T \\ \alpha &= T/4n \end{aligned}$$

  Note that each cosine only removes half of the remaining alias terms. If one starts with the cosine to remove the first alias term (this one has the most energy of all the alias terms) then its cosine frequency is $T/4$, the next alias term exists at $2\omega_0$ and its cosine frequency is $T/8$, the next at $4\omega_0$ and its cosine frequency is $T/16$, the next at $8\omega_0$ and its cosine frequency is $T/32$, and so on. Note the frequency position of the alias term is twice that of its predecessor and the cosine frequency doubles as well.

  In the time domain one convolves the output of the sampling function with the time domain representation of each frequency domain cosine (the cosine is $\frac{\delta(t-\alpha t)+\delta(t+\alpha t)}{2}$ in the time domain). One can regard each frequency domain multiplication by a cosine as an artificial doubling of the sampling rate.

With this approach one may apply the previous method with less aliasing and hopefully faster convergence *or* use enough cosines to unalias the original spectrum and proceed as with a signal sampled at the Nyquist rate or better.

12.4.4 Difficulties with the sampling approaches

The problems of reconstructing between samples and prediction of the bit-rate from statistical considerations are examined here.

- Reconstructing a signal from a sub-sampled version is a time consuming operation. Each Fourier component of the aliased signal at a particular frequency is a linear combination of Fourier components of other frequencies from the unaliased signal. The systems of equations may not be solvable.
- Zeroing alias terms requires a sampling function tailored to the sampling rate and performance required.
- The Chebychev inequality yields an approximation to the real values.
- Estimating the average bit-rate does not provide any information about the peaks which need to be considered in fixed bit-rate coders.

Note that random sampling is more rigorous than simple systematic sampling in that the population estimates are easier to extract and the population variance can be examined.

The above difficulties imply that fitting a model to the data would result in better performance provided a good model fit is achieved. The process of model fitting is examined in Chapter 13.

Chapter 13

Regression

Given data of several independent variables and one dependent variable one can attempt to fit a function to the data. A model can be built up as a truncated Taylor Expansion, e.g., $f(x_1, x_2, \cdots, x_n) = \sum_{i=0}^{N} c_i x_i$.

A number of issues affecting model construction using regression are examined in this chapter. Model fitting using Least Squares; the reduction of dimensionality (eliminating correlated variables) using Principal Component Analysis well as direct elimination of correlated variables; evaluating the fit of a model using tradition statistical techniques; cross-validation and the Jackknife are examined.

13.1 Fitting a Model using Least Squares

The method of Least Squares attempts to minimise the mean of the deviations of the predicted (modelled) results from the actual data. If one has some model with coefficients, $c_0, c_1, \cdots, c_n$, then one can attempt to solve for the coefficients as follows:

$$
\begin{aligned}
E\{(\mathbf{y} - \sum_{i=0}^{N} c_i \mathbf{x_i})^2\} &= 0 \\
\frac{\partial E\{(\mathbf{y} - \sum_{i=0}^{N} c_i \mathbf{x_i})^2\}}{\partial c_j} &= E\{2(\mathbf{y} - \sum_{i=0}^{N} c_i \mathbf{x_i})\mathbf{x_j}\} \\
\text{now } E\{\mathbf{x_i}\} &= \frac{1}{M} \sum_{k=0}^{M} x_{ik} \\
\frac{\partial E\{(\mathbf{y} - \sum_{i=0}^{N} c_i \mathbf{x_i})^2\}}{\partial c_j} &= \frac{2}{M} \sum_{k=0}^{M} y_k x_{ik} - \frac{2}{M} \sum_{i=0}^{N} \sum_{k=0}^{M} c_i x_{ik} x_{jk} \\
&= 0 \\
\sum_{k=0}^{M} y_k x_{ik} &= \sum_{i=0}^{N} \sum_{k=0}^{M} c_i x_{ik} x_{jk}
\end{aligned}
$$

If one creates a matrix A with the column vectors the terms $\mathbf{x_i}$ then $\sum_{i=0}^{N}\sum_{k=0}^{M} x_{ik}x_{jk}$ amounts to a matrix multiplication A^TA, i.e.:

$$\begin{aligned} \mathbf{y} &= A\mathbf{c} \\ A^T\mathbf{y} &= A^TA\mathbf{c}, \end{aligned}$$

where $E\{\cdot\}$ is the expected value operator, and $\mathbf{y}$ is the data to be modelled. Note that A^TA is a square matrix and equation 13.1 can be solved by Gauss reduction or matrix inversion. Note that for c_0, x_0 is a vector of 1's.

13.2 Reduction of Dimensionality

The reduction of the number of variables in the data leads to a corresponding simplification of the number of terms in the model. Given there are a large number of terms if one has a second or higher order model, one can attempt to reduce the number of cross-product terms (e.g., x_ix_j) by reducing the number of variables, since $E\{x_ix_j\} = 0$ for orthogonal variables.

- Variables which correlate *must* be removed otherwise the model coefficients cannot be generated. The rank of A^TA is less than the number of variables, which implies Gauss Reduction fails.
- Variables which are a linear combination of other variables must be removed.

Three methods exist to find a correlation between variables:

1. Examine the scatterplots of each variable against the others and by eye see if there is a linear or other relationship.
2. Examine the correlation coefficient. The correlation coefficient is especially vulnerable to outliers. A high absolute value indicates a linear relationship between the data.

   The correlation coefficient (ρ) is found as follows:

$$\begin{aligned} \rho &= \frac{covariance(\mathbf{x_i}, \mathbf{x_j})}{variance(\mathbf{x_i}).variance(\mathbf{x_j})} \\ \text{and} \quad covariance(\mathbf{x}, \mathbf{y}) &= \sum_{i=0}^{N}(\mathbf{x_i} - m_x)(\mathbf{y_i} - m_y) \\ \text{and} \quad variance(\mathbf{x}) &= covariance(\mathbf{x}, \mathbf{x}) \end{aligned}$$

   The limits on the correlation coefficient are $-1 \leq \rho \leq 1$:

- Case $\mathbf{x_i} = \mathbf{x_j}$ then it can be shown $\rho = 1$.
- Case $\mathbf{x_i} = -\mathbf{x_j}$ then it can be shown $\rho = -1$.

3. Examine the correlation coefficient of the suspect variables, and if the value is low examine the scatterplot to see if there are large outliers. If there is the suspicion that the outlier(s) contribute significantly to the low correlation coefficient, remove the outlier(s) and recalculate the correlation coefficient.

Outliers in general indicate the following:

1. An error in the data, either in the collection or entering of the data.
2. Some relationship between the variables unexplained by the model.
3. Too few samples were taken and the model is inadequate.

Outliers can be handled in two basic ways:

1. Ignore them and continue as if one is certain of the correctness of the data.
2. Remove the outliers and continue.

A more serious problem is the correlation of a variable with a linear combination of the other variables. The only way to find such a variable is to attempt to fit a linear combination of variables to a suspect variable and if the fit is good (a good correlation coefficient between the suspect variable and the linear combination) then the suspect variable should be removed.

Section 13.6 examines Principal Components Analysis which can be effectively used to reduce dimensionality and generate orthogonal variables (orthogonal variables imply no cross-product terms, e.g., $x_i x_j$).

13.3 Evaluating the fit of a Model

One can evaluate the fit of a model by correlating the values predicted by the model and the original values. A good model will have a high positive correlation constant (a negative value indicates an error). A scatterplot of the predicted vs. the actual values should show a linear relationship. Outliers indicate areas where the model does not predict well and may indicate either too few samples taken (and perhaps an inappropriate model) or an error in entering the data. Any trends in the time or sequence plot of the residuals (difference of the model predicted values and the actual values) indicates the model is inadequate. The residuals should be randomly distributed with zero mean.

Confidence intervals on the model coefficient provide some information on the robustness of the model.

13.3.1 Establishing confidence intervals for the model coefficients

Confidence intervals for the model coefficients indicate the immunity of the model to noise in the data. The procedure is as follows:

1. Prove that the estimates of the coefficients are unbiased.
2. Determine the variances of the coefficients.
3. Apply Student's t (described later in this section) test and determine the confidence intervals on the coefficient. This also tests whether the coefficient is zero or not. A potentially zero (or small) coefficient could be removed.

This subsection demonstrates that the model coefficients are unbiased and a method for calculating the variance per coefficient. Calculation of the actual confidence intervals using Students t test and the Analysis of Variance (ANOVA) F statistic is demonstrated.

The model coefficients are unbiased

This implies the expected value of the estimate of the coefficient is the actual coefficient value. The expected values are conditional on the observed values of the measurements of x and y. The estimated model coefficients are g:

$$\begin{aligned} E\{A^T\mathbf{y}\} &= E\{A^TA\mathbf{c}\} \\ E\{A^T\mathbf{y}\} &= E\{A^TA\mathbf{g}\} \\ 0 &= A^TAE\{\mathbf{c}-\mathbf{g}\} \\ \Rightarrow E\{\mathbf{c}\} &= E\{\mathbf{g}\} \\ \mathbf{c} &= E\{g\} \end{aligned}$$

Estimating the variances of the model coefficients

This can be done by solving the following linear equation:

$$\begin{aligned} A^T\mathbf{y} &= A^TA\mathbf{c} \\ A^T\mathbf{y} &= A^TA\mathbf{g} \\ Var\{0\} &= Var\{A^TA\mathbf{c}-A^TA\mathbf{g}\} \\ 0 &= Var\{\mathbf{y}-\epsilon-A^TA\mathbf{g}\} \\ 0 &= Var\{\mathbf{y}\}-Var\{\epsilon\}-Var\{A^TA\mathbf{g}\} \\ 0 &= Var\{\mathbf{y}\}-Var\{\epsilon\}-A^TAVar\{\mathbf{g}\} \\ A^TAVar\{\mathbf{g}\} &= Var\{\mathbf{y}\}-Var\{\epsilon\} \end{aligned}$$

$$A^T A \begin{pmatrix} Var\{g_1\} \\ Var\{g_2\} \\ \vdots \\ Var\{g_n\} \end{pmatrix} = \begin{pmatrix} Var\{y_1\} - Var\{\epsilon_1\} \\ Var\{y_2\} - Var\{\epsilon_2\} \\ \vdots \\ Var\{y_n\} - Var\{\epsilon_n\} \end{pmatrix}$$

$$A^T A \begin{pmatrix} Var\{g_1\} \\ Var\{g_2\} \\ \vdots \\ Var\{g_n\} \end{pmatrix} = \begin{pmatrix} Var\{y\} - Var\{\epsilon\} \\ Var\{y\} - Var\{\epsilon\} \\ \vdots \\ Var\{y\} - Var\{\epsilon\} \end{pmatrix}$$

Where $Var\{\epsilon\}$ is the variance of the difference between the model and the actual y values, and $Var\{y\}$ is the variance of the dependent variable.

Student's t test

The t distribution (probability density function) is an approximation to the normal distribution [37].

Given normally distributed data one can estimate the confidence intervals as follows:

1. Adjust the distribution of the means of the data to zero mean and unit variance: $Z = \frac{\overline{X}-\mu}{\sigma/\sqrt{n}}$.

2. For a particular confidence $(100(1-\alpha)\ \%)$ one can find a range of values with that confidence of finding the real value within the range.

   - The probability of the value being outside the range is α or: $P(-z_{\alpha/2} \leq z \leq z_{\alpha/2}) = 1 - \alpha$.
   - Substituting z and simplifying leads to: $P(\overline{X} - z_{\alpha/2}\frac{\sigma}{\sqrt{n}} \leq \mu \leq \overline{X} + z_{\alpha/2}\frac{\sigma}{\sqrt{n}}) = 1 - \alpha$.
   - This implies the following confidence interval: $(\overline{x} - z_{\alpha/2}\frac{\sigma}{\sqrt{n}}, \overline{x} + z_{\alpha/2}\frac{\sigma}{\sqrt{n}})$.

3. t_α can be estimated by $t_\alpha = \frac{\overline{x}(n-1)}{s}$, where s is the estimate of the variance.

4. The Degrees of Freedom of the t-statistic is the number of independent observations less the number of restrictions.

5. Values of t_α (or z_α) can be found by looking up the values in an appropriate table, e.g., table of Critical Values of t [37]. It is possible to calculate the t distribution for specified degrees of freedom and hence the values of $t_{\alpha/2}$ for which the area (probability) between $+t_{\alpha/2}$ and $-t_{\alpha/2}$ is the desired probability $1 - \alpha$.

6. It should be noted that this is valid for normally distributed means (usually valid, see the Central Limit Theorem [37]) and a large enough sample set.

Now in general σ is not known and an unbiased estimator is: $s = \frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{n-1}$ and the mean of the samples is an unbiased estimator of the population mean. The distribution of $T = \frac{\overline{X}-\mu}{S\sqrt{n}}$ is a t distribution with $n-1$ degrees of freedom. Note that the t distribution approaches the normal distributions for large sample sizes [37]. The t distribution should be used instead of the normal distribution in calculating confidence intervals since it attempts to compensate for non-normal distributions.

Application of Student's t test to estimating the significance of Model Coefficients

The test statistic is $t(v) = \frac{\hat{c}_i - (\textit{estimated model coefficient})}{\textit{variance of the model coefficient}}$, where v is the degrees of freedom (here $n-k$) with n is the number of observations and k is the number of coefficients. Since the test examines whether or not a particular coefficient is zero, the estimated model coefficient is zero. The t statistic is calculated with $(\textit{estimated model coefficient}) = 0$ and the probability of obtaining $t \leq \frac{\hat{c}_i}{\textit{variance of the model coefficient}}$ is determined. If the probability is greater than expected then the estimated model coefficient cannot be zero. The expected probability is determined from the confidence interval $(100(1-\alpha)\ \%)$ selected beforehand and is hence α.

If *estimated* $t >|$ *calculated or tabulated* $t\ |$ then with $100 * \alpha$ significance the coefficient is not significant and can be ignored, i.e., the coefficient cannot be within the $100(1-\alpha)$ confidence interval.

Application of the Analysis of Variance (ANOVA) F statistic to test the significance of Model coefficients

This is essentially hypotheses testing, testing the hypotheses that a particular coefficient is zero or testing whether all the coefficients are zero. The ANOVA F statistic is:

$$
\begin{aligned}
F &= \frac{MSM}{MSE} \\
MSM &= \frac{\sum_i (\hat{y}_i - \overline{y})^2}{DFM} \\
MSE &= \frac{\sum_i (y_i - \hat{y}_i)^2}{DFE} \\
\text{(Degrees of freedom Model)}\ DFM &= 1 \\
\text{(Degrees of freedom Error)}\ DFE &= N-1 \\
\text{(Degrees of freedom Total)}\ DFT &= DFM + DFE
\end{aligned}
$$

The F statistic tests the hypothesis $c_1 = c_2 = \cdots = c_n = 0$. If one of the Model coefficients is non-zero then one would expect $MSM > MSE$. One proceeds as follows:

- Calculate F. The degrees of freedom of the numerator and denominator are 1 and $N-2$ respectively, where N is the number of samples.
- The area under the tail of the distribution of $F(1, N-2)$ above the F value represents the significance level.

Note that the t statistic is related to the F statistic as follows: $F(1,n) = t(n)^2$ [39]. Since the F statistic is calculated using the MSE and MSM it is simpler to calculate than the t statistic and is thus recommended for confidence interval calculations.

13.4 Simple Cross-validation

Cross-validation measures the prediction error of the model under consideration. Simple Cross-validation divides the available data into two sets. One set is used to generate the model and the second set (the test set) used to evaluate the fit of the model [40], [16]. Usually the test and generation sets are equal in size. The performance of the model on the test set is likely to be similar to the performance of the model for arbitrary valued variables usually encountered in practice.

Double cross-validation involves using data different from the data used to generate the coefficients of the model, and different from the data used to estimate the type (form) of the model (e.g., logarithmic, truncated Taylor series, etc).

The regression models generated are tested using double cross-validation.

13.5 Over-fitting

Within all data sets each measurement (data point) is perturbed by noise. Over-fitting involves fitting a model which attempts to predict the noise component of the data [16]. Clearly such a model is not likely to perform well on a test set since it is impossible to derive a deterministic model for probabilistic data (i.e., noise).

13.6 Theory of Principal Components Analysis (PCA)

This section presents the theory of PCA [21], [22]. The process of PCA is essentially a projection of each set of observations (of different variables) onto a set of orthogonal vectors. This process decorrelates each variable.

Given a set of m variables with n observations on each variable arranged in matrix X, it is usual to calculate the Principal Components from the covariance matrix though other methods exist.

The process is as follows:

1. Create matrix $Z = X - m_x$, where each row of m_x has as elements the means of each of the column vectors of X (i.e., from each element of a column vector subtract the mean of the column vector).

2. Create the covariance matrix (dimensions nxn) of A, where $c_{ij} = E\{(x_i - \overline{x_i})(x_j - \overline{x_j})\}$. The covariance matrix (C) is then $Z^T Z$.

3. Extract the Eigenvalues and Eigenvectors from the covariance matrix. Note that the Eigenvectors of a square symmetric matrix are orthogonal [28].

4. Sort the Eigenvectors in order of decreasing Eigenvalues and arrange the Eigenvectors as the column vectors of matrix A.

5. Transform X by $Y = (X - m_x)A$ (the inverse transform is $X = YA^T + m_x$). Note that the matrix of Eigenvectors is orthonormal [28]: $A^{-1} = A^T$. The columns of Y are the Principal Components. The variance of the Principal Components are:

$$\begin{aligned} y_i &= (X - m_x)e_i \\ &= Ze_i \text{ variable substitution} \\ Var\{y_i\} &= E\{(Ze_i)^2\} \\ &= E\{(e_i^T Z^T)(Ze_i)\} \\ &= Var\{e_i^T (Z^T Z)e_i \\ &= e_i^T C e_i \\ &= e_i^T \lambda_i e_i \\ &= \lambda_i \end{aligned}$$

   where λ_i are the eigenvalues of $Z^T Z$ and e_i are the eigenvectors of $Z^T Z$. Thus total variance $= \prod_{i=1}^{n} \lambda_i$.

6. Selecting the k Eigenvectors associated with the largest k Eigenvectors accounts for $\frac{\prod_{i=1}^{k} \lambda_i}{\prod_{i=1}^{n} \lambda_i}$ of the total variance. This allows one to use a subset of the total Eigenvectors available provided they explain enough of the variance.

13.7 The Jackknife

When modelling data is is usual to apply cross-validation. Cross-validation involves dividing the data into a training set (which is used to create the model) and a validation set (used to test the fit of the model). When too little data exists then one does not have sufficient data either to test or generate the model. The Jackknife allows one to use most of the data for

generating the model and then estimate the mean and confidence intervals for each model coefficient.

The Jackknife [17], [5] generates a series of models by in turn removing one (or more) data points from the data set to form the model generation set. A series of models is generated and hence samples of the mean and variance of each coefficient and the error generated. Clearly one can then generate confidence intervals for the model and error.

If one removes more than one data point per model then the number of models to be generated are:

$$\begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{(n-k)!k!}$$

where k is the number of data points removed from the data set per model generated.

The simplest model is the "leave out one" [5] model where only one datum per variable is removed at a time (or one row if the data is arranged as column vectors in matrix form). Note that applying PCA allows one to calculate the variances directly without reference to any other variables.

Strictly speaking the above is usually described as a form of cross-validation [17]. The Jackknife is usually applied to parameter estimation and estimating the standard deviation of the parameter as described above (e.g., estimating the error of a median).

The estimate of the variance is defined as follows (with x representing the coefficient or estimate to be Jackknifed):

$$\frac{1}{n(n-1)} \sum_{1=1}^{n} (x_i - \overline{x})$$

Removing one datum at i leads to:

$$\overline{x_{(i)}} = \frac{n\overline{x} - x_i}{n-1} = \frac{1}{n-1} \sum_{i \neq j} x_j$$

where $\overline{x_{(i)}}$ is the Jackknifed estimate of the mean.

Now the average of the removed data is:

$$\overline{x_{(\cdot)}} = \sum_i \frac{x_{(i)}}{n} = \overline{x}$$

and

$$\sigma_{Jackknife} = \frac{n-1}{n} \sum_{i=1}^{n} (\overline{x_{(i)}} - \overline{x_{(\cdot)}})$$

[17].

Thus the the coefficient value is taken as $\overline{x}$ and the variance of the coefficient value is $\sigma_{Jackknife}$.

13.8 Summary

This chapter covers some of the issues involved in fitting a model. The issues relating to generating confidence intervals were examined. It was shown that the model coefficients are unbiased. Methods for generating the model coefficient confidence intervals using the ANOVA F statistic and the t-statistic were examined. The process of model validation using cross validation and the Jackknife is examined. The use of the Principal Components Analysis for decorrelating variables and hence simplifying the modelling process, is discussed.

The results of the model fitting process is summarised in Chapter 14.

Chapter 14

Fitting measured variables to the subjective quality variables

Some variables were measured on the sampled data measured on the frames themselves and on the even and odd fields: mean and variance of the motion vectors, mean and variance of the difference image and the absolute values of the difference image.

Twenty-five frames were extracted at each of the second, fifth, eighth, eleventh and fourteenth seconds (this is systematic sampling) of the following video sequences: Animation1, Animation2, Fast Film, Cycling, Boxing, Cross Country, Soccer, Noisy News 1, Noisy News 2, Noisy News 3, Talking Heads, A Few Good Men, Egoli, Scrolling, Crawls, Flower Garden and Disc3. The frames were grabbed using an EPIX4M framegrabber card on an IBM compatible PC (Personal Computer). Due to hardware limitations on the card a greyscale block of 512 by 512 pixels in the centre of the frame were grabbed.

Two bit-rates were generated for each sequence at the following subjective quality assessment levels: excellent and acceptable (or good).

The methods of generating the raw data are examined and the data itself shown. MATLAB was used to plot and generate the results with confidence intervals generated for 5% error.

Firstly the objective variables measured were examined to remove redundant variables, next linear and then quadratic functions were fitted to the data and the prediction accuracy of these models examined. Finally the confidence intervals of the model coefficients are generated.

14.1 Generating the raw data

This section describes the parameters measured on the test sequences. The bit-rates generated were for colour sequences but the objective variables were extracted from greyscale versions of the same images because of hardware limitations. This is not as great a limitation as it

seems since the Human Visual System is more sensitive to greyscale spatial frequency content than colour spatial frequency content [26]. The bit-rates were generated by a fixed bit-rate MPEG-2 encoder. The mean and variance of the original frames and fields were extracted. The mean and variance of the motion vectors and the difference between the best-fit block and the original search block are found. The motion vectors are calculated on both field and frames using the Three Step Search (see Section 4.17) on eight by eight search blocks. The number of non-zero motion vectors are counted. The frames and fields are edge detected using the Sobel edge detector and the mean and variance of the edge detected frames and fields extracted. The features were chosen for the following reasons:

- Both fields and frames were examined to see if measurements on one were better than on the other.
- The variance of the frame/field are an indicator of the frequency content of the frame/field.
- The mean of the frame/field contains more energy than any of the other frequency components.
- The mean of the length of the motion vectors is an indicator of the amount of motion inside the frame/field.
- The variance of the motion vectors is an indicator of the degree and uniformity of motion within the frame/field.
- The number of non-zero motion vectors is another indicator of the amount of motion within the frame/field.
- The mean and variance of the edge detected frames/fields estimate the high frequency content of the images.

The data is summarised in Tables 14.1, 14.2 and 14.3.

14.2 Removing redundant variables

The correlation coefficient of the variables against each other are examined and any variables with a coefficient greater than 0.8 examined and removed. Tables 14.4 and 14.5 list the regression coefficients. The following variables are retained:

- Motion Vector Frame Mean.
- Motion Vector Frame Variance.
- Number of active (non-zero) Motion Vectors.

Name of sequence	Motion Vectors on Frame				Motion Vectors on Field			
	Mean	Variance	RMS	MtnVec	Mean	Variance	RMS	MtnVec
animate1	37.056	2745.3	64.175	490690	30.032	1608.0	50.110	242006
animate2	39.064	2465.6	63.179	490557	33.835	1486.5	51.297	242080
fastfilm	34.054	1805.4	54.452	491299	29.641	1264.2	46.291	242059
cycling	43.792	2647.0	67.563	491533	38.045	1715.9	56.244	242105
boxing	39.695	2036.0	60.097	491562	35.082	1280.9	50.117	242114
cross country	49.244	3648.6	77.933	490640	40.835	2042.6	60.911	242066
soccer	31.533	1841.2	53.250	490688	28.327	1384.1	46.760	242015
news	43.050	2296.3	64.417	491482	37.082	1489.9	53.526	242097
talking heads	27.282	1424.6	45.571	488624	23.977	888.11	38.249	241899
fgmen	41.364	2824.6	67.347	490437	35.503	1446.3	52.027	242003
scrolling	37.579	2253.6	60.543	449299	30.627	1176.2	45.981	231562
egoli	24.963	1297.0	43.820	489681	22.903	832.07	36.832	241909
crawls	26.504	1851.6	50.538	490511	23.736	1241.0	42.478	241965
flower garden	32.38	1867.4	53.999	488838	29.590	1557	49.321	241790
disc3	43.914	2699.9	68.032	491412	42.429	1959.6	61.317	242112

Table 14.1: First part of the raw data for the bit-rate model. MtnVec denotes the number of non-zero motion vectors.

Name of sequence	Difference Frame			Sobel filtered Frame		
	Mean	Variance	RMS	Mean	Variance	RMS
animate1	17.240	718.33	31.868	39.722	2501.3	63.868
animate2	12.204	488.97	25.257	51.272	3889.7	80.737
fastfilm	15.045	466.80	26.328	38.619	1880.8	58.071
cycling	32.875	1816.9	53.830	99.919	7324.8	131.56
boxing	17.589	523.1	28.852	45.960	2569.6	68.425
cross country	22.476	920.36	37.756	45.867	3381.3	74.062
soccer	14.592	581.38	28.183	63.669	4075.0	90.160
news	18.783	457.16	28.460	57.659	1842.7	71.884
talking heads	7.1927	180.74	15.247	40.762	1914.8	59.803
fgmen	14.347	437.32	26.060	34.647	2526.9	61.052
scrolling	24.522	2205.9	52.984	75.755	8957	121.22
egoli	7.2215	142.10	13.937	37.716	142.10	13.937
crawls	14.603	607.27	28.645	67.299	4481.0	94.922
flower garden	92.565	786.65	33.030	92.565	9481.7	134.72
disc3	45.809	3771.3	76.614	77.686	7090.4	114.56

Table 14.2: Second part of the raw data for the bit-rate model.

Name of sequence	Sobel Filtered Field			Bit-rates	
	Mean	Variance	RMS	Quality excellent	Quality good
animate1	45.716	3202.1	72.746	6.7	4.0
animate2	62.359	5067.6	94.638	4.9	3.0
fastfilm	47.676	2659.0	70.228	5.9	3.5
cycling	110.41	8152.7	142.63	10.3	5.7
boxing	60.574	4056.9	87.898	5.3	3.8
cross country	56.213	4435.9	87.154	5.2	3.9
soccer	73.842	5244.8	103.42	6.2	3.4
news	67.143	2856.1	85.816	5.1	3.2
talking heads	45.748	2380.9	66.886	5.4	2.1
fgmen	44.111	3367.2	72.890	5.9	2.5
scrolling	83.628	9819.7	129.66	4.8	3.7
egoli	44.757	2915.6	70.134	3.6	2.1
crawls	75.986	5503.4	106.19	4.4	2.6
flower garden	99.675	10316.4	142.30	5.4	2.8
disc3	90.206	8678.3	129.67	5	3.7

Table 14.3: Third part of the raw data for the bit-rate model.

- Frame Difference Mean.
- Frame Difference Variance.
- Sobel filtered Frame Mean.
- Sobel filtered Frame Variance.

14.3 Fitting a Linear model

This section describes the process of fitting a linear model to the data. A linear model is first fitted to all the data to test the validity of a linear model. Cross validation is then applied to the data and the validated linear model fitted.

The following linear model, with x_i representing the measured variables on the video sequences and c_i representing the coefficients, is proposed:

$$y_{pred} = c_0 + c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 + c_5x_5 + c_6x_6 + c_7x_7 \tag{14.1}$$

where c_0 is the DC coefficient.

Table 14.6 explains the variables retained. Table 14.7 lists the coefficient values (generated by MATLAB). Table 14.8 compares the predicted and error values of the linear model for excellent and good quality bit-rates respectively.

Variable No.	Correlation Coefficients for all input data.								
	1	2	3	4	5	6	7	8	9
1	1	0.88	0.96	0.02	0.96	0.81	0.93	-0.01	0.14
2	0.88	1	0.97	0.03	0.81	0.86	0.86	0.01	0.08
3	0.96	0.97	1	0.02	0.91	0.86	0.93	-0.01	0.12
4	0.02	0.03	0.02	1	0.12	0.24	0.18	0.99	-0.03
5	0.96	0.81	0.91	0.12	1	0.84	0.96	0.09	0.23
6	0.81	0.86	0.86	0.24	0.84	1	0.95	0.21	0.42
7	0.93	0.86	0.93	0.18	0.96	0.95	1	0.15	0.34
8	-0.01	0.01	-0.01	0.99	0.09	0.21	0.15	1	-0.03
9	0.14	0.08	0.12	-0.03	0.23	0.42	0.34	-0.03	1
10	0.44	0.37	0.43	-0.34	0.55	0.52	0.56	-0.36	0.39
11	0.56	0.49	0.56	-0.29	0.65	0.65	0.67	-0.32	0.43
12	0.14	0.05	0.12	-0.23	0.23	0.35	0.31	-0.24	0.69
13	0.2	0.18	0.22	-0.48	0.25	0.38	0.33	-0.48	0.75
14	0.29	0.24	0.3	-0.32	0.34	0.48	0.44	-0.33	0.69
15	0.21	0.09	0.18	-0.2	0.3	0.4	0.37	-0.22	0.68
16	0.17	0.14	0.18	-0.47	0.24	0.35	0.31	-0.48	0.75
17	0.21	0.14	0.2	-0.32	0.3	0.41	0.37	-0.33	0.72

Table 14.4: First part of correlation coefficients for all variables.

Variable No.	Correlation Coefficients for all input data.							
	10	11	12	13	14	15	16	17
1	0.44	0.56	0.14	0.2	0.29	0.21	0.17	0.21
2	0.37	0.49	0.05	0.18	0.24	0.09	0.14	0.14
3	0.43	0.56	0.12	0.22	0.3	0.18	0.18	0.2
4	-0.34	-0.29	-0.23	-0.48	-0.32	-0.2	-0.47	-0.32
5	0.55	0.65	0.23	0.25	0.34	0.3	0.24	0.3
6	0.52	0.65	0.35	0.38	0.48	0.4	0.35	0.41
7	0.56	0.67	0.31	0.33	0.44	0.37	0.31	0.37
8	-0.36	-0.32	-0.24	-0.48	-0.33	-0.22	-0.48	-0.33
9	0.39	0.43	0.69	0.75	0.69	0.68	0.75	0.72
10	1	0.97	0.59	0.66	0.63	0.62	0.7	0.67
11	0.97	1	0.65	0.7	0.7	0.68	0.72	0.72
12	0.59	0.65	1	0.88	0.91	0.99	0.88	0.97
13	0.66	0.7	0.88	1	0.95	0.87	0.98	0.94
14	0.63	0.7	0.91	0.95	1	0.91	0.9	0.93
15	0.62	0.68	0.99	0.87	0.91	1	0.88	0.97
16	0.7	0.72	0.88	0.98	0.9	0.88	1	0.96
17	0.67	0.72	0.97	0.94	0.93	0.97	0.96	1

Table 14.5: Second part of the correlation coefficients for all variables.

Variable	Physical Parameter
y_{pred}	Bit-rate
x_1	Motion Frame Mean
x_2	Motion Frame Variance
x_3	Number of active (non-zero) Motion Vectors
x_4	Frame Difference Mean
x_5	Frame Difference Variance
x_6	Sobel Frame Mean
x_7	Sobel Frame Variance

Table 14.6: Description of Variables.

Coefficient	Excellent Quality Value	Good Quality Value
c_0	-30.47	-3.689
c_1	90.06×10^{-3}	70.79×10^{-3}
c_2	-0.1382×10^{-3}	55.98×10^{-6}
c_3	63.39×10^{-6}	5.957×10^{-6}
c_4	-44.10×10^{-3}	-17.15×10^{-3}
c_5	-0.5257×10^{-3}	-4.711×10^{-6}
c_6	41.13×10^{-3}	31.87×10^{-3}
c_7	0.3169×10^{-3}	-11.12×10^{-6}

Table 14.7: List of Coefficient values for the Linear Model.

Excellent Quality			Good Quality		
Actual	Predicted	% Error	Actual	Predicted	%Error
6.7	4.884	27.1	4.0	2.95	26.25
4.9	6.353	-29.66	3.0	3.516	-17.2
5.9	4.77	19.16	3.5	2.699	22.88
10.3	8.296	19.46	5.7	5.018	11.96
5.3	5.641	-6.433	3.8	3.295	13.28
5.2	6.049	-16.32	3.9	3.958	-1.499
6.2	6.184	0.2515	3.4	3.3	2.932
5.1	6.135	-20.29	3.2	3.908	-22.12
5.4	4.639	14.1	2.1	2.387	-13.64
5.9	5.32	9.827	2.5	3.147	-25.88
4.8	4.801	-0.01233	3.7	3.658	1.141
3.6	3.846	-6.842	2.1	2.144	-2.09
4.4	5.983	-35.97	2.6	3.055	-17.49
5.4	5.495	-1.756	2.8	2.873	-2.617
5	5.705	-14.1	3.7	4.092	-10.59

Table 14.8: Comparison of Actual, Predicted and Error bit-rates for the Linear Model.

The scatter-plots of predicted versus actual bit-rates and the percentage error of the predicted bit-rates (Figures 14.1, 14.2, 14.3 and 14.4) show a poor fit: the highest error is approximately 30%. The percentage error is calculated as follows for a particular observation: $\frac{(\text{Observation}-\text{Predicted})*100}{\text{Observation}}$. The Figures illustrating the error (Figs 14.2 and 14.4) show the percentage error per *observation.*

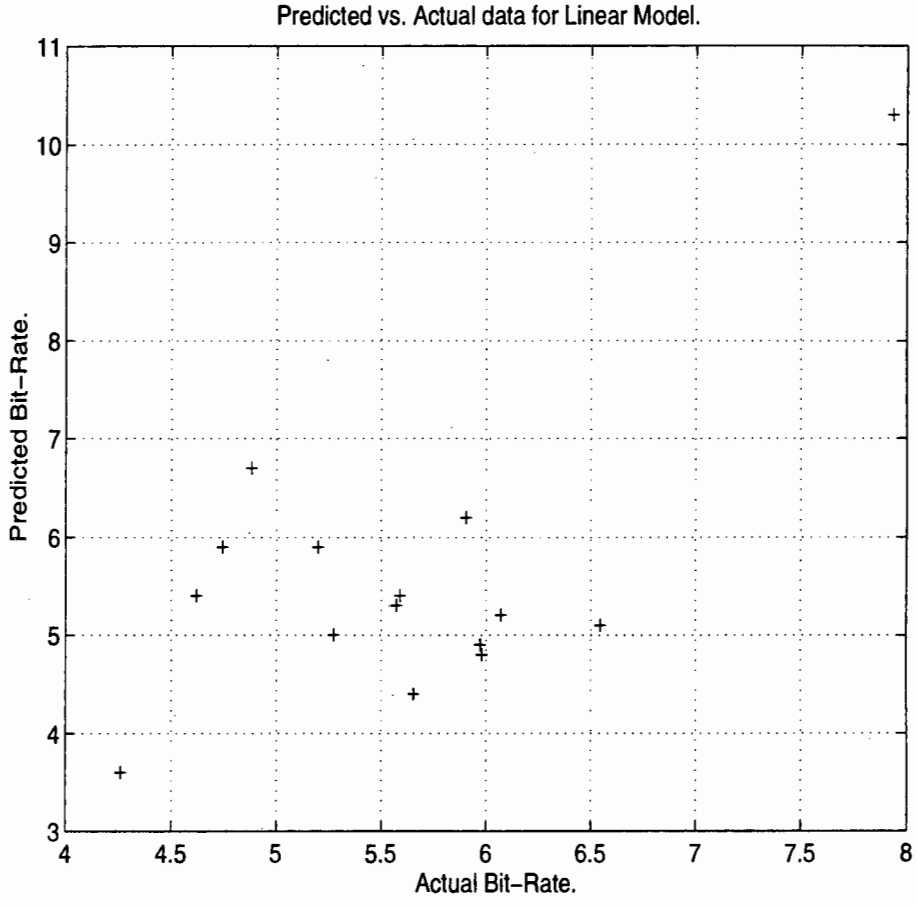


Figure 14.1: Performance of the Linear Model for the Excellent Quality bit-rates.

14.3.1 Validated Linear Model

This subsection describes the application of cross-validation to the Linear Model generation process. The Linear Model is tested (validated) by removing the second, sixth and fourteenth data points in the model generation stage and observing the predicted error at these points in the model testing stage. The stability of the model generation stage is ensured by rejecting any variable (term) with a correlation coefficient greater than 0.9 (see section 13.2 for a justification). See Table 14.11 for a description of the variables used. Table 14.9 lists the coefficients and associated confidence intervals and Table 14.10 lists the percentage error on

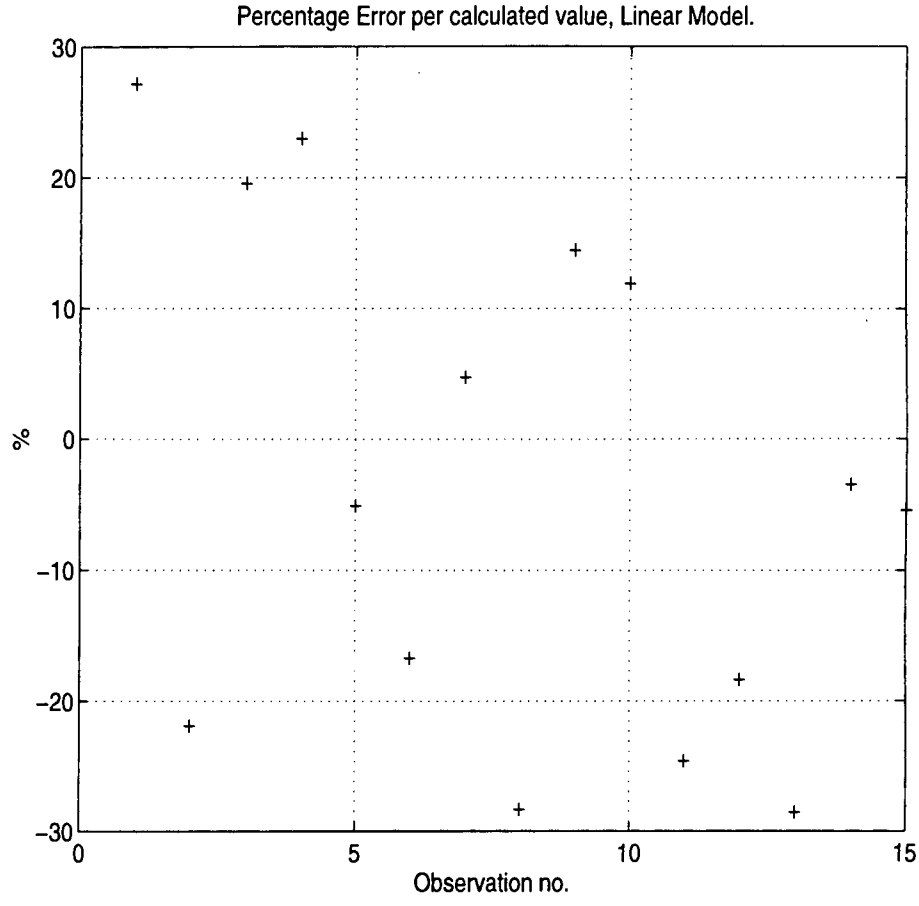


Figure 14.2: Error performance of the Linear Model for the Excellent Quality bit-rates. The horizontal axis is not a variable. Data points occur in the order corresponding to entries in Table 14.8.

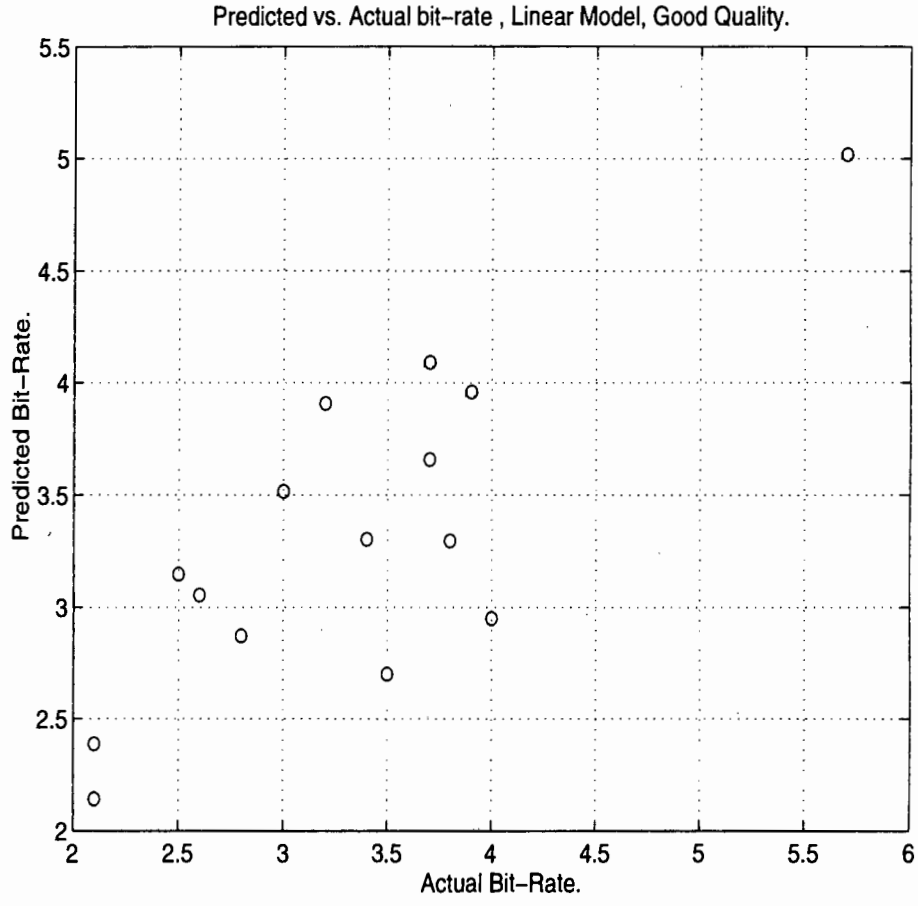


Figure 14.3: Performance of the Linear Model for the Good Quality bit-rates.

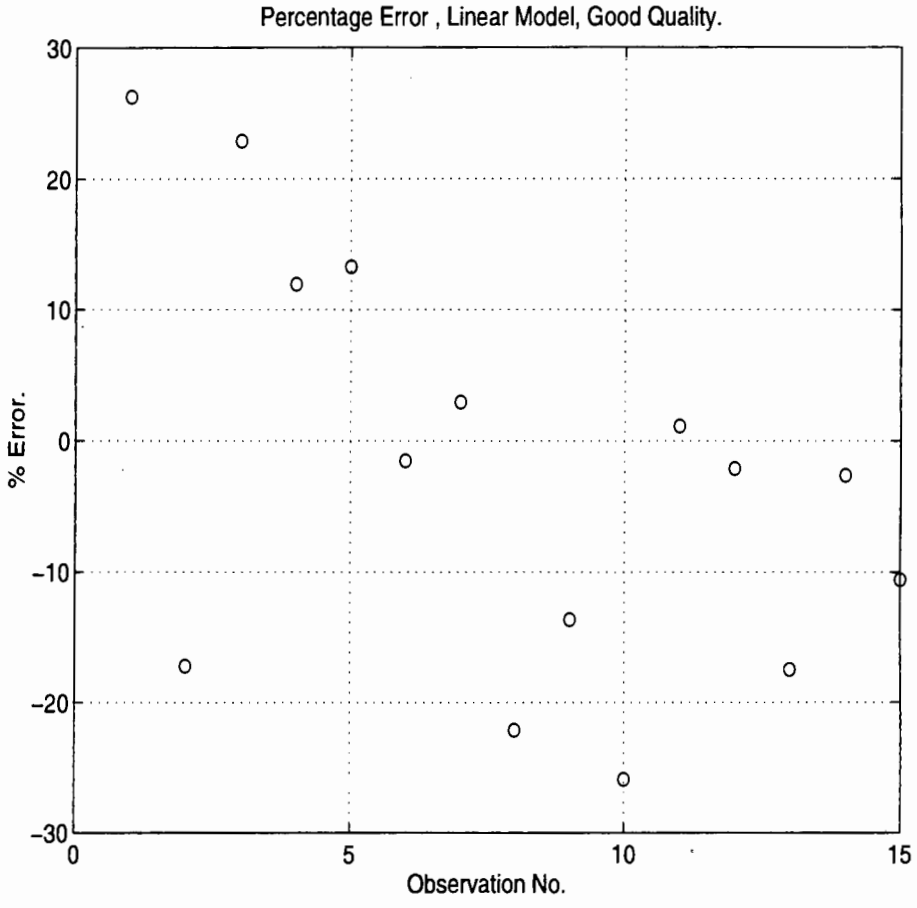


Figure 14.4: Error performance of the Linear Model for the Good Quality bit-rates. The horizontal axis is not a variable. Data points occur in the order corresponding to entries in Table 14.8.

Coefficient	Excellent Quality		Good Quality	
	Value	Confidence Interval	Value	Confidence Interval
c_0	-25.58	$-26.79 \cdots -24.37$	-4.744	$-5.376 \cdots -4.112$
$c_1(\times 10^{-3})$	162.1	$156.1 \cdots 168.1$	78.21	$75.07 \cdots 81.34$
$c_2(\times 10^{-6})$	48.77	$46.36 \cdots 51.17$	8.043	$6.787 \cdots 9.299$
$c_3(\times 10^{-3})$	-110.1	$-115.4 \cdots -104.8$	-17.00	$-19.75 \cdots -14.25$
c_4	**reject**	**reject**	**reject**	**reject**
$c_5(\times 10^{-3})$	67.55	$65.39 \cdots 69.72$	30.05	$28.92 \cdots 31.17$
c_6	**reject**	**reject**	**reject**	**reject**
c_7	**reject**	**reject**	**reject**	**reject**

Table 14.9: List of Coefficient values for the Validated Linear Model.

the predicted values of the model. Figures 14.5, 14.6, 14.7 and 14.8 show the relationship between predicted and actual values and the percentage error of the excellent and good video quality bit-rate linear models. The error is shown per *observation.*

14.4 Fitting a Quadratic model

The task of fitting a quadratic model is approached as follows; first a model is fitted to all the data to see if the model explains the observations of the bit-rate with sufficient accuracy, then a new model is fitted to all the data less a validation set and the performance of the model examined. It is ensured that the model can be generated (i.e., that the matrix is invertible) by rejecting all terms with correlation coefficients greater than 0.8 for the linear model data and then eliminating all terms with correlation coefficients greater than 0.6 in the quadratic model data. This ensures that the model can be generated (see section 13.2 for a justification of this).

The following quadratic model is proposed with the variables labelled as in Table 14.11. As before x_i refers to the variables measured and c_{ij} refers to the model coefficients:

$$
\begin{aligned}
y_{pred} \;=\; & c_{00} + c_{10}x_1 + c_{11}x_1^2 + \\
& c_{20}x_2 + c_{21}x_2x_1 + c_{22}x_2^2 + \\
& c_{30}x_3 + c_{31}x_3x_1 + c_{32}x_3x_2 + c_{33}x_3^2 + \\
& c_{40}x_4 + c_{41}x_4x_1 + c_{42}x_4x_2 + c_{43}x_4x_3 + c_{44}x_4^2 \\
& c_{50}x_5 + c_{51}x_5x_1 + c_{52}x_5x_2 + c_{53}x_5x_3 + c_{54}x_5x_4 + c_{55}x_5^2 \\
& c_{60}x_6 + c_{61}x_6x_1 + c_{62}x_6x_2 + c_{63}x_6x_3 + c_{64}x_6x_4 + c_{65}x_6x_5 + c_{66}x_6^2 \\
& c_{70}x_7 + c_{71}x_7x_1 + c_{72}x_7x_2 + c_{73}x_7x_3 + c_{74}x_7x_4 + c_{75}x_7x_5 + c_{76}x_7x_6 + c_{77}x_7^2
\end{aligned}
$$

Excellent Quality			Good Quality		
Actual	Predicted	% Error	Actual	Predicted	%Error
6.7	5.14	23.28	4.0	3.001	24.97
4.9	6.794	-38.65	3.0	3.59	-19.66
5.9	4.85	17.79	3.5	2.775	20.7
10.3	8.619	16.32	5.7	5.078	10.92
5.3	5.994	-13.09	3.8	3.396	10.63
5.2	6.953	-33.7	3.9	4.05	-3.838
6.2	6.154	0.7427	3.4	3.334	1.948
5.1	7.192	-41.03	3.2	3.989	-24.66
5.4	4.631	14.24	2.1	2.422	-15.35
5.9	5.802	1.66	2.5	3.233	-29.31
4.8	4.839	-0.8133	3.7	3.668	0.8636
3.6	4.098	-13.83	2.1	2.157	-2.734
4.4	5.574	-26.68	2.6	3.048	-17.23
5.4	-0.4314	108	2.8	2.928	-4.557
5	5.707	-14.13	3.7	4.198	-13.47

Table 14.10: Comparison of Actual, Predicted and Error bit-rates For the Validated Linear Model.

Variable	Physical Parameter
y_{pred}	Bit-rate
x_1	Motion Frame Mean
x_2	Motion Frame Variance
x_3	Number of active (non-zero) Motion Vectors
x_4	Frame Difference Mean
x_5	Frame Difference Variance
x_6	Sobel Frame Mean
x_7	Sobel Frame Variance

Table 14.11: Description of Variables.

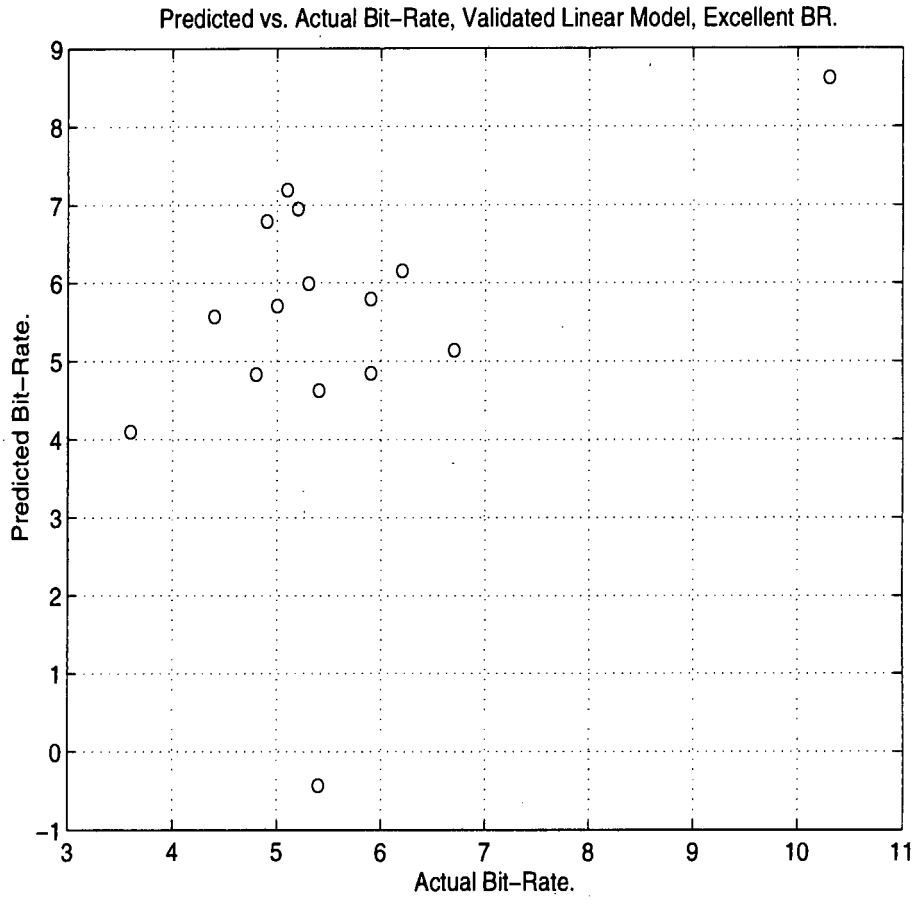


Figure 14.5: Performance of the Validated Linear Model for the Excellent Quality bit-rates.

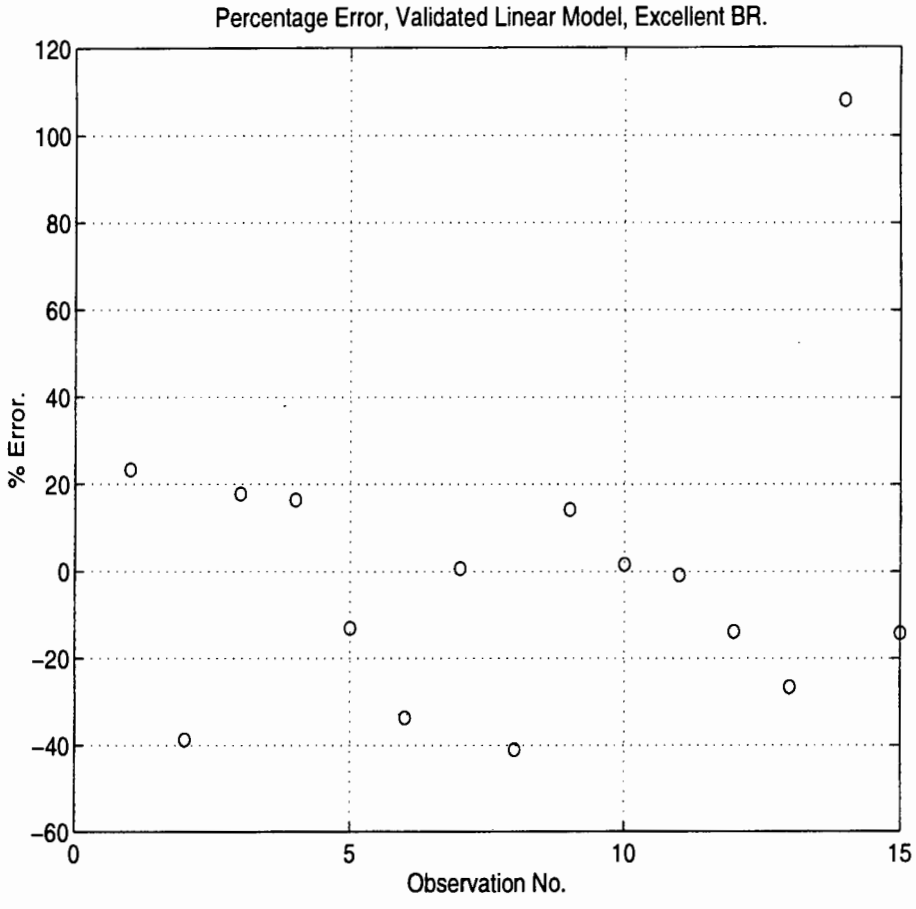


Figure 14.6: Error performance of the Validated Linear Model for the Excellent Quality bit-rates. The horizontal axis is not a variable. Data points occur in the order corresponding to entries in Table 14.10.

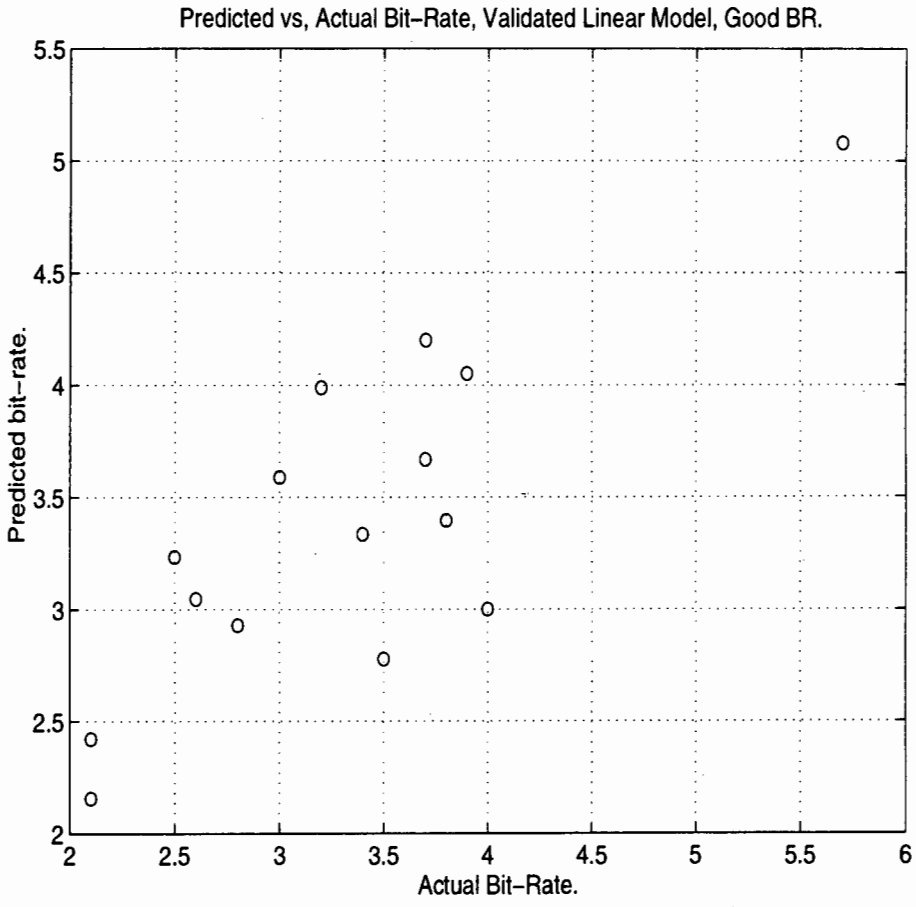


Figure 14.7: Performance of the Validated Linear Model for the Good Quality bit-rates.

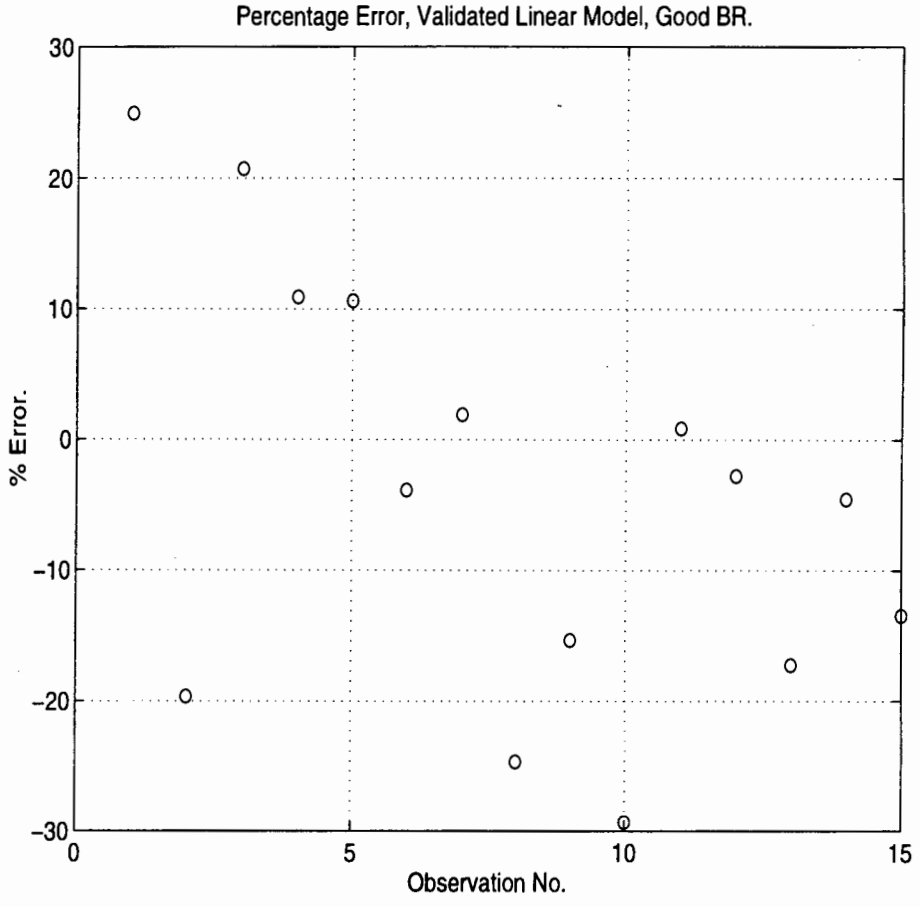


Figure 14.8: Error performance of the Validated Linear Model for the Good Quality bit-rates. The horizontal axis is not a variable. Data points occur in the order corresponding to entries in Table 14.10.

Where c_{00} is the DC coefficient.

Table 14.12 compares the model coefficients and confidence intervals for the Excellent and Good video quality bit-rates. Note **reject** denotes coefficients removed because their variable products correlate with other variable products. Table 14.13 compares the data, predicted bit-rates and the percentage error.

The following scatter-plots of the estimated bit-rates versus the actual bit-rates and the percentage error of the predicted values (Figures 14.9, 14.10, 14.11, and 14.12) show the quadratic model predicts the bit-rates within 37% for the excellent video quality sequence and within 21% for the good video quality sequence.

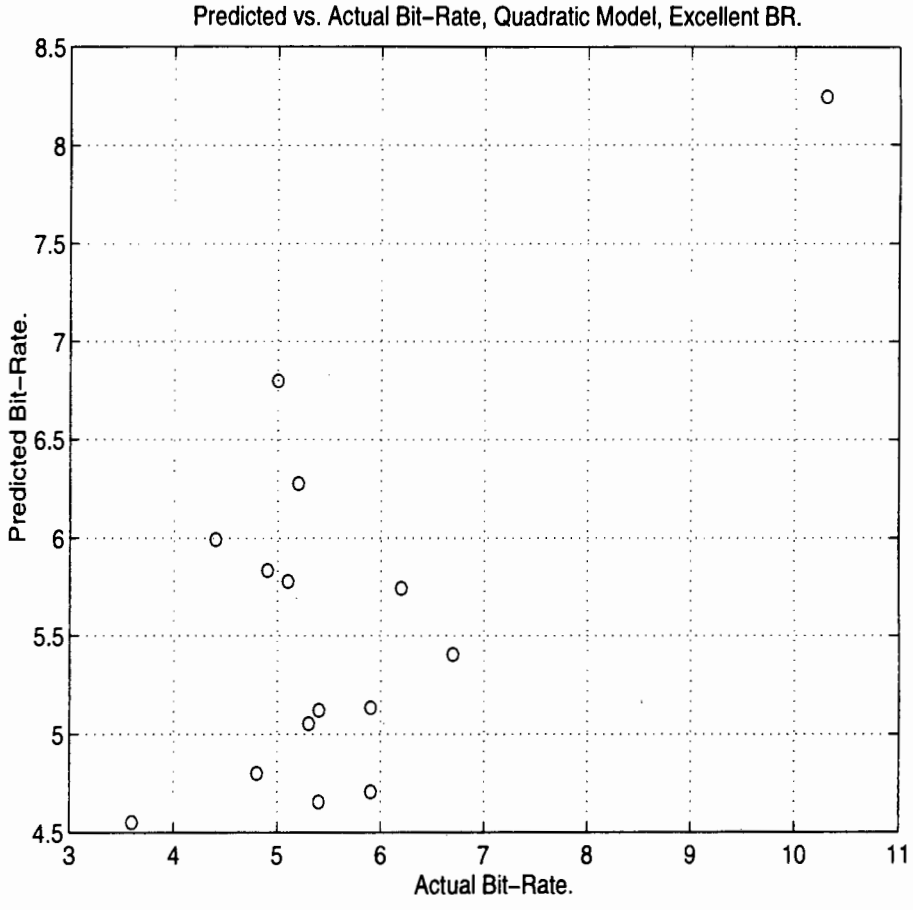


Figure 14.9: Performance of the Quadratic Model for the Excellent Quality bit-rates.

14.4.1 Validation of the Quadratic model

This subsection describes the cross-validation of the Quadratic Model fitting process. The model is tested (validated) by removing three data points from the data at random, generating the coefficients and testing the predicted bit-rates for these data points (sequences). The data

Coefficient	Excellent Bit-Rate		Good Bit-Rate	
	Value	Confidence interval	Value	Confidence interval
c_{00}	-16.38	$-17.46 \cdots -15.3$	-1.274	$-1.787 \cdots -0.7611$
$c_{10}(\times 10^{-3})$	-19.84	$-24.1 \cdots -15.58$	26.84	$24.82 \cdots 28.86$
c_{11}	**reject**	**reject**	**reject**	**reject**
c_{20}	**reject**	**reject**	**reject**	**reject**
c_{21}	**reject**	**reject**	**reject**	**reject**
c_{22}	**reject**	**reject**	**reject**	**reject**
$c_{30}(\times 10^{-6})$	41.96	$39.74 \cdots 44.18$	4.795	$3.74 \cdots 5.85$
c_{31}	**reject**	**reject**	**reject**	**reject**
c_{32}	**reject**	**reject**	**reject**	**reject**
c_{33}	**reject**	**reject**	**reject**	**reject**
$c_{40}(\times 10^{-3})$	-21.83	$-23.2 \cdots -20.46$	-13.11	$-13.76 \cdots -12.46$
c_{41}	**reject**	**reject**	**reject**	**reject**
c_{42}	**reject**	**reject**	**reject**	**reject**
c_{43}	**reject**	**reject**	**reject**	**reject**
c_{44}	**reject**	**reject**	**reject**	**reject**
c_{50}	**reject**	**reject**	**reject**	**reject**
c_{51}	**reject**	**reject**	**reject**	**reject**
c_{52}	**reject**	**reject**	**reject**	**reject**
c_{53}	**reject**	**reject**	**reject**	**reject**
c_{54}	**reject**	**reject**	**reject**	**reject**
c_{55}	**reject**	**reject**	**reject**	**reject**
c_{60}	**reject**	**reject**	**reject**	**reject**
c_{61}	**reject**	**reject**	**reject**	**reject**
$c_{62}(\times 10^{-6})$	21.11	$20.47 \cdots 21.74$	12.17	$11.87 \cdots 12.48$
c_{63}	**reject**	**reject**	**reject**	**reject**
c_{64}	**reject**	**reject**	**reject**	**reject**
c_{65}	**reject**	**reject**	**reject**	**reject**
c_{66}	**reject**	**reject**	**reject**	**reject**
c_{70}	**reject**	**reject**	**reject**	**reject**
c_{71}	**reject**	**reject**	**reject**	**reject**
c_{72}	**reject**	**reject**	**reject**	**reject**
c_{73}	**reject**	**reject**	**reject**	**reject**
c_{74}	**reject**	**reject**	**reject**	**reject**
c_{75}	**reject**	**reject**	**reject**	**reject**
c_{76}	**reject**	**reject**	**reject**	**reject**
c_{77}	**reject**	**reject**	**reject**	**reject**

Table 14.12: List of Coefficient values for the Quadratic Model.

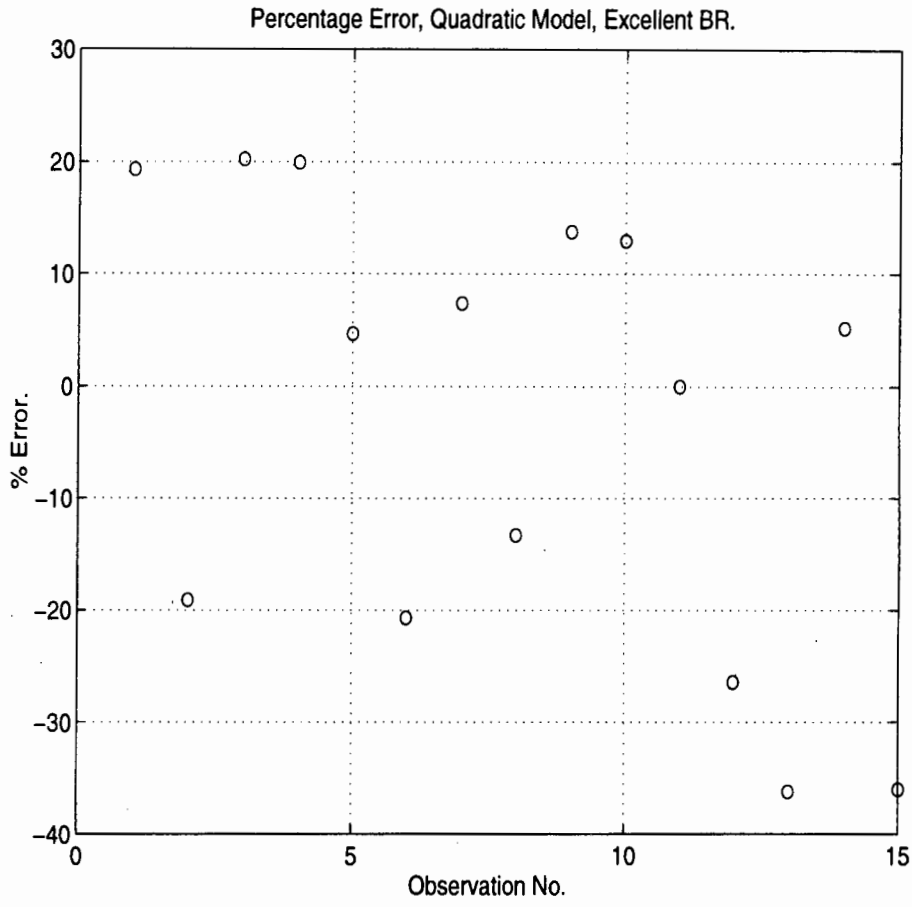


Figure 14.10: Error performance of the Quadratic Model for the Excellent Quality bit-rates. The horizontal axis is not a variable. Data points occur in the order corresponding to entries in Table 14.13.

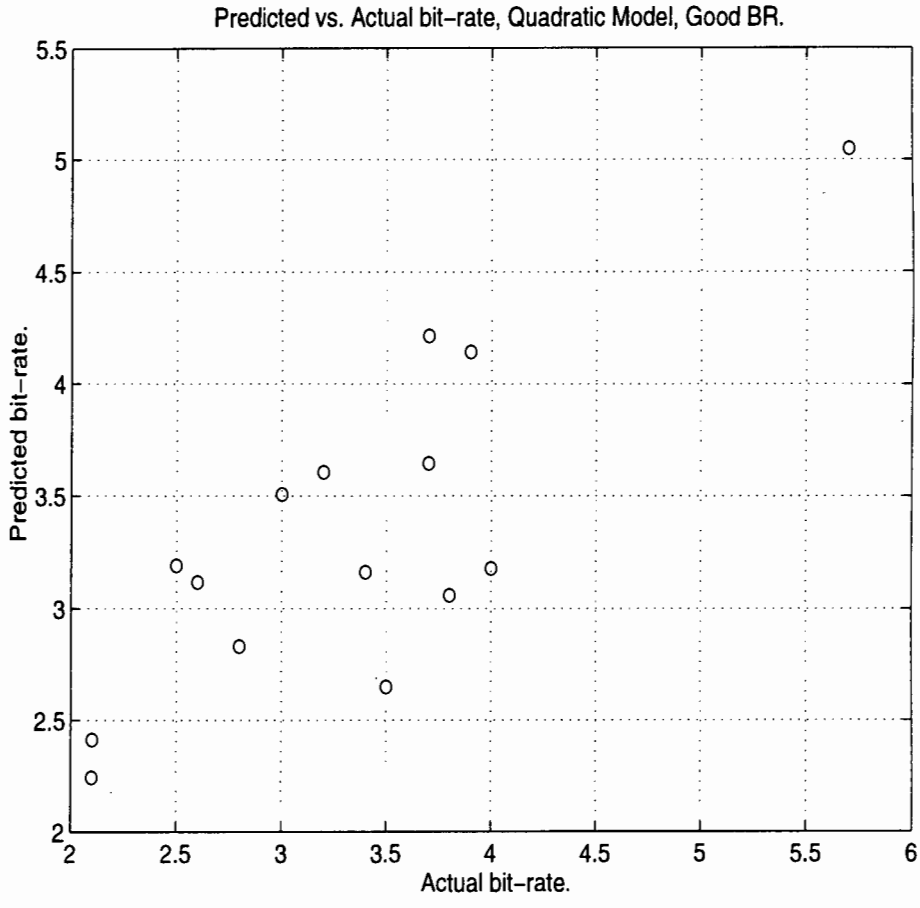


Figure 14.11: Performance of the Quadratic Model for the Good Quality bit-rates.

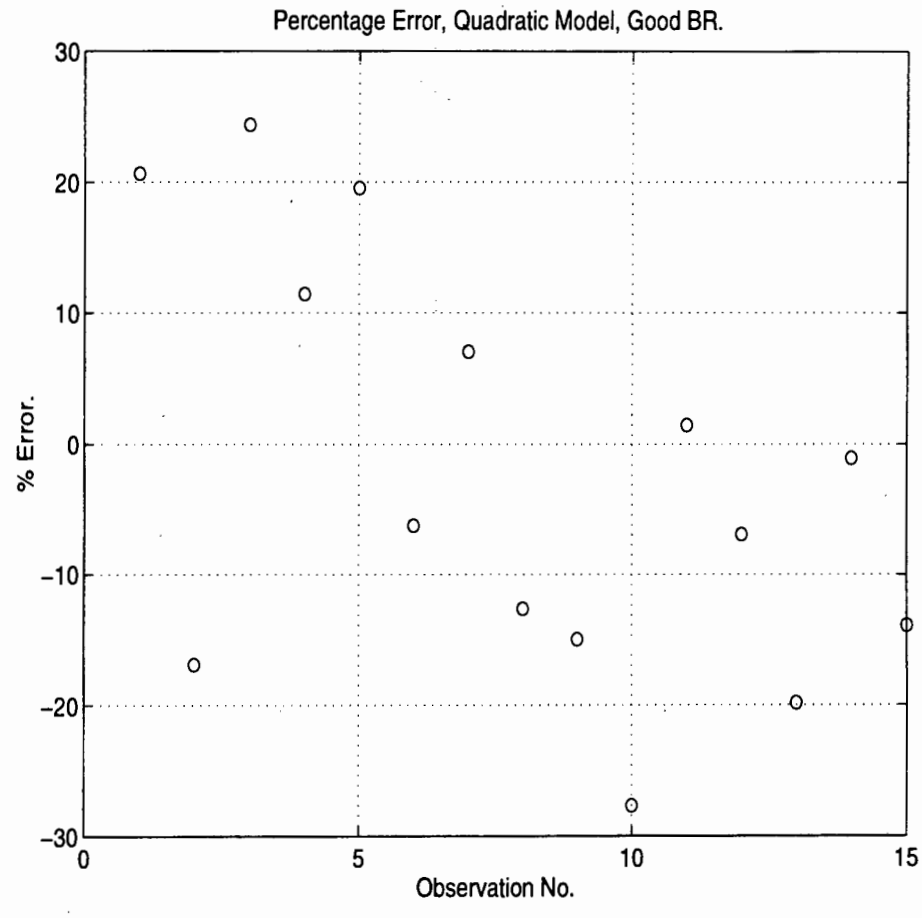


Figure 14.12: Error performance of the Quadratic Model for the Good Quality bit-rates. The horizontal axis is not a variable. Data points occur in the order corresponding to entries in Table 14.13.

Excellent Quality			Good Quality		
Actual	Predicted	% Error	Actual	Predicted	%Error
6.7	5.404	19.34	4.0	3.175	20.62
4.9	5.835	-19.09	3.0	3.506	-16.87
5.9	4.707	20.22	3.5	2.648	24.35
10.3	8.245	19.95	5.7	5.048	11.44
5.3	5.054	4.636	3.8	3.057	19.54
5.2	6.276	-20.7	3.9	4.143	-6.241
6.2	5.744	7.352	3.4	3.161	7.018
5.1	5.778	-13.29	3.2	3.604	-12.63
5.4	4.655	13.8	2.1	2.414	-14.96
5.9	5.135	12.96	2.5	3.192	-27.66
4.8	4.8	0.005039	3.7	3.646	1.452
3.6	4.551	-26.43	2.1	2.245	-6.916
4.4	5.992	-36.19	2.6	3.115	-19.82
5.4	5.122	5.157	2.8	2.83	-1.08
5	6.8	-36	3.7	4.214	-13.9

Table 14.13: Comparison of Actual, Predicted and Error bit-rates for the Quadratic Model.

points removed are: the second, sixth and the twelfth, i.e., the data points corresponding to the sequences *Animate2, Cross Country* and *Egoli.* The data used is identical to the data used for creating the linear model.

Table 14.14 compares the model coefficients and confidence intervals for the Excellent and Good video quality bit-rates. Table 14.15 compares the data, predicted bit-rates and the percentage error.

The following scatter-plots of the estimated bit-rates vs. the actual bit-rates and the percentage error of the predicted values (Figures 14.13, 14.14, 14.15 and 14.16) show the quadratic model predicts the bit-rates within 38% for the excellent video sequence quality and within 37% for the good video sequence quality.

Chapter 15 attempts to model the data using PCA followed by Jackknifing to derive confidence intervals for the model coefficients.

Coefficient	Excellent Quality		Good Quality	
	Value	Confidence Interval	Value	Confidence Interval
c_{00}	-12.72	$-14.05 \cdots -11.38$	-2.521	$-3.11 \cdots -1.933$
$c_{10}(\times 10^{-3})$	107.7	$100.9 \cdots 114.4$	45.2	$42.28 \cdots 48.13$
c_{11}	**reject**	**reject**	**reject**	**reject**
c_{20}	**reject**	**reject**	**reject**	**reject**
c_{21}	**reject**	**reject**	**reject**	**reject**
c_{22}	**reject**	**reject**	**reject**	**reject**
$c_{30}(\times 10^{-6})$	29.61	$26.88 \cdots 32.35$	6.401	$5.2 \cdots 7.602$
c_{31}	**reject**	**reject**	**reject**	**reject**
c_{32}	**reject**	**reject**	**reject**	**reject**
c_{33}	**reject**	**reject**	**reject**	**reject**
$c_{40}(\times 10^{-3})$	7.198	$2.86 \cdots 11.54$	-28.3	$-31.16 \cdots -25.44$
c_{41}	**reject**	**reject**	**reject**	**reject**
c_{42}	**reject**	**reject**	**reject**	**reject**
c_{43}	**reject**	**reject**	**reject**	**reject**
c_{44}	**reject**	**reject**	**reject**	**reject**
c_{50}	**reject**	**reject**	**reject**	**reject**
c_{51}	**reject**	**reject**	**reject**	**reject**
c_{52}	**reject**	**reject**	**reject**	**reject**
c_{53}	**reject**	**reject**	**reject**	**reject**
c_{54}	**reject**	**reject**	**reject**	**reject**
c_{55}	**reject**	**reject**	**reject**	**reject**
c_{60}	**reject**	**reject**	**reject**	**reject**
c_{61}	**reject**	**reject**	**reject**	**reject**
$c_{62}(\times 10^{-6})$	**reject**	**reject**	13.47	$13.01 \cdots 13.93$
c_{63}	**reject**	**reject**	**reject**	**reject**
c_{64}	**reject**	**reject**	**reject**	**reject**
c_{65}	**reject**	**reject**	**reject**	**reject**
c_{66}	**reject**	**reject**	**reject**	**reject**
c_{70}	**reject**	**reject**	**reject**	**reject**
c_{71}	**reject**	**reject**	**reject**	**reject**
c_{72}	**reject**	**reject**	**reject**	**reject**
c_{73}	**reject**	**reject**	**reject**	**reject**
c_{74}	**reject**	**reject**	**reject**	**reject**
c_{75}	**reject**	**reject**	**reject**	**reject**
c_{76}	**reject**	**reject**	**reject**	**reject**
c_{77}	**reject**	**reject**	**reject**	**reject**

Table 14.14: List of Coefficient values for Validated Quadratic Model.

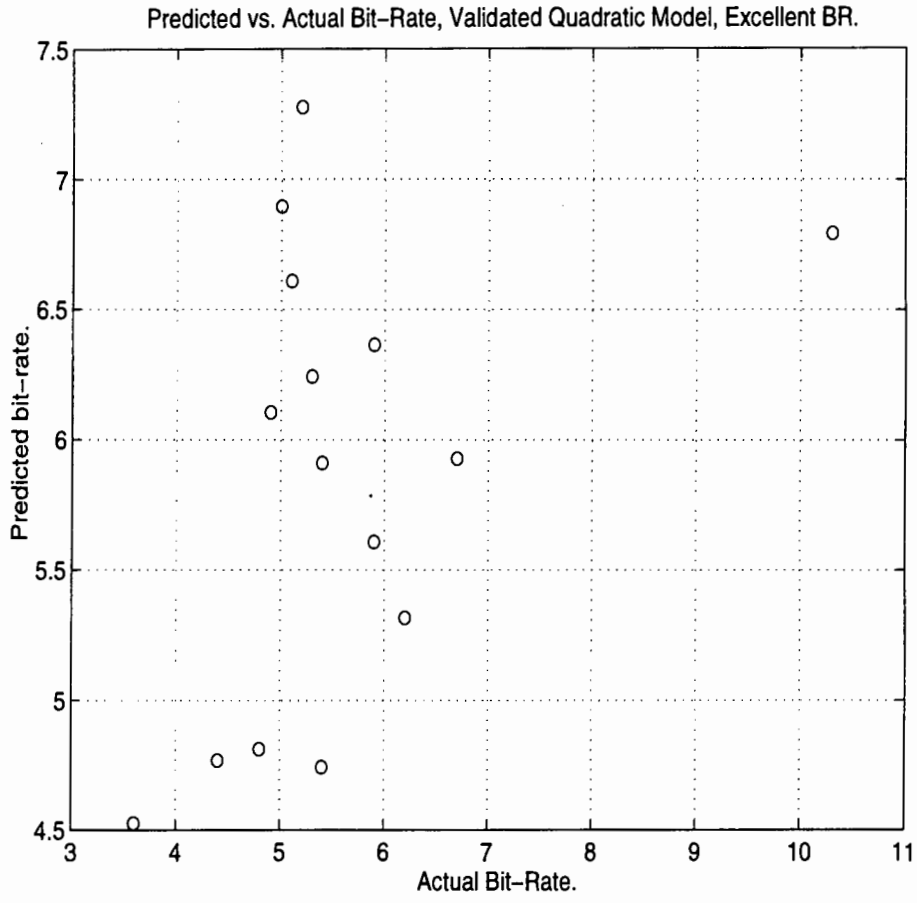


Figure 14.13: Performance of the Validated Quadratic Model for the Excellent Quality bit-rates.

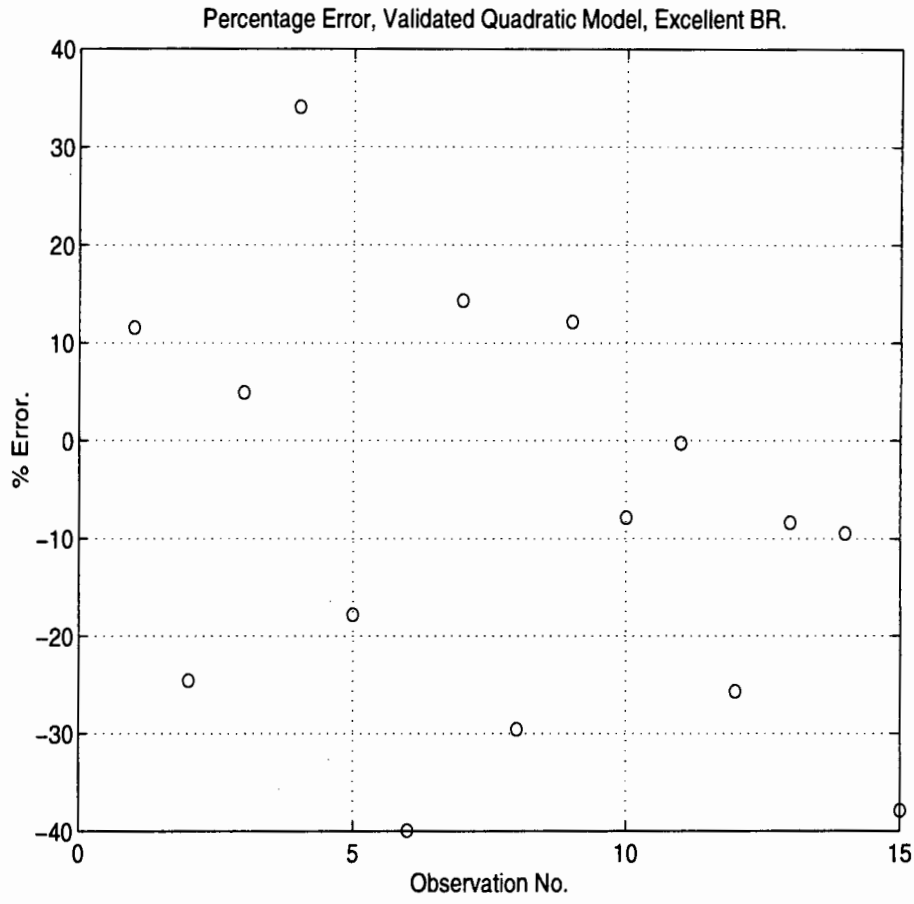


Figure 14.14: Error performance of the Validated Quadratic Model for the Excellent Quality bit-rates. The horizontal axis is not a variable. Data points occur in the order corresponding to entries in Table 14.15.

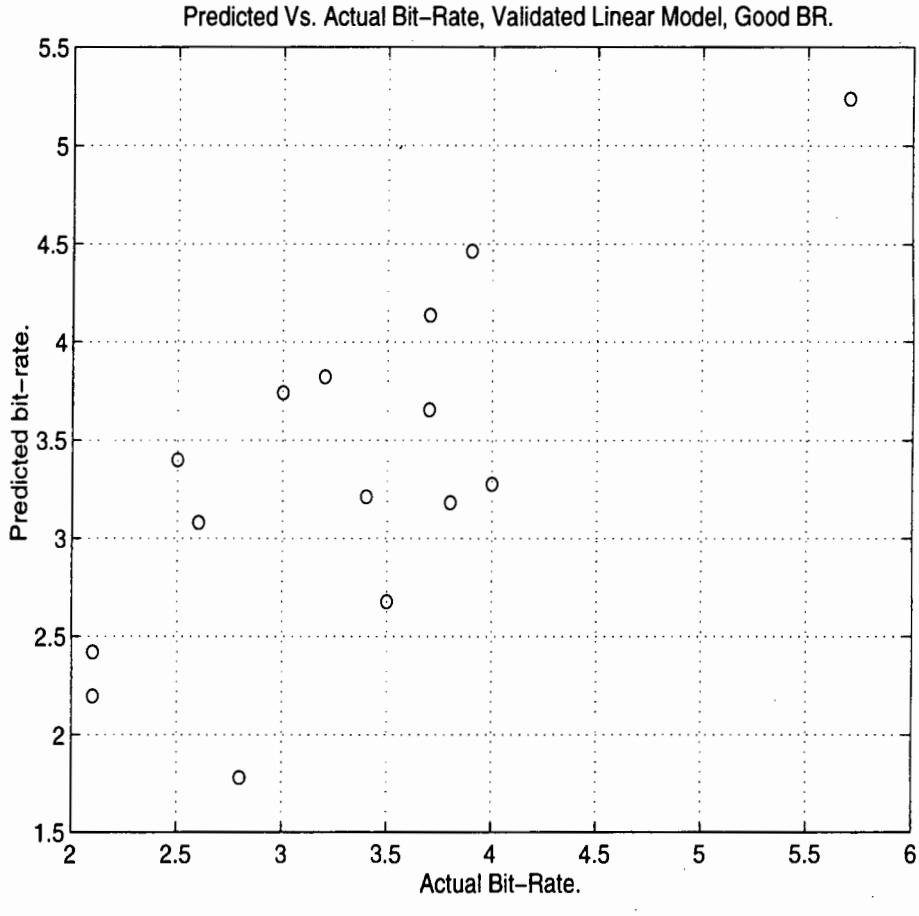


Figure 14.15: Performance of the Validated Quadratic Model for the Good Quality bit-rates.

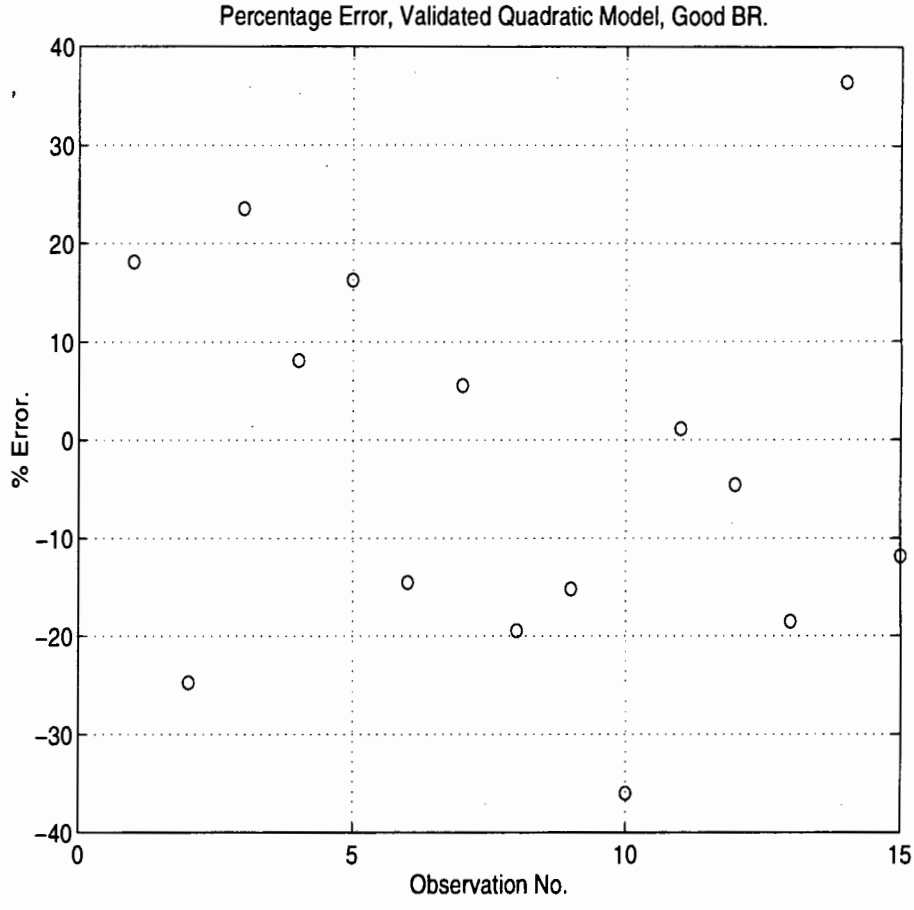


Figure 14.16: Error performance of the Validated Quadratic Model for the Good Quality bit-rates. The horizontal axis is not a variable. Data points occur in the order corresponding to entries in Table 14.15.

Excellent Quality			Good Quality		
Actual	Predicted	% Error	Actual	Predicted	%Error
6.7	5.929	11.51	4.0	3.276	18.1
4.9	6.105	-24.59	3.0	3.742	-24.74
5.9	5.608	4.951	3.5	2.676	23.53
10.3	6.791	34.06	5.7	5.237	8.123
5.3	6.241	-17.76	3.8	3.182	16.25
5.2	7.277	-39.95	3.9	4.464	-14.45
6.2	5.315	14.27	3.4	3.211	5.552
5.1	6.609	-29.58	3.2	3.823	-19.46
5.4	4.743	12.16	2.1	2.418	-15.16
5.9	6.364	-7.869	2.5	3.4	-36.01
4.8	4.812	-0.2465	3.7	3.659	1.103
3.6	4.525	-25.69	2.1	2.196	-4.586
4.4	4.769	-8.376	2.6	3.082	-18.54
5.4	5.913	-9.496	2.8	1.781	36.41
5	6.894	-37.88	3.7	4.138	-11.85

Table 14.15: Comparison of Actual, Predicted and Error bit-rates for Validated Quadratic Model.

Chapter 15

A Jackknifed Linear Model after Principal Component Analysis

The previous chapter examined the results of fitting Linear and Quadratic Models to the data described. A problem apparent from the previous chapter is that only fifteen data points are used for the model and validation set combined. Better use of the data is made by the Jackknife method which uses all the data, less one point, at each stage in the method. Principal Components Analysis decorrelates the variables, simplifying the model generation step since cross-product terms disappear. PCA offers the opportunity to reduce the dimensionality of the data in a meaningful way as explained in Section 13.6.

This chapter approaches the modelling problem by first performing a Principal Components Analysis to decorrelate the final variables and then generating Jackknifed estimates of the coefficients and error. The data used is identical to the linear data of Chapter 14.

15.1 Principal Components Analysis (PCA)

This section describes the effect of applying PCA to the data. The number of variables retained is determined by comparing the the relative magnitude of the eigenvalues. Examination of the screeplot (Fig. 15.1) indicates the first four values are significant. Table 15.1 lists the four retained variables after PCA.

15.2 Performance of the Jackknifed Linear Model

Since PCA orthogonolizes the variables, cross product terms vanish and the following linear model is possible in three variables:

$$y_{pred} = c_0 + c_1 x_1 + c_2 x_2 + c_3 x_3 + c_4 x_4$$

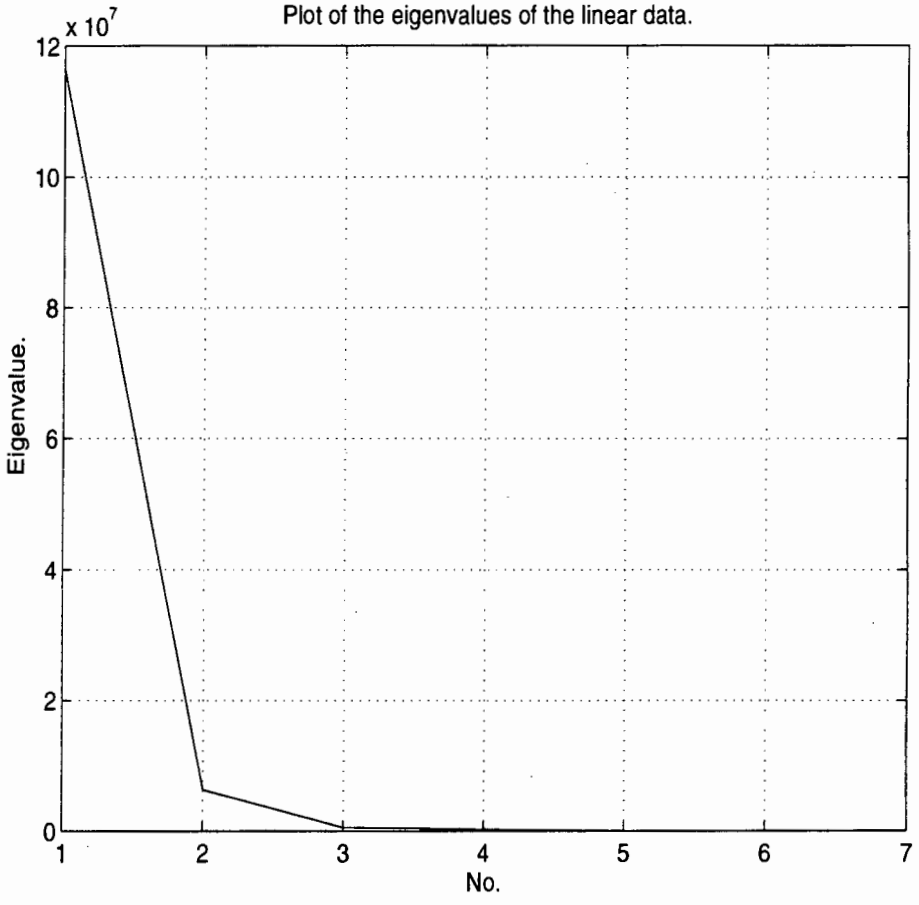


Figure 15.1: Screeplot of the Eigenvalues of the linear data.

x10^5			
-4.8587	0.6868	0.0172	-0.0184
-4.8556	0.6992	0.0238	-0.0186
-4.8656	0.6804	0.0227	-0.0256
-4.8604	0.7368	0.0202	-0.0215
-4.8673	0.6877	0.0227	-0.0234
-4.8570	0.6962	0.0129	-0.0112
-4.8566	0.7010	0.0268	-0.0242
-4.8675	0.6806	0.0202	-0.0214
-4.8391	0.6762	0.0270	-0.0271
-4.8562	0.6861	0.0191	-0.0163
-4.4396	0.6964	0.0203	-0.0223
-4.8519	0.6605	0.0238	-0.0294
-4.8543	0.7047	0.0275	-0.0240
-4.8311	0.7509	0.0377	-0.0209
-4.8589	0.7390	0.0035	-0.0315

Table 15.1: The retained variables after PCA.

Coefficient	Good BR Model	Confidence Interval	Excellent BR Model	Confidence Interval
c_0	-25.72	$-71.41 \cdots 19.97$	-44.09	$-89.13 \cdots 0.9472$
$c_1(\times 10^{-4})$	-0.3649	$-1.287 \cdots 0.5574$	-0.6472	$-1.549 \cdots 0.2549$
$c_2(\times 10^{-4})$	1.882	$1.33 \cdots 2.434$	2.85	$1.868 \cdots 3.833$
$c_3(\times 10^{-4})$	-5.248	$-7.982 \cdots -2.514$	-2.347	$-7.004 \cdots 2.311$
$c_4(\times 10^{-4})$	2.954	$0.1862 \cdots 5.722$	4.506	$-0.3779 \cdots 9.39$

Table 15.2: The Linear Model Coefficients after PCA and the Jackknife.

The model is tested on both the Good and Excellent Quality bit-rates and the Jackknife used to generate an estimate of the average expected error as well as the model coefficient values.

Table 15.2 lists the coefficient values and Jackknifed 95% confidence intervals. Table 15.3 lists the average error and its 95% confidence intervals. Figures 15.2, 15.3, 15.4, 15.5 and Table 15.4 compare the percentage error against the actual values and the predicted versus the actual values for both the Good and Excellent Quality bit-rates.

The average predicted error from the Jackknifed model predicted by the Jackknife is 15.4% and 15.29% and the peak error is 50% and 49% for the Good and Excellent Quality Models respectively.

Model	Error	Confidence Interval
Good BR	15.4	$14.08 \cdots 16.72$
Excellent BR	15.29	$14.21 \cdots 16.38$

Table 15.3: The Linear Model Coefficients after PCA and the Jackknife.

Good BR Model			Excellent BR Model		
Actual BR	Predicted BR	Percentage Error	Actual BR	Predicted BR	Percentage Error
4.0	3.493	12.69	6.7	5.698	14.96
3.0	3.363	-12.1	4.9	5.87	-19.8
3.5	2.895	17.29	5.9	5.109	13.41
5.7	4.187	26.55	10.3	6.921	32.8
3.8	3.102	18.37	5.3	5.423	-2.328
3.9	4.105	-5.254	5.2	6.384	-22.77
3.4	3.077	9.514	6.2	5.603	9.636
3.2	3.164	1.13	5.1	5.374	-5.379
2.1	2.453	-16.79	5.4	4.651	13.87
2.5	3.434	-37.36	5.9	5.715	3.133
3.7	1.866	49.58	4.8	3.011	37.27
2.1	2.301	-9.574	3.6	4.256	-18.23
2.6	3.11	-19.62	4.4	5.69	-29.31
2.8	3.447	-23.11	5.4	6.752	-25.04
3.7	4.811	-30.02	5	6.921	-38.42

Table 15.4: Actual, Predicted and Percentage Error values for the Jackknifed Model.

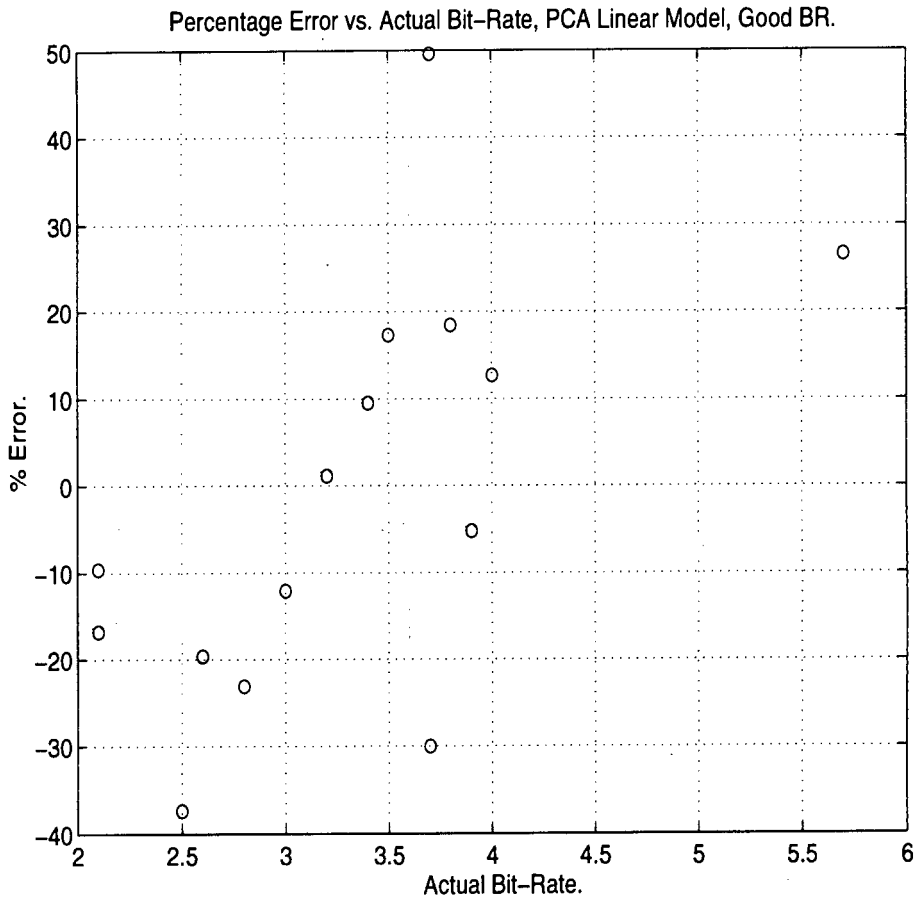


Figure 15.2: Percentage Error versus Actual Data, Good Quality BR.

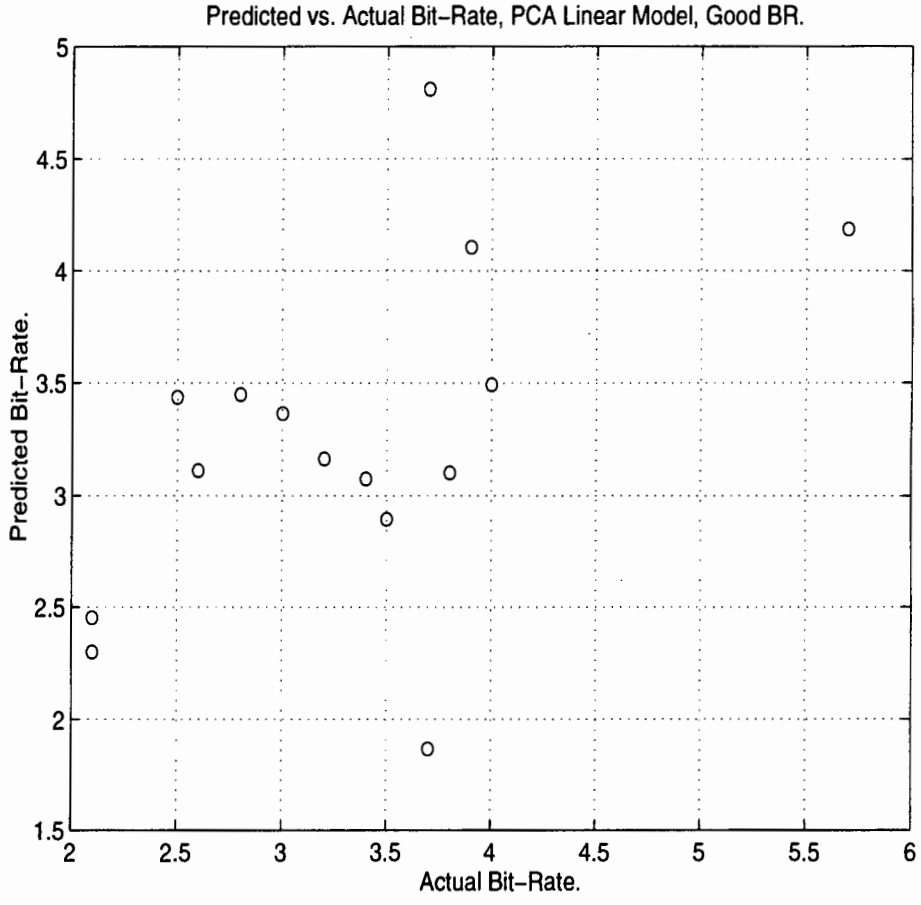


Figure 15.3: Predicted versus Actual Data, Good Quality BR.

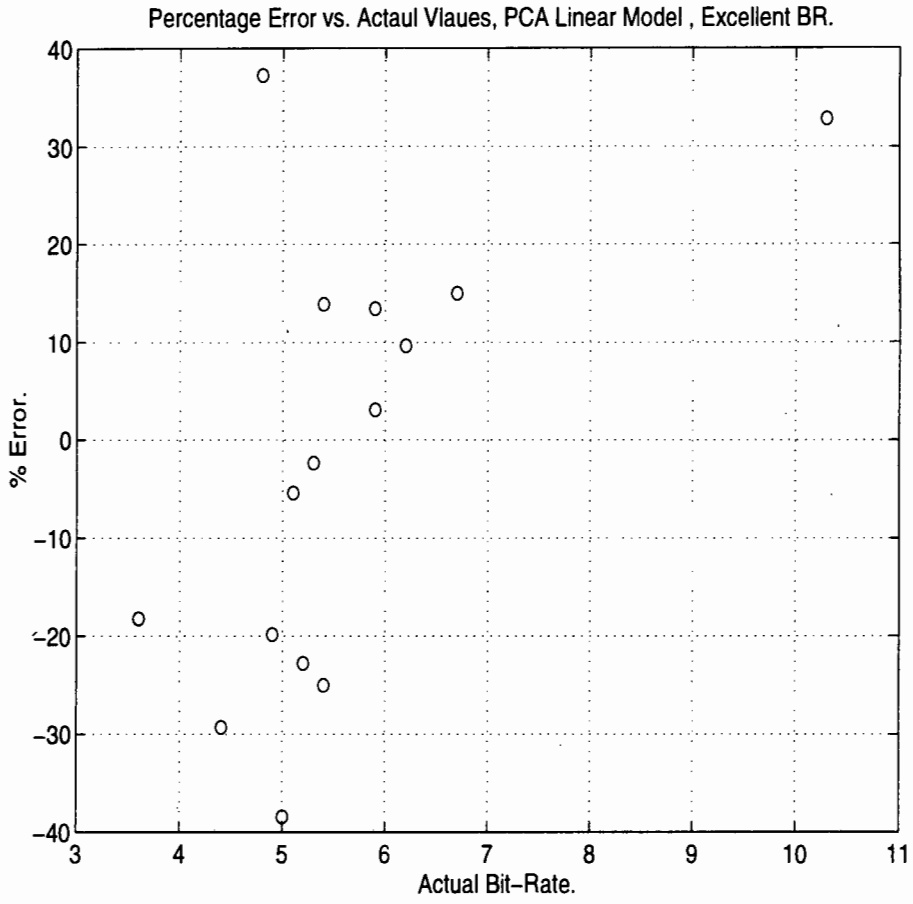


Figure 15.4: Percentage Error versus Actual Data, Excellent Quality BR.

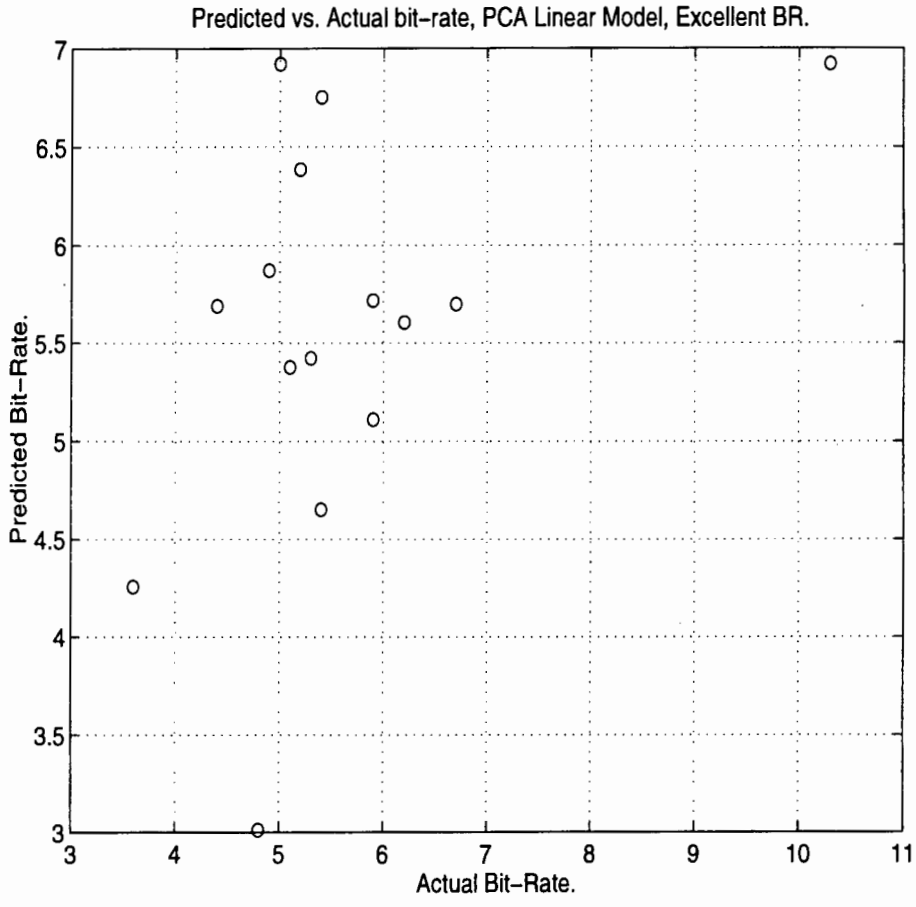


Figure 15.5: Predicted versus Actual Data, Excellent Quality BR.

Chapter 16

Conclusion to Part II Comparison of the Linear and Quadratic models

Considering the results in Chapters 14 and 15 the conclusions in this chapter are drawn and the weaknesses of the model building approach examined. The absolute value prediction error of the validated quadratic model is 40% for excellent image quality bit-rate and 37% for good image quality bit-rate. The validated linear model has absolute value prediction error of 108% for excellent image quality bit-rate and 30% for good image quality bit-rate.

It is then reasonable to suggest using the linear model to predict the bit-rates for the good image quality and the quadratic model for the excellent quality bit-rates. The quadratic model is overfitted since the model performance after cross-validation decreases significantly.

The Jackknifed linear model after PCA performs particularly well with a peak prediction error of 39% and 50% for the excellent and good image quality bit-rates. The expected average error predicted by the Jackknife is 15.29% and 15.4% per observation for excellent and good image quality bit-rates.

16.1 Weaknesses of this approach

The bit-rate is to some extent overestimated since for fixed bit-rate encoders the bit-rate chosen must be greater than the largest bit-rate peaks. The sampling of the data is simple systematic with the problem of estimating the variance of the population (though certain approximations exist). The number of bit-rate samples are too few and this presents problems in properly validating the models. It is possible that some other set of features better describe the relationship between bit-rate and the input sequence. Note that a bias is introduced into the model by measuring the objective variables on greyscale instead of colour data.

Appendix A

An overview of the Hadamard Transform, Hashing and Sobel Filtering

The Hadamard transform is used by several motion compensation algorithms because of its speed. The Projection Search (defined in 5.3) employs Hashing, and Sobel filtering is used in modelling the final bit-rate of an MPEG-2 sequence in terms of certain objective variables.

A.1 The Ordered Hadamard Transform

By definition [26]:

$$\begin{aligned} H(u,v) &= \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)(-1)^{\sum_{i=0}^{n}(b_i(x)p_i(u)+b_i(y)p_i(v))} \\ N &= 2^n \end{aligned}$$

The summation in the exponent of (-1) is modulo 2. $b_i(k)$ is the i^{th} bit of k in the binary representation of k (e.g., $b_2(6) = 1 \ \ldots \ 6_{10} = 110_2$.

$p_i(k)$ is defined as:

$$\begin{aligned} p_0(k) &= b_{n-1}(k) \\ p_1(k) &= b_{n-1}(k) + b_{n-2}(k) \\ p_2(k) &= b_{n-2}(k) + b_{n-3}(k) \end{aligned}$$

$$\vdots$$

$$p_n(k) = b_1(k) + b_0(k)$$

A.2 Hashing

Hashing [54] attempts to solve two problems:

1. Memory Optimisation: The range of data values is greater than can be contained in an array of convenient size.

2. Access Time Optimisation: One wants to be able to place and find an entry within a fixed number of operations (fixed time).

The hash table is an array of fixed size and each cell is accessed via a hash function. The hash function is usually $f(\text{data})mod(\text{size of hash table})$. The size of the hash table is predetermined as some convenient size, (the larger the better) and is usually a prime number. The hash function is designed to evenly distribute the entries over the table.

In open hashing each cell maintains a list of the entries contained in it. Each cell could hold more than one entry.

A.3 Sobel Filtering

The Sobel filter [26] is a standard filter as follows:

$$\begin{aligned} \nabla f &= \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} \\ |\nabla f| &= (\frac{\partial f}{\partial x}^2 + \frac{\partial f}{\partial y}^2)^{\frac{1}{2}} \\ \text{or } |\nabla f| &= |\frac{\partial f}{\partial x}| + |\frac{\partial f}{\partial y}| \end{aligned}$$

For approximating the gradient a number of methods are possible such as the Roberts, Prewitt and Sobel operators. The gradient is a sum of the absolute values of square roots of the convolution of the following masks on the input image (Tables A.1 and A.2).

-1	-2	-1
0	0	0
1	2	1

Table A.1: Y derivative Sobel Mask.

-1	0	1
-2	0	2
-1	0	1

Table A.2: X derivative Sobel Mask.

Note that during the convolution process the derivatives are smoothed which compensates for noise in the high frequency components.

Appendix B

Performance of motion search algorithms - Raw data

A good algorithm minimises the DFD, this implies the PSNR is maximised and the mean of the absolute value of difference image is minimised. The pixel, motion vector and cumulative entropy, and the mean of the motion vectors is minimised. The variance of the motion vectors is an indication of the non-uniformity of the motion. The variance of the absolute value of the difference image is an indication of the frequency content excluding the DC coefficient, ideally this should be minimised.

Algorithm	Difference block			
			Absolute	
	Mean	Variance	Mean	Variance
2Dlog Search	-0.61	686.0	12.9	518.7
Block Based Gradient Descent Search	-0.56	731.9	13.5	548.8
Conjugate Directional Search	-0.56	740.1	13.7	553.6
Conjugate Search	-0.61	714.3	13.4	534.8
Cross Pattern Search	-0.54	607.4	12.1	460.5
Cross Search	-0.66	630.2	12.3	478.2
Dynamic SW Adjustment	-0.60	695.4	13.1	524.5
Energy	-0.37	43.5	4.1	26.7
Four Step Search	-0.62	652.5	12.5	496.1
Improved TSS	-0.61	409.2	9.3	322.3
Modified Conjugate Search	-0.61	632.5	12.3	481.8
Modified Feature Search	-0.19	79.4	5.1	53.0
Modified Monotony	-0.02	43.2	4.2	25.5
Modified SEA	-0.05	32.6	3.7	18.8
Modified SEA v2	-0.04	41.9	4.2	24.3
Modified TSS	-0.62	623.0	12.2	475.4
Monotony	-0.03	42.7	4.2	25.2
New DMD	-0.75	580.8	11.8	441.3
One Dimensional Full Search	-0.57	694.6	13.5	511.9
PBIL	-0.21	81.1	5.2	54.2
Parallel Hierarchical One D Search	-0.24	1015.5	17.2	718.3
Plus Pattern Search	-0.57	619.2	12.3	468.6
Prefiltered Sampled Points 2	0.32	70.6	5.2	43.7
Prefiltered Sampled Points 3	-0.07	38.9	4.0	22.7
Prefiltered Sampled Points 4	0.25	46.0	4.3	27.2
Prefiltered Sampled Points 5	-0.10	177.8	7.6	120.7
Projection Search	-0.16	111.6	5.9	76.8
Sampled Pts 2	-0.06	35.5	3.9	20.6
Sampled Pts 3	-0.07	38.0	4.0	22.1
Sampled Pts 4	-0.07	39.5	4.0	23.2
Sampled Pts 5	-0.08	41.0	4.1	24.2
Subblock Full Search	-0.58	559.8	11.2	434.8
TSS	-0.63	619.0	12.1	472.9
Tile search	-0.07	48.5	4.8	25.6
Variance	-0.04	87.5	5.5	57.5

Table B.1: Comparison of Algorithms for the sequence **animate1**

Algorithm	Motion Vector		PSNR	DFD
	Mean	Variance		(x 10^8)
2Dlog Search	3.185	5.022	19.77	4.209
Block Based Gradient Descent Search	2.115	5.312	19.49	4.403
Conjugate Directional Search	2.405	6.284	19.44	4.444
Conjugate Search	2.307	5.369	19.59	4.359
Cross Pattern Search	6.206	7.062	20.30	3.944
Cross Search	4.908	8.478	20.14	4.013
Dynamic SW Adjustment	3.124	3.526	19.71	4.255
Energy	171.703	18489	31.75	1.337
Four Step Search	3.771	6.379	19.98	4.071
Improved TSS	9.807	27.515	22.01	3.038
Modified Conjugate Search	5.159	8.093	20.12	3.996
Modified Feature Search	75.929	16192	29.13	1.670
Modified Monotony	198.710	21172	31.77	1.369
Modified SEA	151.607	16346	33.00	1.208
Modified SEA v2	171.398	17573	31.91	1.360
Modified TSS	4.101	8.005	20.19	3.955
Monotony	194.650	20812	31.83	1.361
New DMD	6.204	10.562	20.49	3.847
One Dimensional Full Search	2.681	16.923	19.71	4.398
PBIL	96.294	11648	29.04	1.686
Parallel Hierarchical One D Search	0.001	0.004	18.06	5.604
Plus Pattern Search	6.196	6.961	20.21	3.994
Prefiltered Sampled Points 2	194.893	29187	29.64	1.689
Prefiltered Sampled Points 3	154.772	16534	32.24	1.306
Prefiltered Sampled Points 4	176.844	20502	31.50	1.412
Prefiltered Sampled Points 5	36.813	8302	25.63	2.457
Projection Search	171.746	17875	27.65	1.919
Sampled Pts 2	153.214	16396	32.63	1.255
Sampled Pts 3	154.951	16610	32.34	1.295
Sampled Pts 4	156.389	16583	32.17	1.310
Sampled Pts 5	156.433	16672	32.00	1.334
Subblock Full Search	6.789	8.898	20.65	3.640
TSS	4.957	6.510	20.21	3.935
Tile search	27.231	3859	31.27	1.554
Variance	195.015	19462	28.71	1.780

Table B.2: Comparison of Algorithms for the sequence **animate1**

Algorithm	Entropy		
	Pixel	Motion Vector	Cumulative
2Dlog Search	5.72	3.31	5.68
Block Based Gradient Descent Search	5.79	2.81	5.74
Conjugate Directional Search	5.81	2.80	5.76
Conjugate Search	5.79	2.77	5.74
Cross Pattern Search	5.68	3.96	5.65
Cross Search	5.68	3.80	5.64
Dynamic SW Adjustment	5.74	3.25	5.70
Energy	4.46	9.09	4.72
Four Step Search	5.68	3.54	5.64
Improved TSS	5.31	4.78	5.31
Modified Conjugate Search	5.66	3.63	5.62
Modified Feature Search	4.74	6.13	4.84
Modified Monotony	4.50	9.26	4.78
Modified SEA	4.33	8.89	4.59
Modified SEA v2	4.50	9.09	4.76
Modified TSS	5.65	3.59	5.60
Monotony	4.49	9.24	4.77
New DMD	5.64	4.28	5.61
One Dimensional Full Search	5.86	1.92	5.81
PBIL	4.76	6.08	4.88
Parallel Hierarchical One D Search	6.17	0.00	6.10
Plus Pattern Search	5.70	3.96	5.67
Prefiltered Sampled Points 2	4.79	8.22	5.00
Prefiltered Sampled Points 3	4.44	8.91	4.69
Prefiltered Sampled Points 4	4.54	8.46	4.78
Prefiltered Sampled Points 5	0.22	0.34	0.23
Projection Search	4.92	9.12	5.15
Sampled Pts 2	4.38	8.28	4.63
Sampled Pts 3	4.43	8.92	4.68
Sampled Pts 4	4.44	8.05	4.68
Sampled Pts 5	4.47	8.92	4.72
Subblock Full Search	5.51	3.99	5.49
TSS	5.64	3.75	5.60
Tile search	4.70	5.21	4.74
Variance	4.84	9.30	5.09

Table B.3: Comparison of Algorithms for the sequence **animate1**

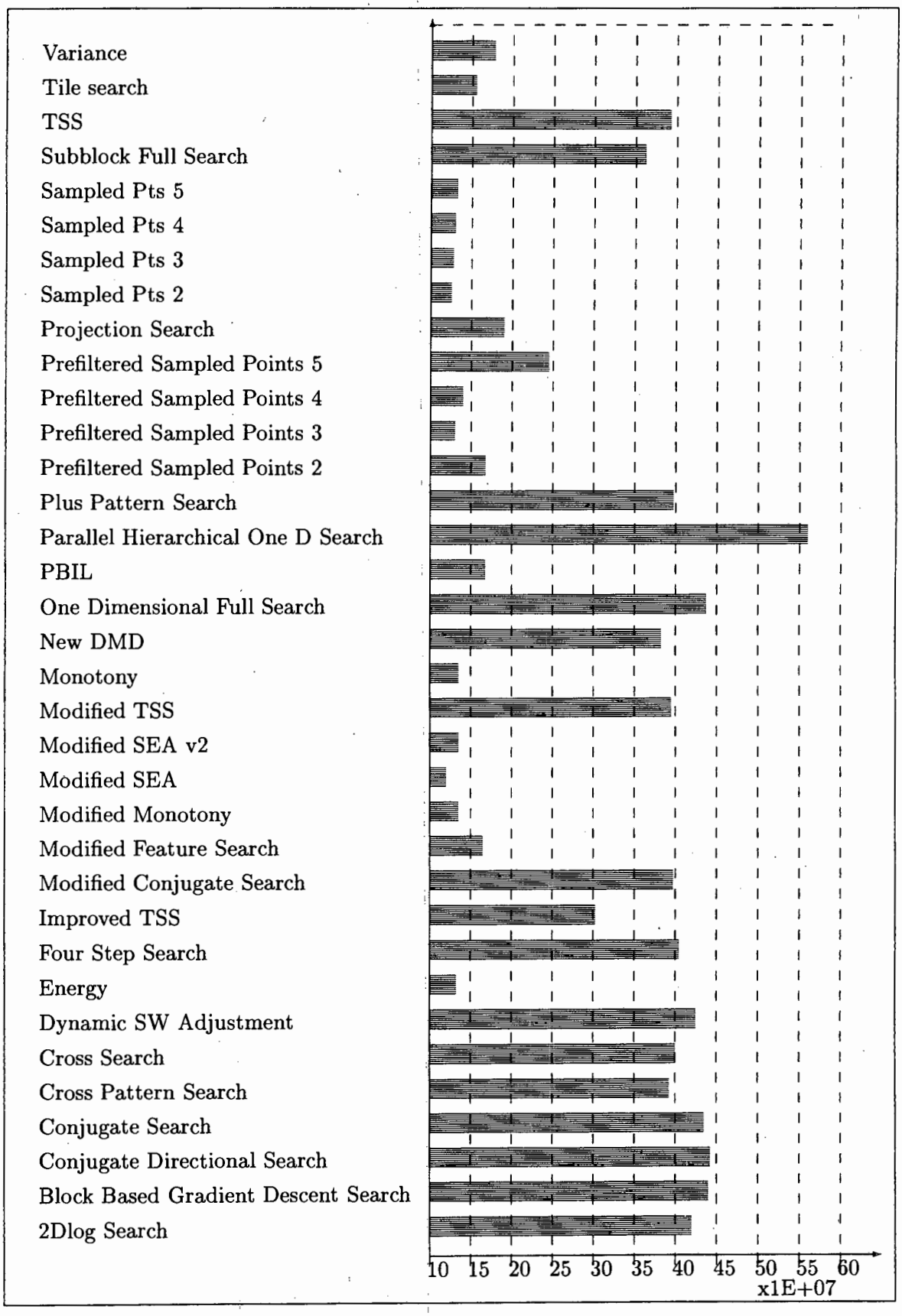


Figure B.1: **DFD** for sequence **animate1**

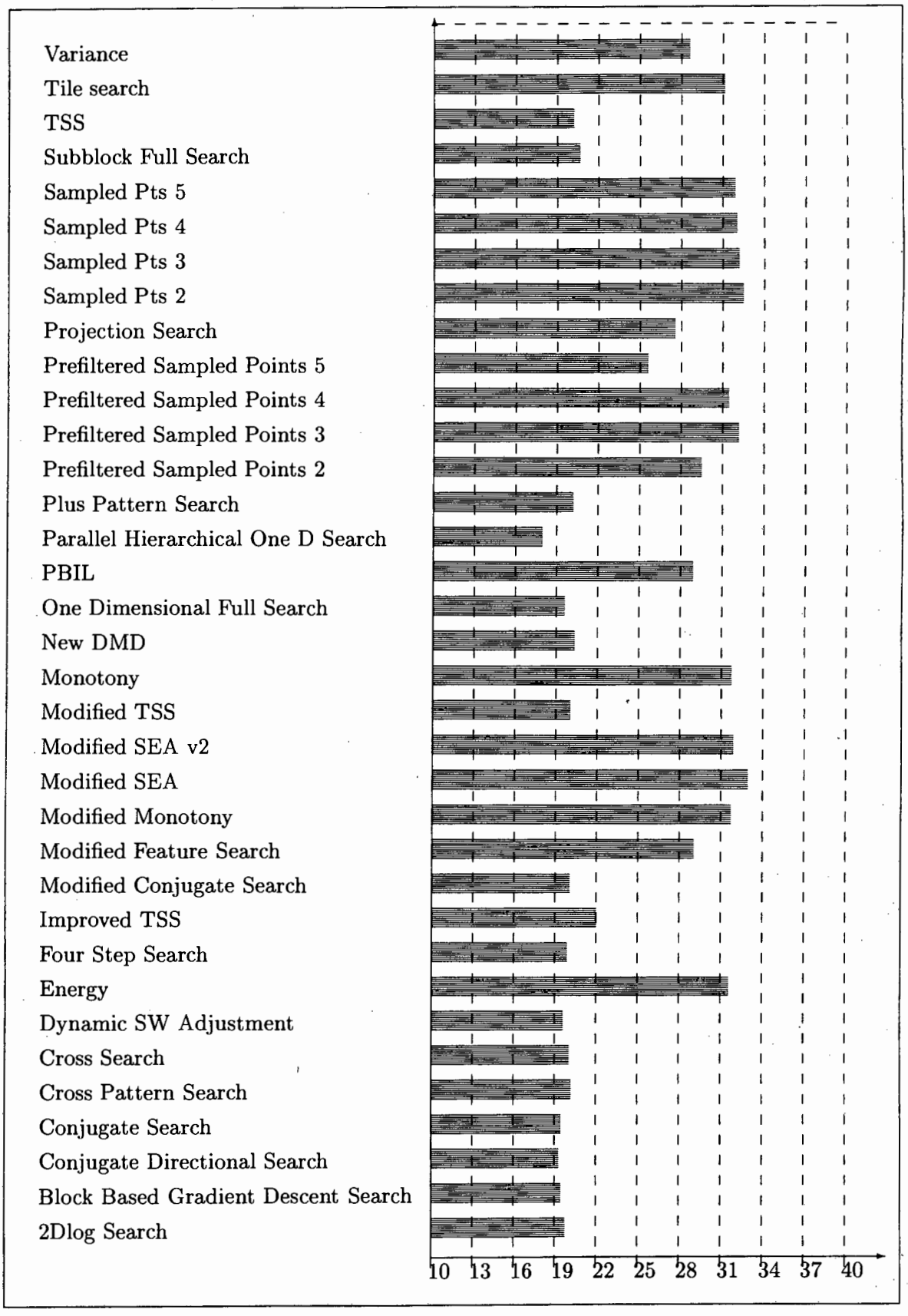


Figure B.2: **PSNR** for sequence **animate1**

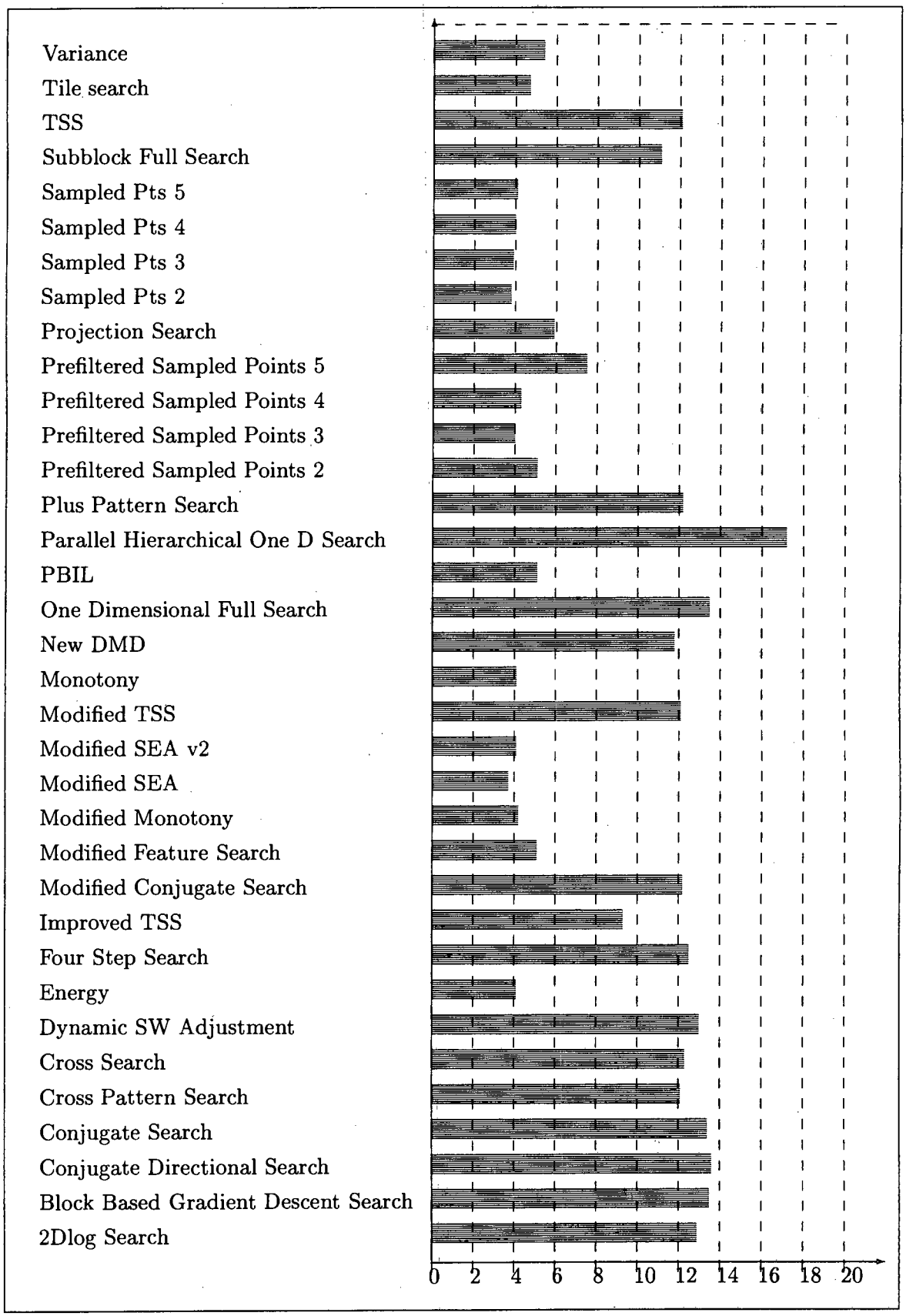


Figure B.3: **Di abs mean** for sequence **animate1**

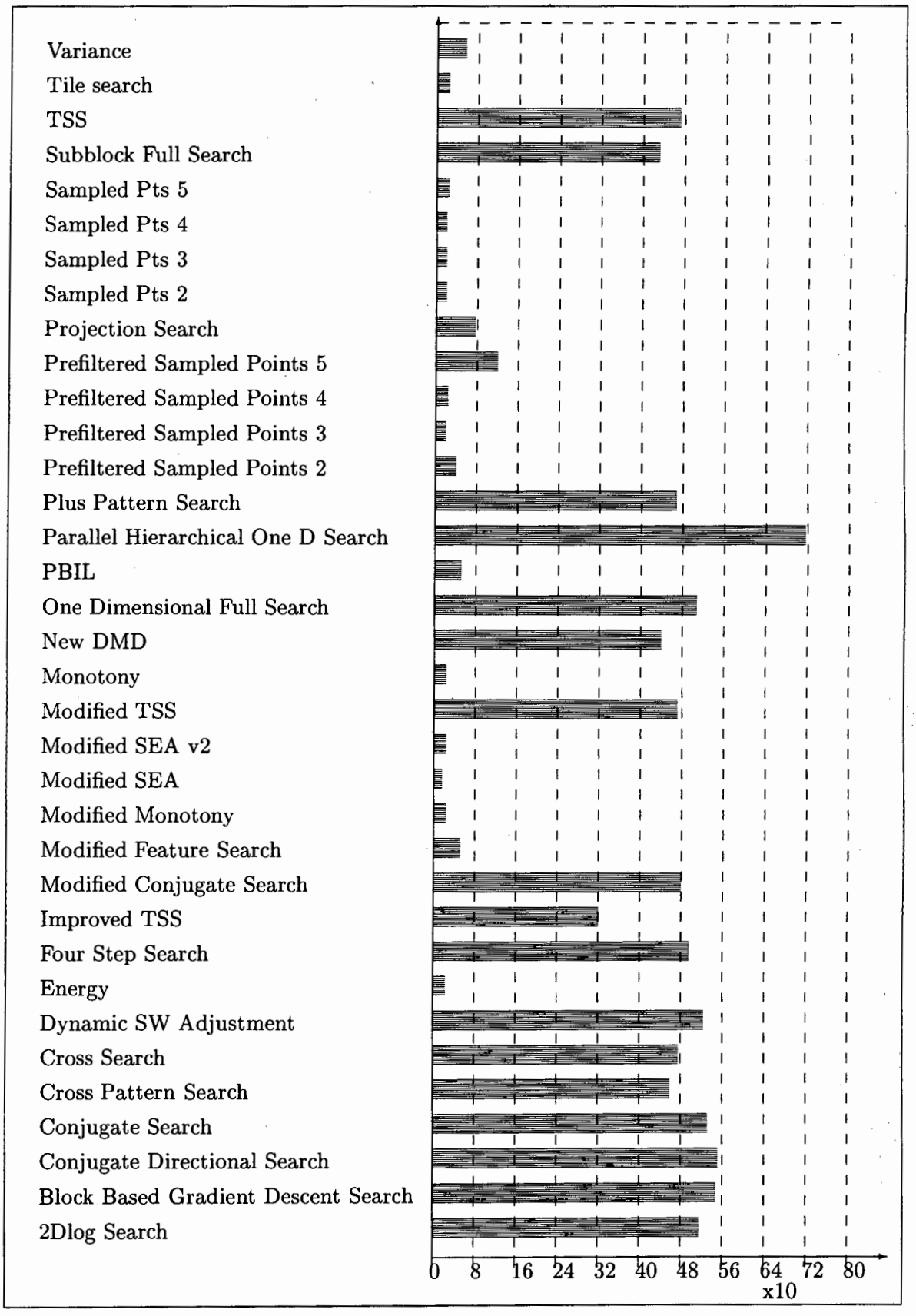


Figure B.4: **Di abs var** for sequence **animate1**

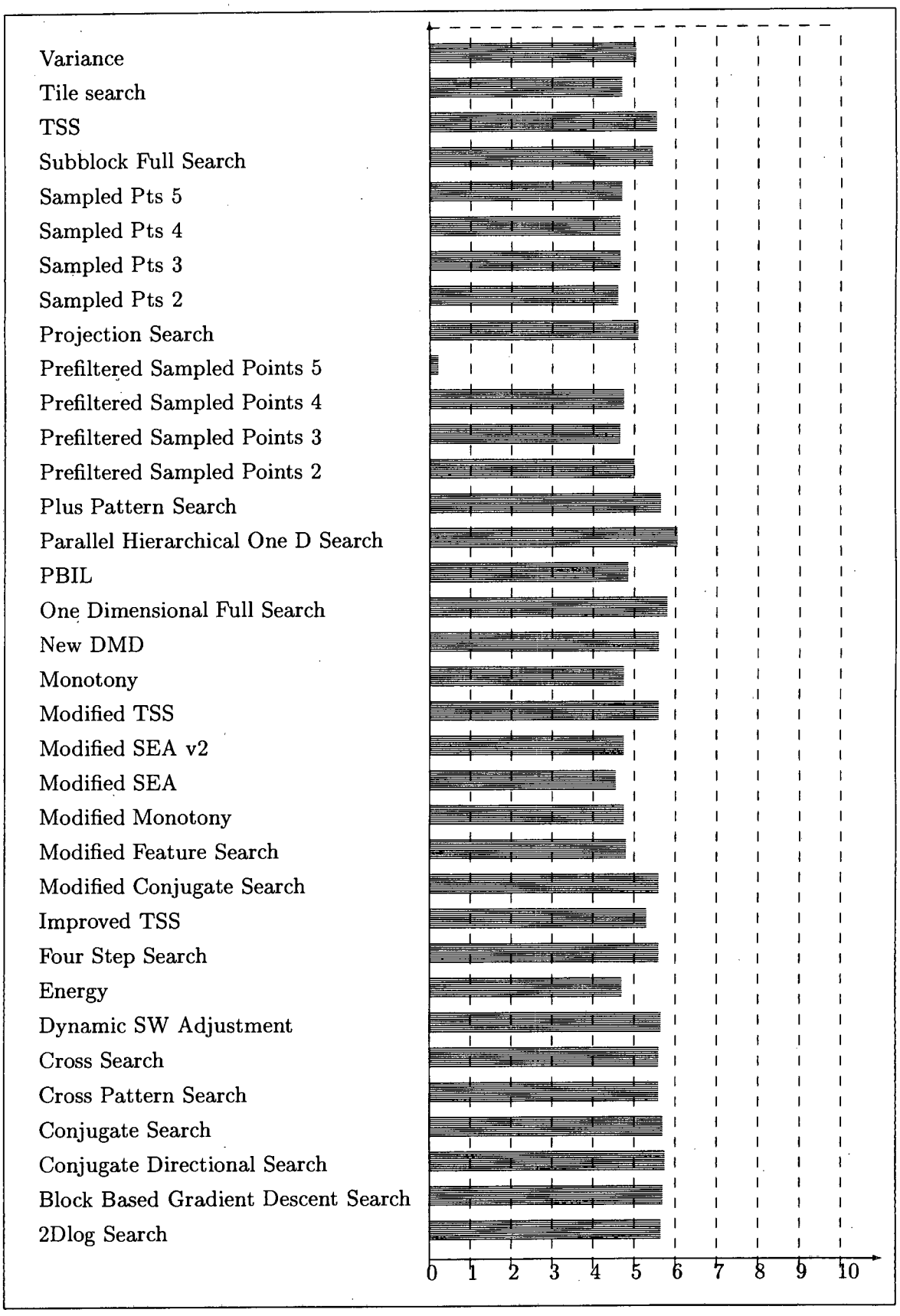


Figure B.5: **Entropy cumulative** for sequence **animate1**

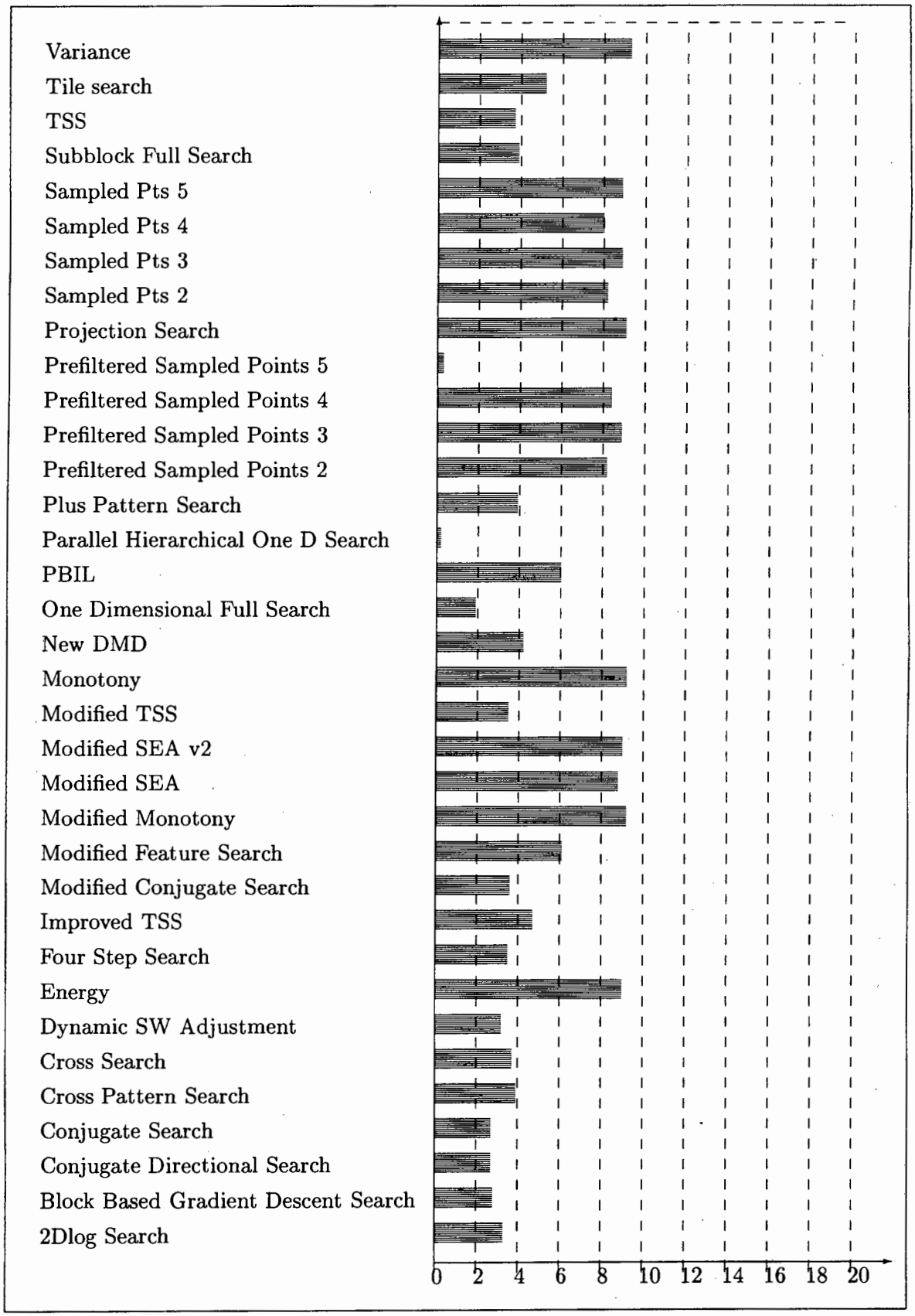


Figure B.6: **Entropy mtn vec** for sequence **animate1**

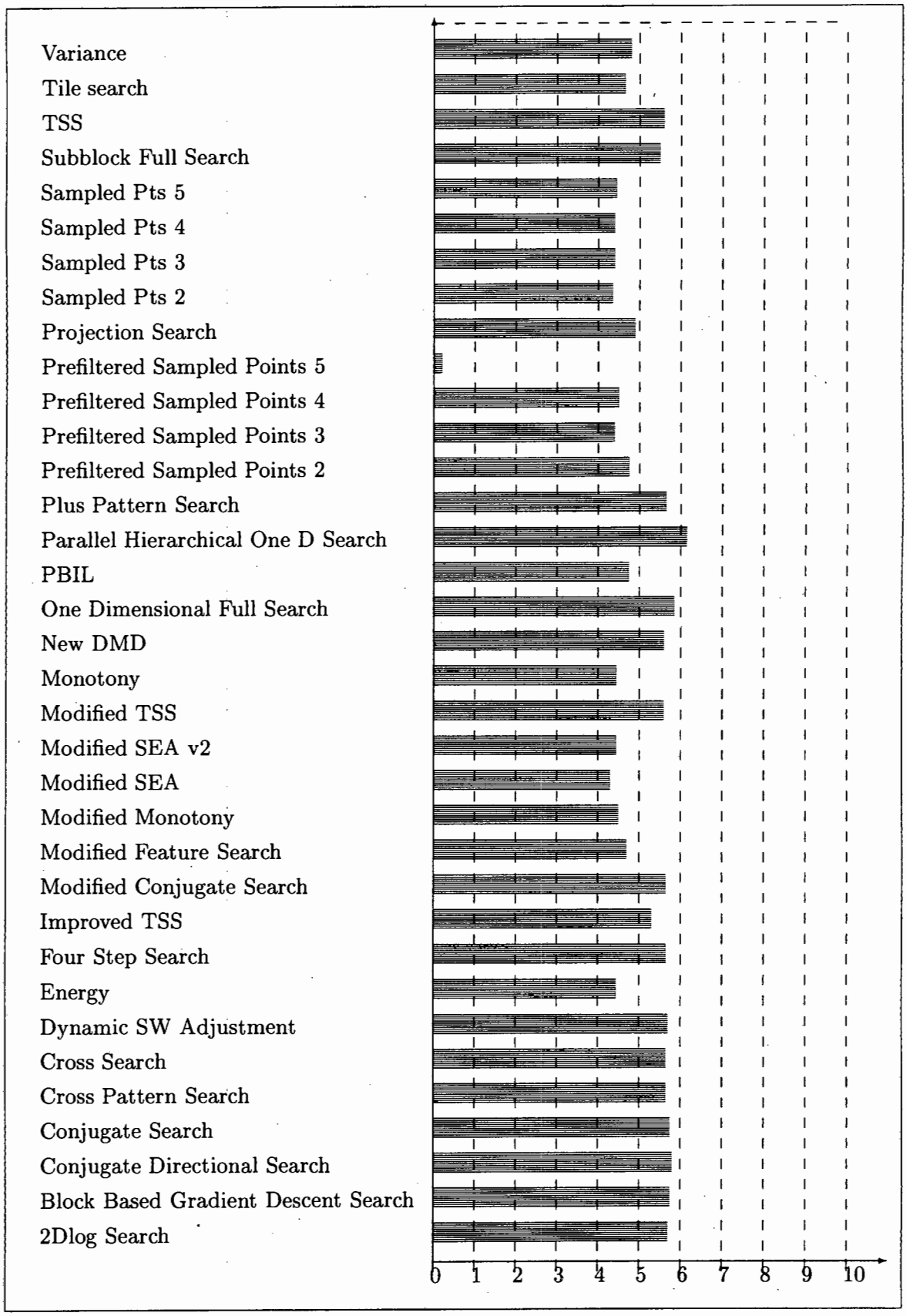


Figure B.7: **Entropy pixel** for sequence **animate1**

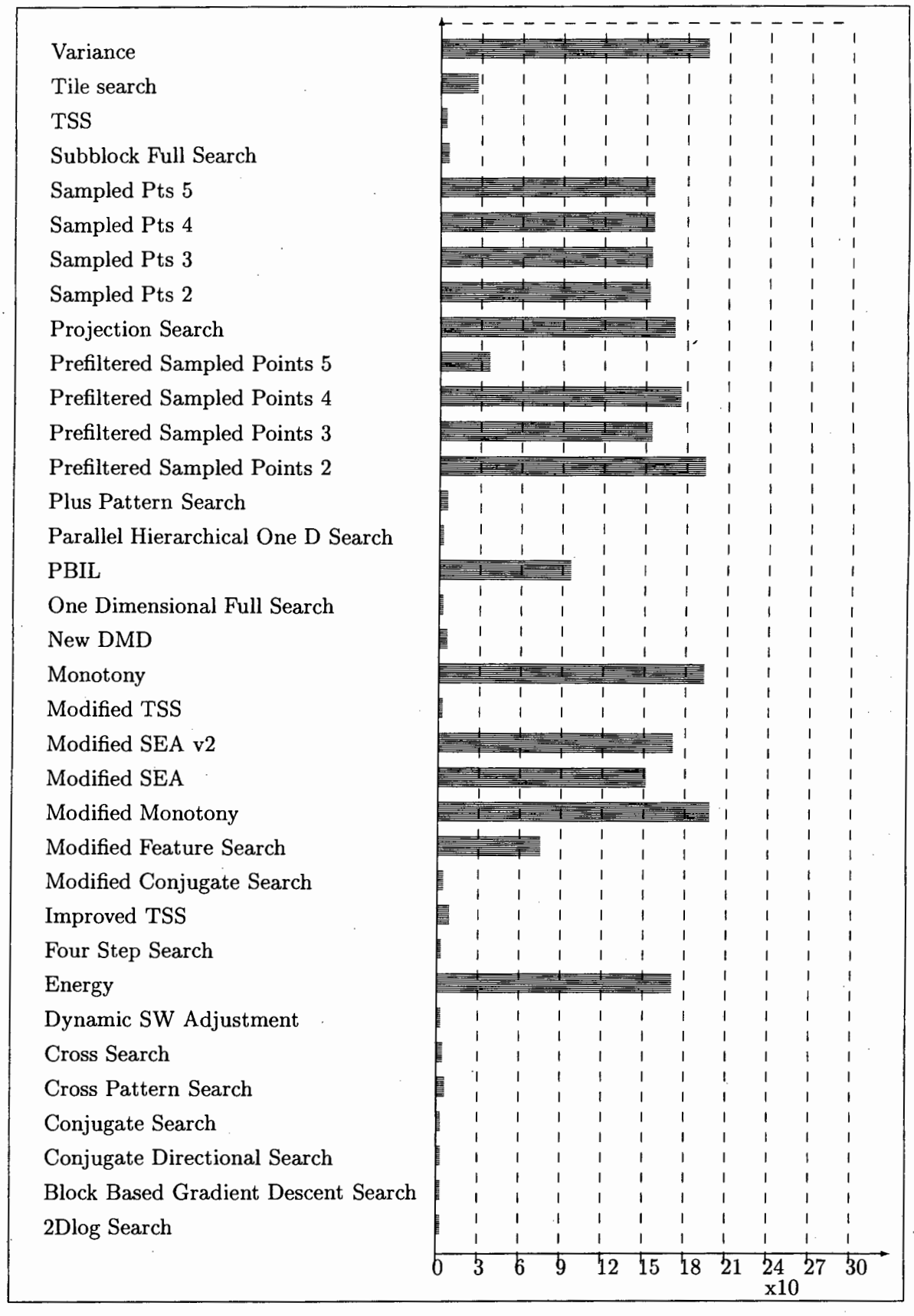


Figure B.8: **Motion Vector Means** for sequence **animate1**

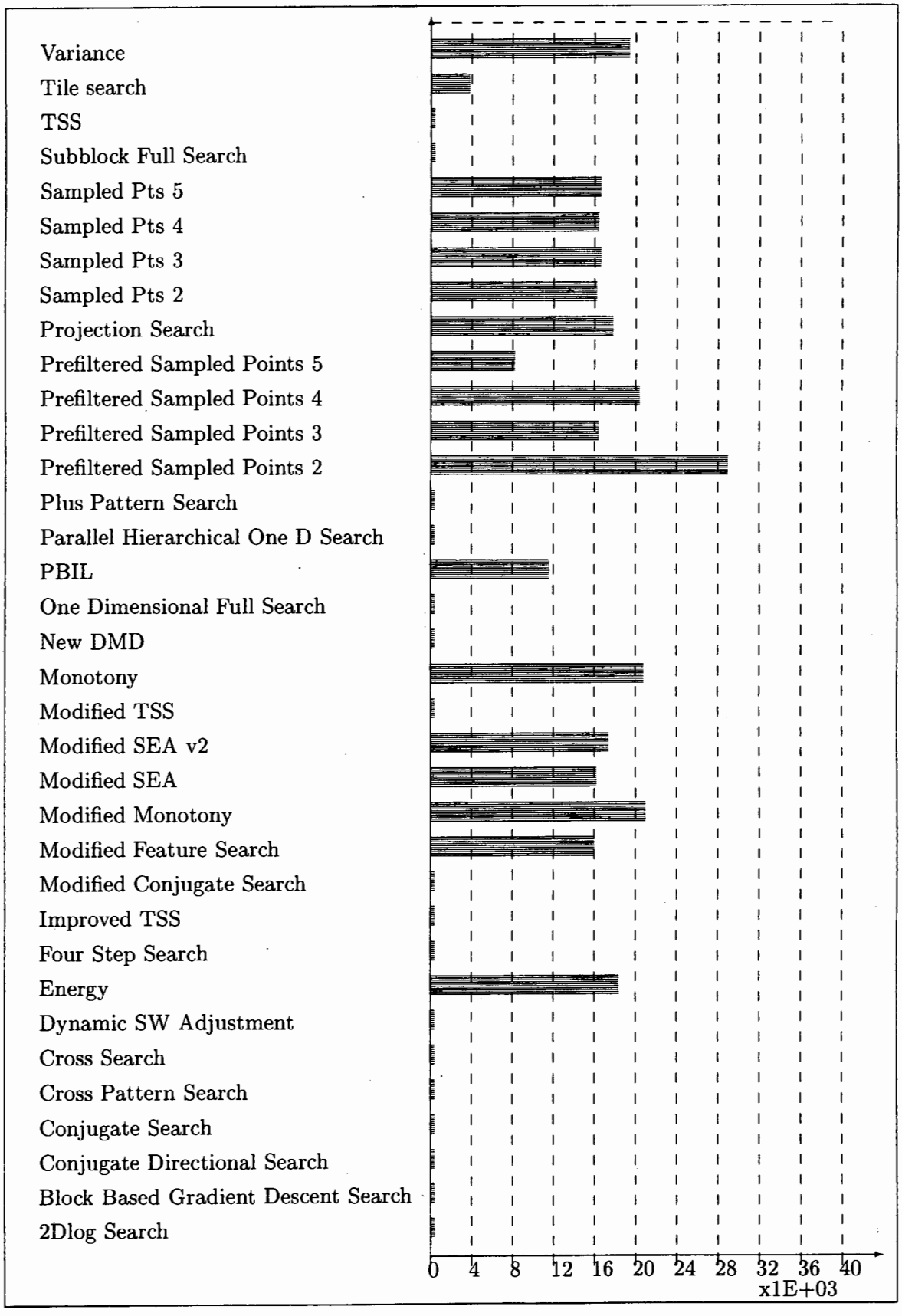


Figure B.9: **Motion Vector Var** for sequence **animate1**

Algorithm	Difference block			
			Absolute	
	Mean	Variance	Mean	Variance
2Dlog Search	0.07	407.2	9.4	319.2
Block Based Gradient Descent Search	0.11	441.9	9.8	345.8
Conjugate Directional Search	0.08	447.6	9.9	349.5
Conjugate Search	0.08	433.9	9.8	338.3
Cross Pattern Search	0.14	378.0	9.3	292.2
Cross Search	0.08	384.0	9.2	299.2
Dynamic SW Adjustment	0.06	423.5	9.6	331.2
Energy	-0.33	114.3	5.7	81.9
Four Step Search	0.09	392.0	9.2	307.3
Improved TSS	0.23	263.5	7.6	205.1
Modified Conjugate Search	0.11	384.1	9.2	300.1
Modified Feature Search	0.06	141.0	6.3	100.9
Modified Monotony	-0.01	117.3	5.8	83.6
Modified SEA	0.04	97.0	5.1	71.2
Modified SEA v2	-0.02	138.3	6.4	96.8
Modified TSS	0.10	379.5	9.1	297.4
Monotony	-0.01	116.6	5.8	83.1
New DMD	0.11	367.2	9.3	280.6
One Dimensional Full Search	0.13	446.8	10.2	343.2
PBIL	0.17	156.4	6.5	114.0
Parallel Hierarchical One D Search	-0.02	637.9	12.2	489.0
Plus Pattern Search	0.17	401.2	9.7	307.9
Prefiltered Sampled Points 2	0.71	119.7	6.0	84.3
Prefiltered Sampled Points 3	0.01	104.7	5.3	76.3
Prefiltered Sampled Points 4	0.32	110.0	5.6	79.0
Prefiltered Sampled Points 5	0.03	109.6	5.5	79.7
Projection Search	0.09	192.6	7.8	132.0
Sampled Pts 2	0.05	102.5	5.3	74.8
Sampled Pts 3	0.01	104.0	5.3	75.9
Sampled Pts 4	0.07	106.5	5.4	77.5
Sampled Pts 5	0.04	107.4	5.4	78.2
Subblock Full Search	0.14	334.1	8.5	262.6
TSS	0.10	373.9	9.0	292.8
Tile search	0.08	108.0	5.8	74.9
Variance	0.04	223.9	8.2	156.4

Table B.4: Comparison of Algorithms for the sequence **animate2**

Algorithm	Motion Vector		PSNR	DFD
	Mean	Variance		(x 10^8)
2Dlog Search	2.459	5.742	22.03	3.050
Block Based Gradient Descent Search	1.600	4.778	21.68	3.187
Conjugate Directional Search	1.895	5.717	21.62	3.220
Conjugate Search	1.836	5.072	21.76	3.178
Cross Pattern Search	4.603	10.345	22.36	3.011
Cross Search	3.184	9.860	22.29	2.994
Dynamic SW Adjustment	2.257	3.976	21.86	3.123
Energy	159.205	21456	27.55	1.853
Four Step Search	2.804	7.141	22.20	2.991
Improved TSS	6.554	33.910	23.92	2.484
Modified Conjugate Search	3.394	10.030	22.29	2.980
Modified Feature Search	60.337	13950	26.64	2.058
Modified Monotony	176.490	22574	27.44	1.888
Modified SEA	127.120	18932	28.26	1.652
Modified SEA v2	177.882	18390	26.72	2.095
Modified TSS	2.732	8.045	22.34	2.945
Monotony	174.920	22434	27.46	1.881
New DMD	4.488	12.303	22.48	3.025
One Dimensional Full Search	1.972	12.650	21.63	3.309
PBIL	58.601	9958	26.19	2.117
Parallel Hierarchical One D Search	0.001	0.004	20.08	3.967
Plus Pattern Search	4.909	10.221	22.10	3.140
Prefiltered Sampled Points 2	131.206	26578	27.35	1.948
Prefiltered Sampled Points 3	127.758	19405	27.93	1.733
Prefiltered Sampled Points 4	127.569	22121	27.72	1.811
Prefiltered Sampled Points 5	126.761	19629	27.73	1.777
Projection Search	166.761	19498	25.28	2.530
Sampled Pts 2	127.374	18999	28.02	1.710
Sampled Pts 3	128.251	19416	27.96	1.724
Sampled Pts 4	127.366	19383	27.86	1.751
Sampled Pts 5	127.128	19530	27.82	1.758
Subblock Full Search	4.619	13.775	22.89	2.749
TSS	3.400	8.809	22.40	2.927
Tile search	39.114	8131	27.80	1.872
Variance	198.228	19955	24.63	2.670

Table B.5: Comparison of Algorithms for the sequence **animate2**

Algorithm	Entropy		
	Pixel	Motion Vector	Cumulative
2Dlog Search	5.38	2.91	5.34
Block Based Gradient Descent Search	5.44	2.33	5.39
Conjugate Directional Search	5.46	2.38	5.41
Conjugate Search	5.45	2.37	5.40
Cross Pattern Search	5.41	3.72	5.38
Cross Search	5.38	3.25	5.34
Dynamic SW Adjustment	5.42	2.76	5.37
Energy	4.86	8.54	5.05
Four Step Search	5.36	3.11	5.32
Improved TSS	5.15	4.15	5.14
Modified Conjugate Search	5.36	3.07	5.33
Modified Feature Search	5.00	5.22	5.05
Modified Monotony	4.89	8.77	5.10
Modified SEA	4.69	7.79	4.85
Modified SEA v2	5.04	9.06	5.26
Modified TSS	5.34	3.01	5.30
Monotony	4.89	8.75	5.09
New DMD	5.43	3.83	5.40
One Dimensional Full Search	5.52	1.59	5.47
PBIL	5.03	4.44	5.07
Parallel Hierarchical One D Search	5.73	0.00	5.67
Plus Pattern Search	5.47	3.78	5.44
Prefiltered Sampled Points 2	4.94	6.48	5.07
Prefiltered Sampled Points 3	4.76	7.66	4.92
Prefiltered Sampled Points 4	4.83	7.10	4.97
Prefiltered Sampled Points 5	4.79	7.58	4.95
Projection Search	5.31	8.86	5.50
Sampled Pts 2	4.74	7.41	4.90
Sampled Pts 3	4.75	7.69	4.91
Sampled Pts 4	4.77	7.29	4.93
Sampled Pts 5	4.78	7.62	4.94
Subblock Full Search	5.26	3.59	5.23
TSS	5.34	3.28	5.30
Tile search	4.90	4.89	4.93
Variance	5.37	9.27	5.58

Table B.6: Comparison of Algorithms for the sequence **animate2**

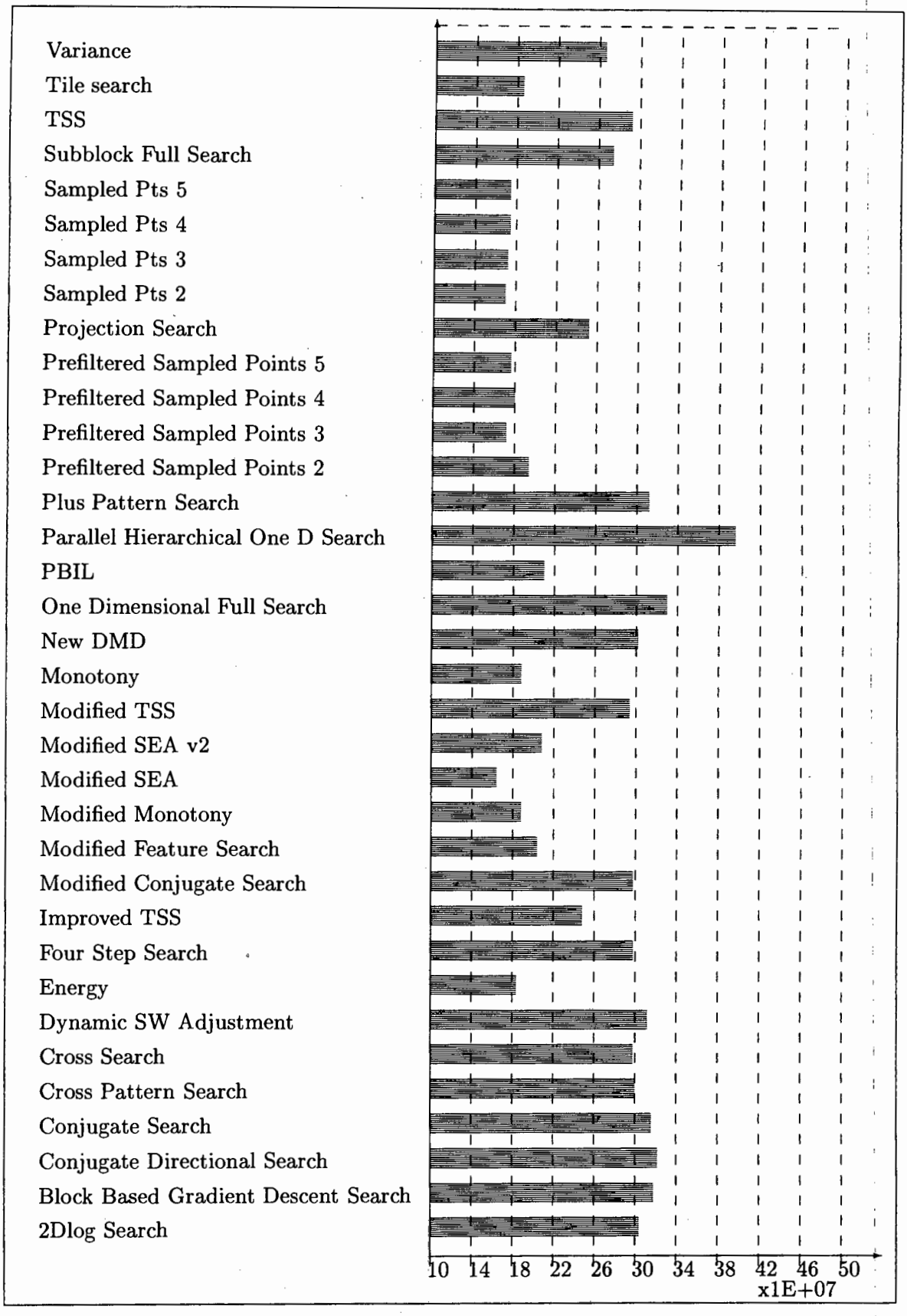


Figure B.10: **DFD** for sequence **animate2**

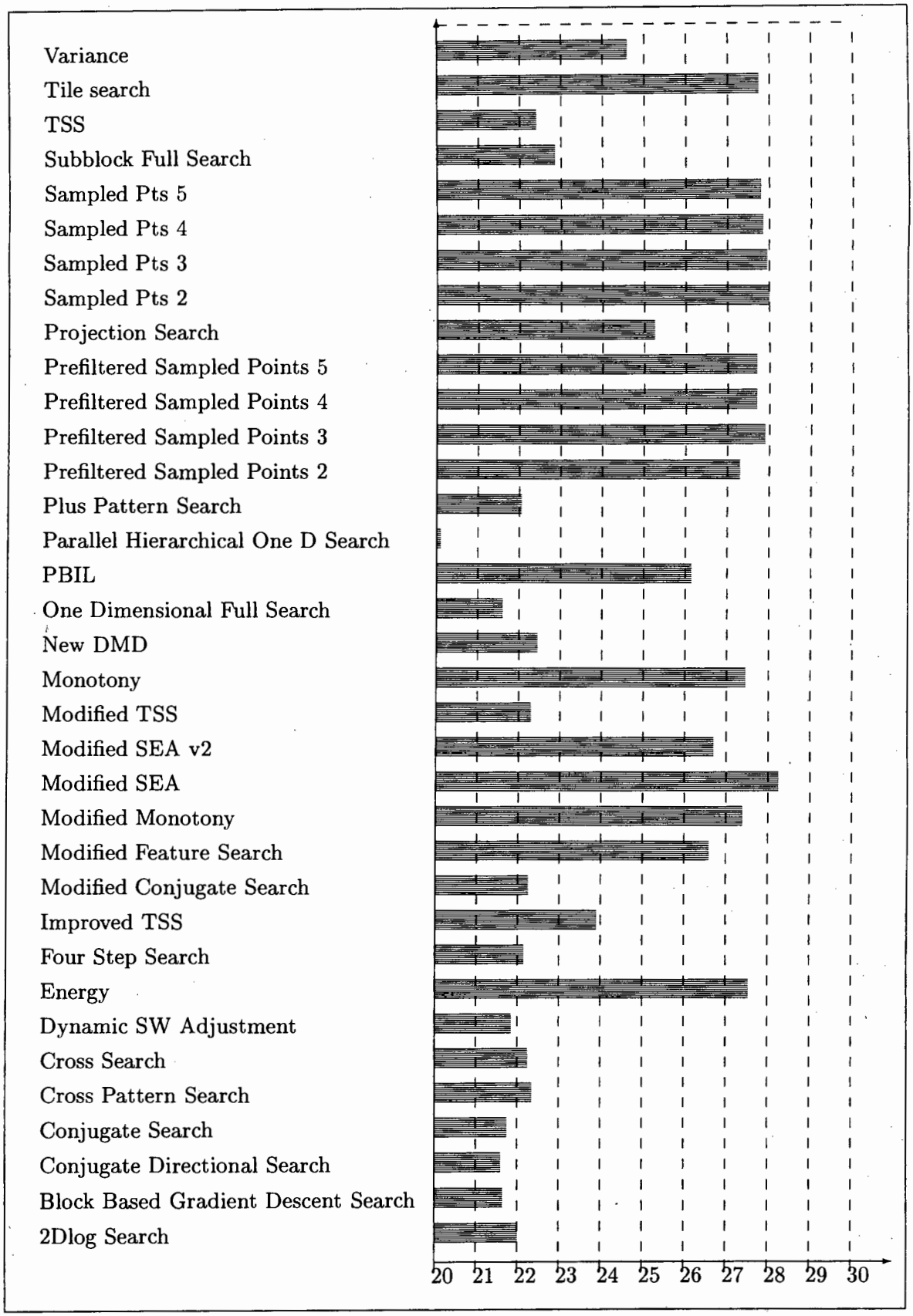


Figure B.11: **PSNR** for sequence **animate2**

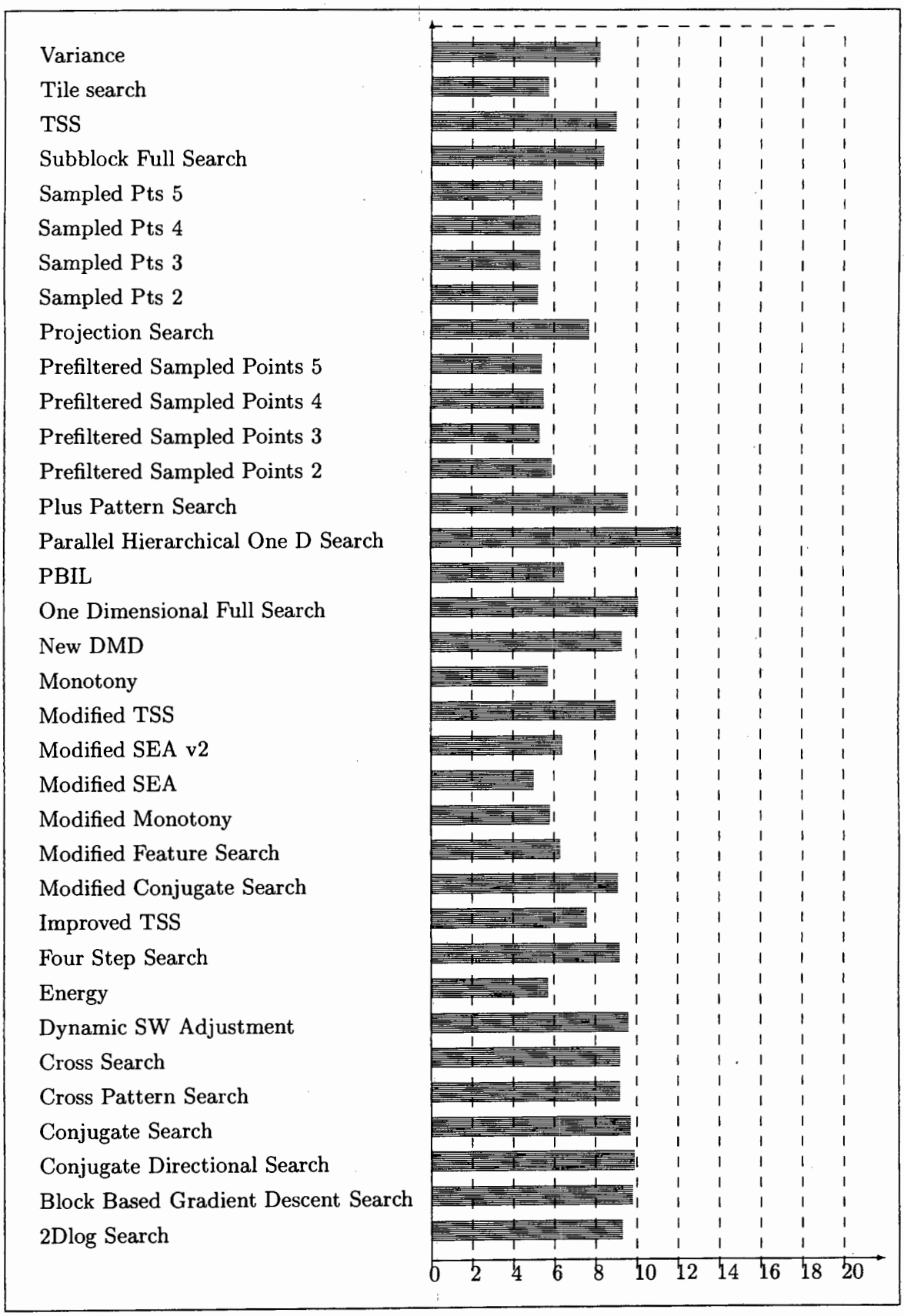


Figure B.12: **Di abs mean** for sequence **animate2**

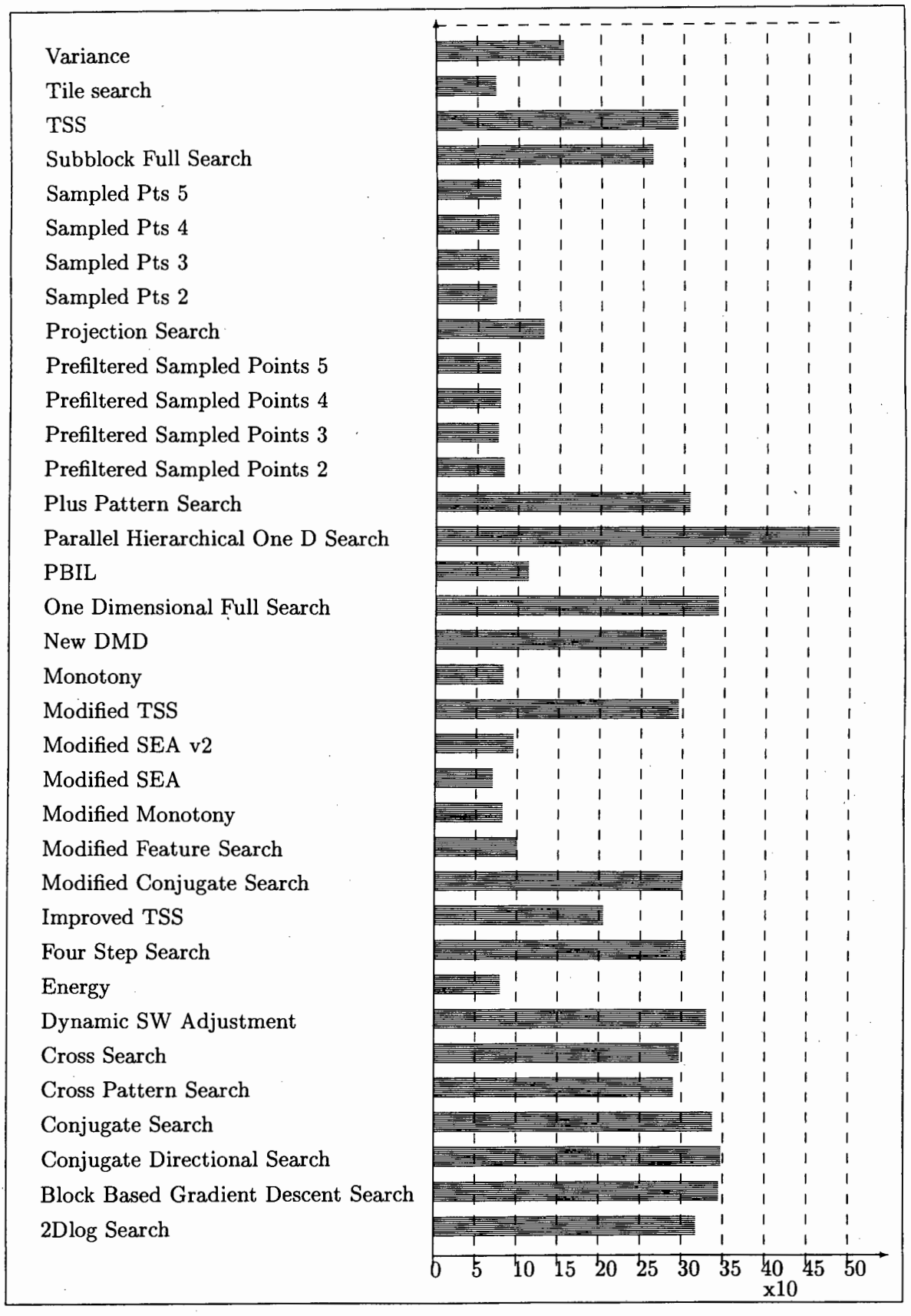


Figure B.13: **Di abs var** for sequence **animate2**

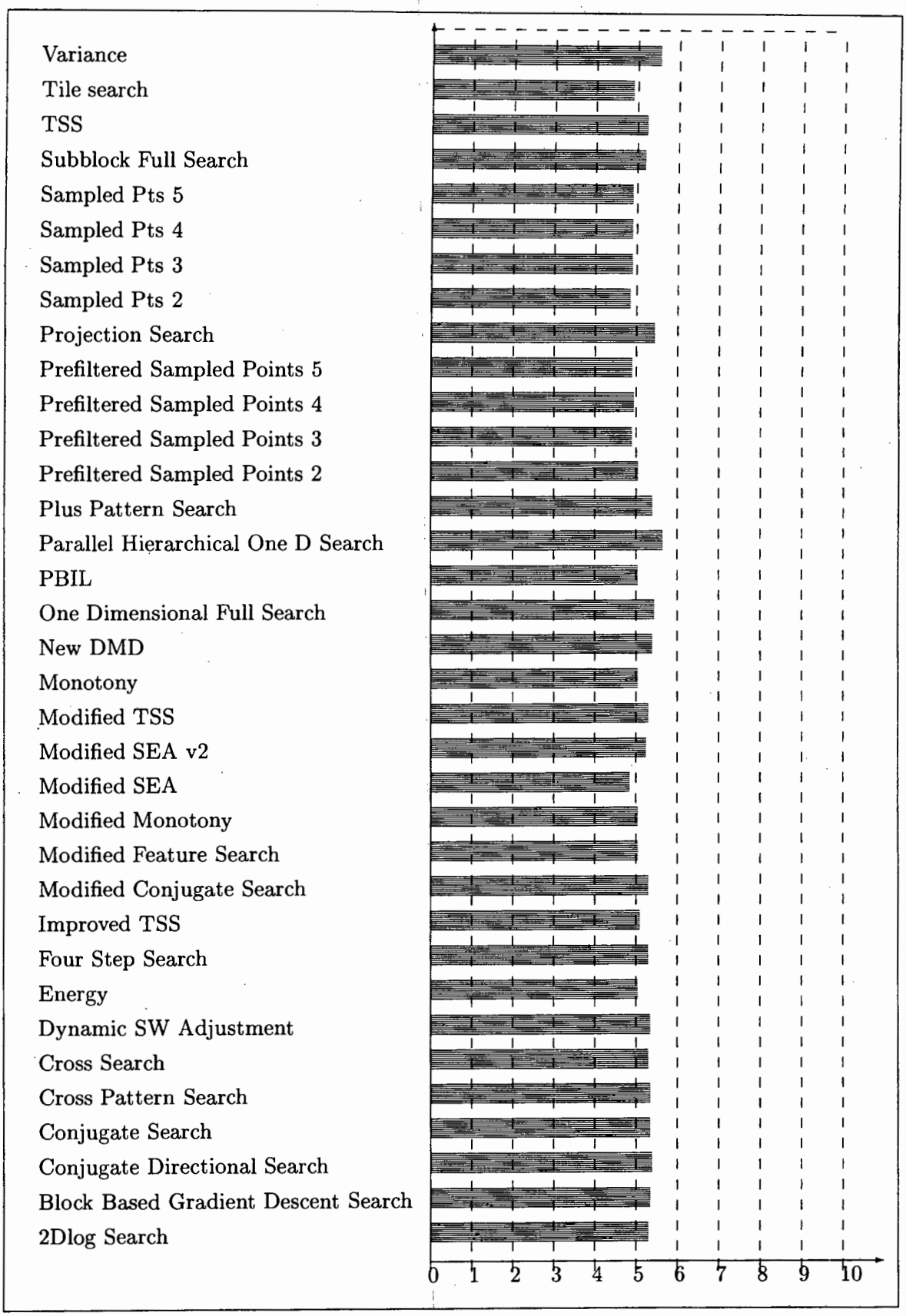


Figure B.14: **Entropy cumulative** for sequence **animate2**

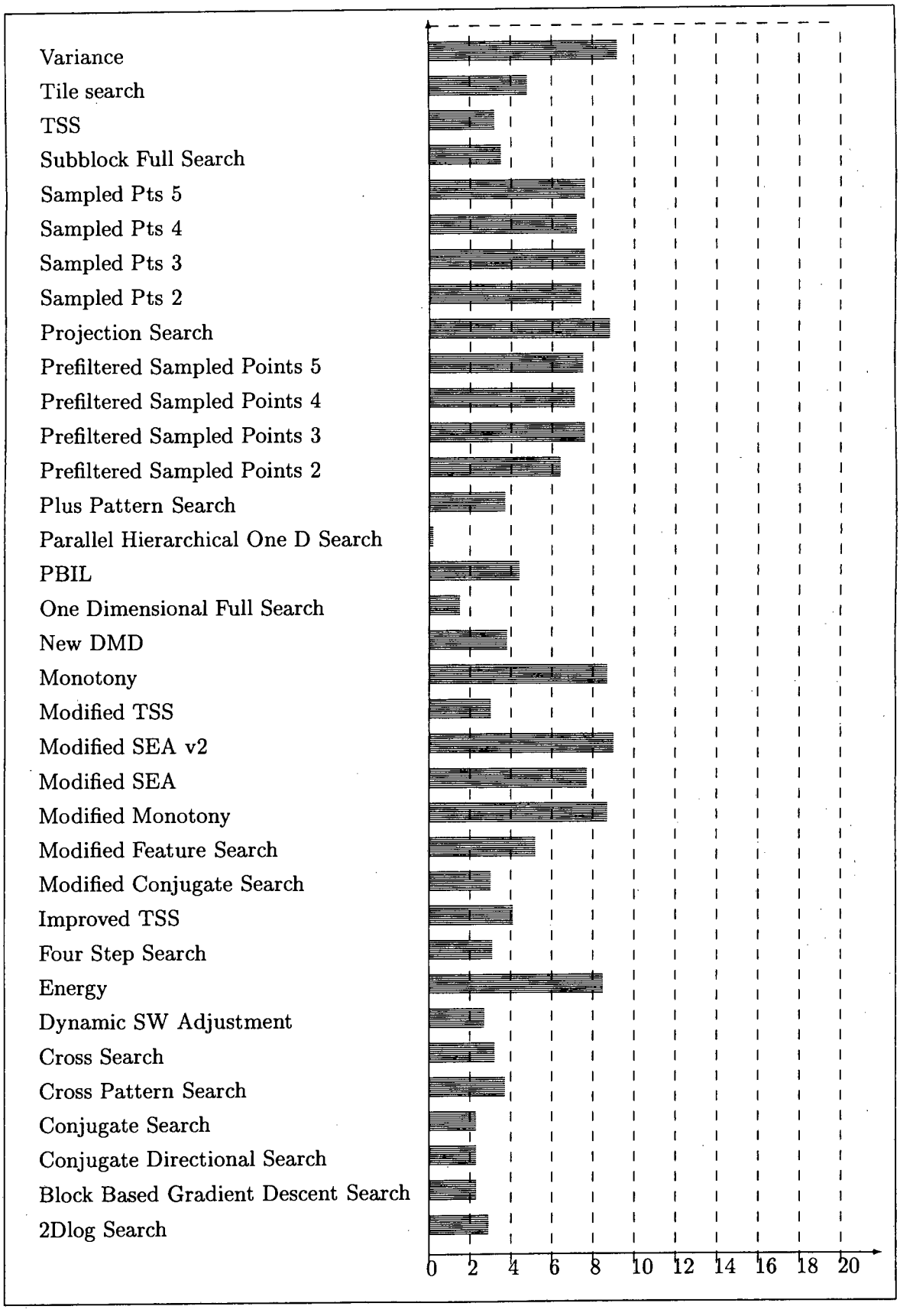


Figure B.15: **Entropy mtn vec** for sequence **animate2**

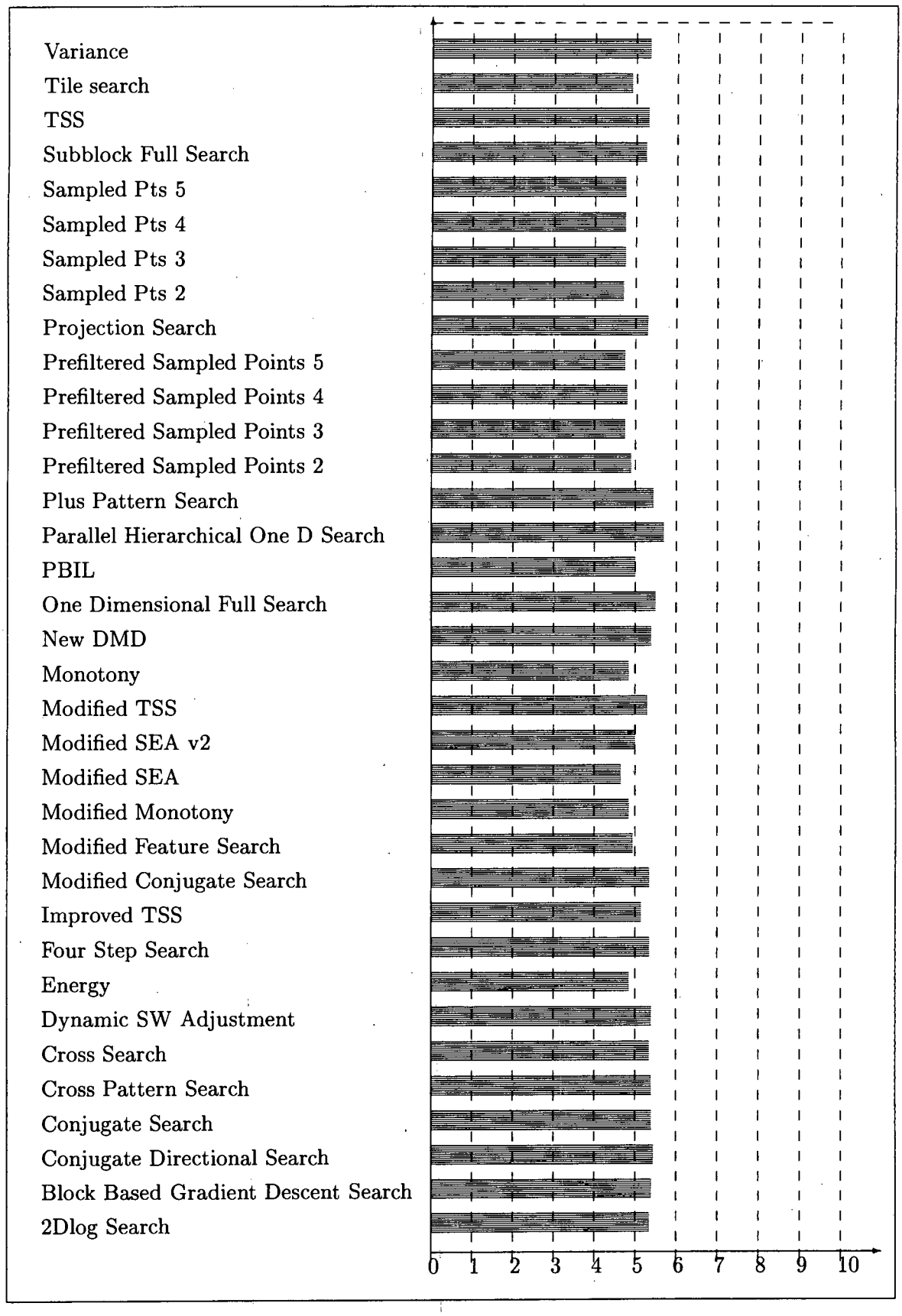


Figure B.16: **Entropy pixel** for sequence **animate2**

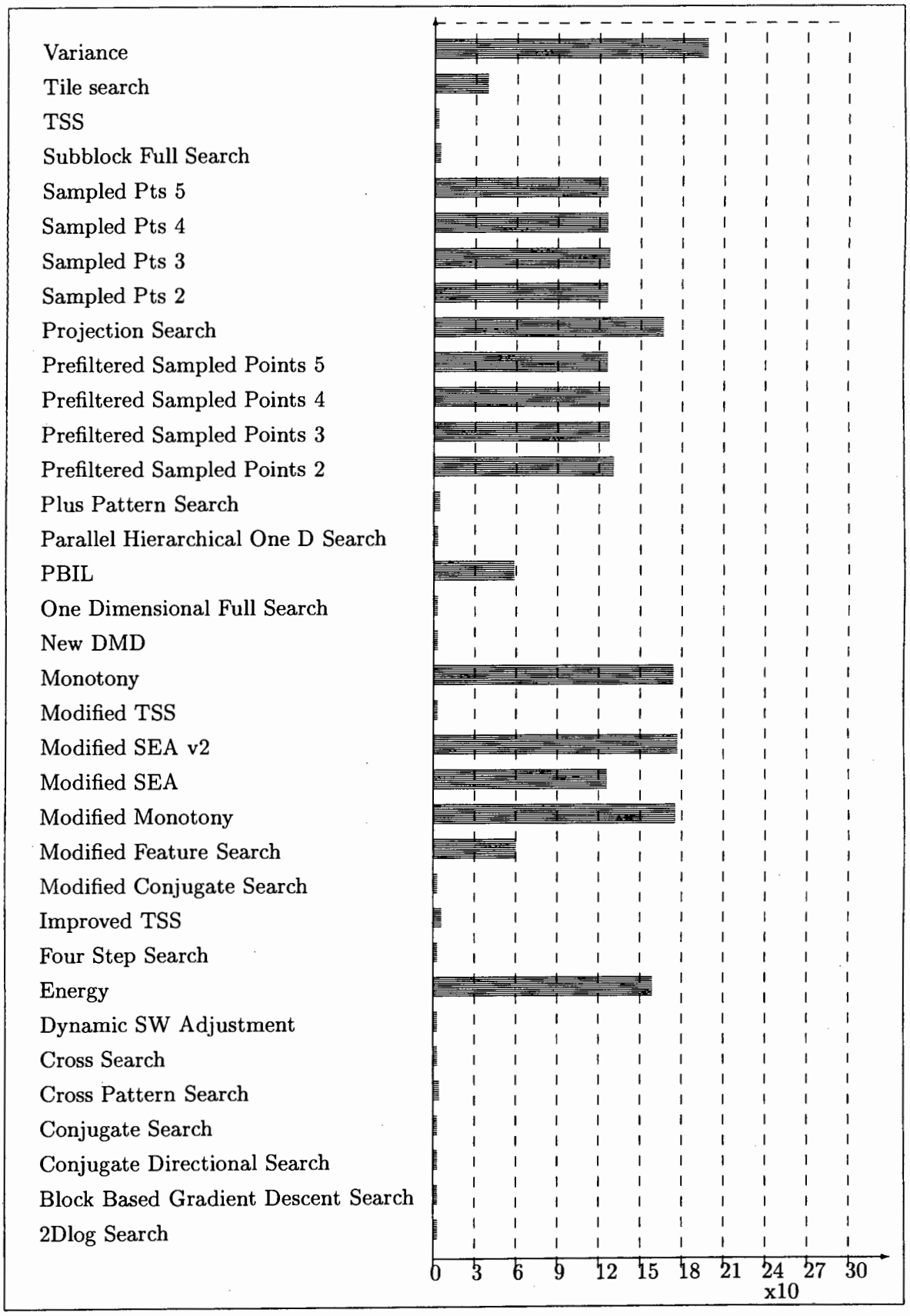


Figure B.17: **Motion Vector Means** for sequence **animate2**

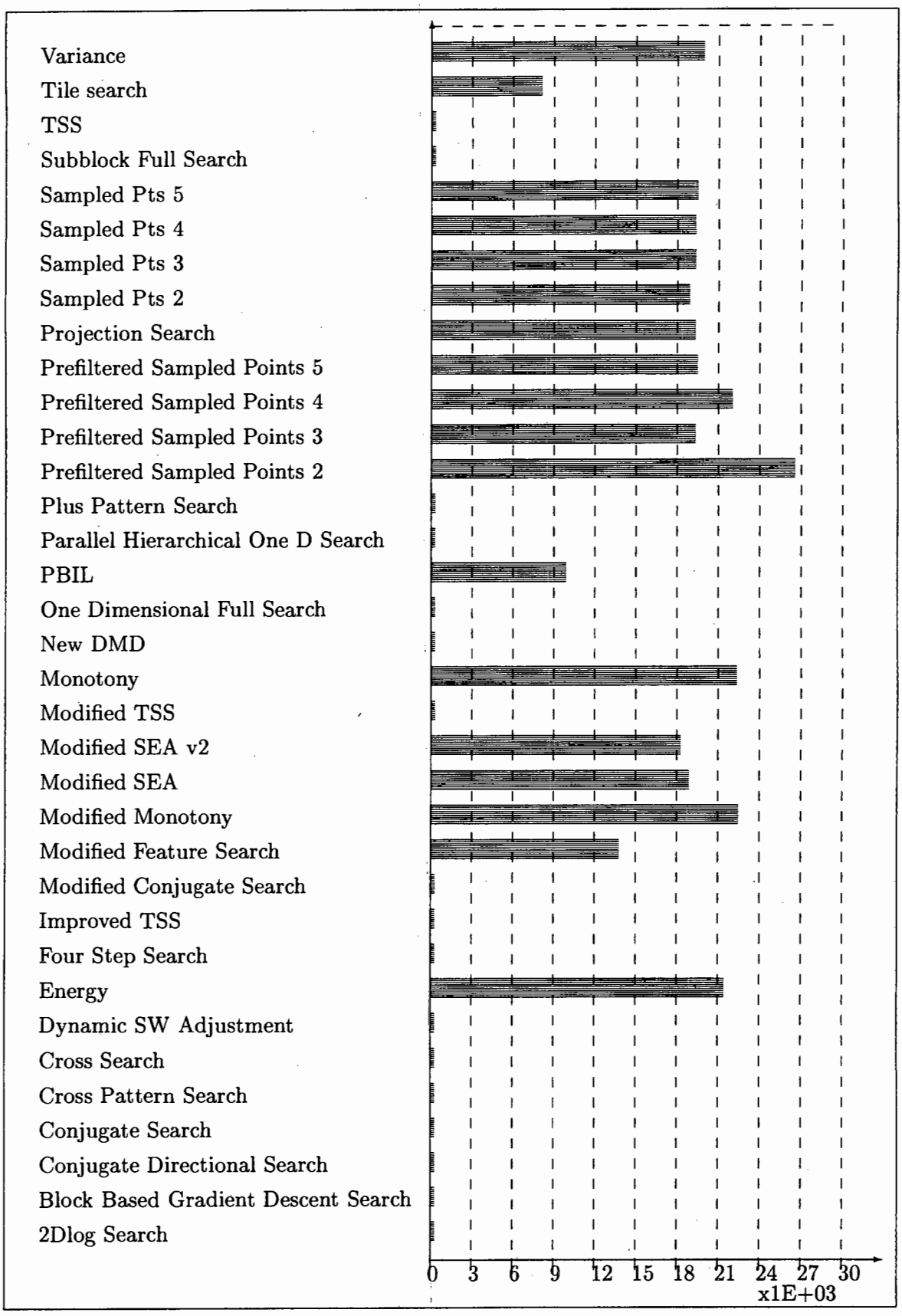


Figure B.18: **Motion Vector Var** for sequence **animate2**

Algorithm	Difference block			
			Absolute	
	Mean	Variance	Mean	Variance
2Dlog Search	0.09	431.4	12.7	270.0
Block Based Gradient Descent Search	0.05	478.0	13.7	291.4
Conjugate Directional Search	0.11	487.7	13.8	296.4
Conjugate Search	0.09	472.2	13.7	284.3
Cross Pattern Search	0.03	404.8	12.4	251.4
Cross Search	0.11	409.6	12.5	252.8
Dynamic SW Adjustment	0.08	468.2	13.4	289.3
Energy	-0.39	78.7	6.5	36.1
Four Step Search	0.09	403.5	12.3	252.2
Improved TSS	0.02	250.8	9.8	155.4
Modified Conjugate Search	0.09	401.2	12.3	249.4
Modified Feature Search	0.05	102.5	7.4	47.2
Modified Monotony	0.00	79.4	6.6	36.5
Modified SEA	0.00	66.4	6.0	30.5
Modified SEA v2	0.13	123.6	7.9	61.9
Modified TSS	0.08	401.6	12.3	250.7
Monotony	-0.00	79.2	6.5	36.4
New DMD	0.07	342.2	11.7	206.3
One Dimensional Full Search	-0.02	470.7	14.2	269.9
PBIL	0.00	124.8	8.1	59.0
Parallel Hierarchical One D Search	-0.13	809.9	18.8	457.1
Plus Pattern Search	0.04	405.6	12.7	243.3
Prefiltered Sampled Points 2	0.32	172.0	9.3	85.6
Prefiltered Sampled Points 3	3.48	277.1	11.8	150.7
Prefiltered Sampled Points 4	2.18	317.5	11.8	183.5
Prefiltered Sampled Points 5	1.43	271.2	11.0	153.3
Projection Search	0.11	146.0	8.8	68.6
Sampled Pts 2	0.01	70.8	6.2	32.4
Sampled Pts 3	0.01	74.7	6.4	33.9
Sampled Pts 4	0.01	76.5	6.5	34.8
Sampled Pts 5	0.01	79.0	6.6	35.9
Subblock Full Search	0.03	337.1	11.1	213.0
TSS	0.10	394.1	12.1	246.6
Tile search	-0.05	85.5	7.0	36.6
Variance	0.02	145.2	8.7	69.8

Table B.7: Comparison of Algorithms for the sequence **cnn_news**

Algorithm	Motion Vector		PSNR	DFD
	Mean	Variance		(x 10^8)
2Dlog Search	3.847	6.301	21.78	4.129
Block Based Gradient Descent Search	2.549	6.810	21.34	4.440
Conjugate Directional Search	3.197	7.532	21.25	4.496
Conjugate Search	3.085	6.548	21.39	4.456
Cross Pattern Search	5.855	8.501	22.06	4.026
Cross Search	5.010	9.551	22.01	4.070
Dynamic SW Adjustment	3.318	3.488	21.43	4.348
Energy	196.382	21501	29.17	2.124
Four Step Search	4.340	8.003	22.07	3.999
Improved TSS	8.826	32.431	24.14	3.174
Modified Conjugate Search	5.336	9.613	22.10	4.005
Modified Feature Search	92.480	18674	28.02	2.418
Modified Monotony	204.135	22185	29.13	2.131
Modified SEA	170.510	20046	29.91	1.948
Modified SEA v2	229.765	21278	27.21	2.554
Modified TSS	4.146	9.135	22.09	3.993
Monotony	203.235	22103	29.14	2.128
New DMD	6.511	14.479	22.79	3.789
One Dimensional Full Search	3.788	20.419	21.40	4.606
PBIL	83.880	11731	27.17	2.637
Parallel Hierarchical One D Search	0.001	0.005	19.05	6.105
Plus Pattern Search	6.017	8.079	22.05	4.141
Prefiltered Sampled Points 2	195.302	36344	25.77	3.025
Prefiltered Sampled Points 3	76.179	23526	23.71	3.825
Prefiltered Sampled Points 4	50.527	14920	23.11	3.829
Prefiltered Sampled Points 5	52.363	13652	23.80	3.560
Projection Search	197.904	19501	26.49	2.860
Sampled Pts 2	169.757	19805	29.63	2.015
Sampled Pts 3	175.042	20068	29.40	2.075
Sampled Pts 4	172.409	19835	29.30	2.098
Sampled Pts 5	175.534	20120	29.15	2.134
Subblock Full Search	6.252	11.520	22.85	3.621
TSS	5.087	7.974	22.18	3.947
Tile search	46.885	7769	28.81	2.273
Variance	223.836	20166	26.51	2.821

Table B.8: Comparison of Algorithms for the sequence **cnn_news**

Algorithm	Entropy		
	Pixel	Motion Vector	Cumulative
2Dlog Search	6.06	3.50	6.02
Block Based Gradient Descent Search	6.18	2.97	6.13
Conjugate Directional Search	6.20	3.04	6.15
Conjugate Search	6.19	3.03	6.14
Cross Pattern Search	6.03	3.93	6.00
Cross Search	6.05	3.79	6.01
Dynamic SW Adjustment	6.14	3.34	6.10
Energy	5.14	9.19	5.38
Four Step Search	6.02	3.66	5.98
Improved TSS	5.69	4.64	5.67
Modified Conjugate Search	6.02	3.61	5.99
Modified Feature Search	5.33	7.07	5.43
Modified Monotony	5.15	9.23	5.39
Modified SEA	5.02	8.92	5.25
Modified SEA v2	5.41	9.49	5.66
Modified TSS	6.02	3.58	5.98
Monotony	5.14	9.23	5.39
New DMD	5.95	4.30	5.92
One Dimensional Full Search	6.25	2.40	6.20
PBIL	5.45	5.60	5.53
Parallel Hierarchical One D Search	6.65	0.00	6.59
Plus Pattern Search	6.08	3.94	6.05
Prefiltered Sampled Points 2	5.65	7.15	5.81
Prefiltered Sampled Points 3	5.96	4.69	6.00
Prefiltered Sampled Points 4	5.97	4.55	5.99
Prefiltered Sampled Points 5	5.88	5.00	5.90
Projection Search	5.57	9.29	5.79
Sampled Pts 2	5.06	8.46	5.28
Sampled Pts 3	5.11	8.95	5.33
Sampled Pts 4	5.12	8.22	5.33
Sampled Pts 5	5.15	8.93	5.37
Subblock Full Search	5.87	3.92	5.84
TSS	6.00	3.77	5.96
Tile search	5.23	6.24	5.30
Variance	5.55	9.47	5.79

Table B.9: Comparison of Algorithms for the sequence **cnn_news**

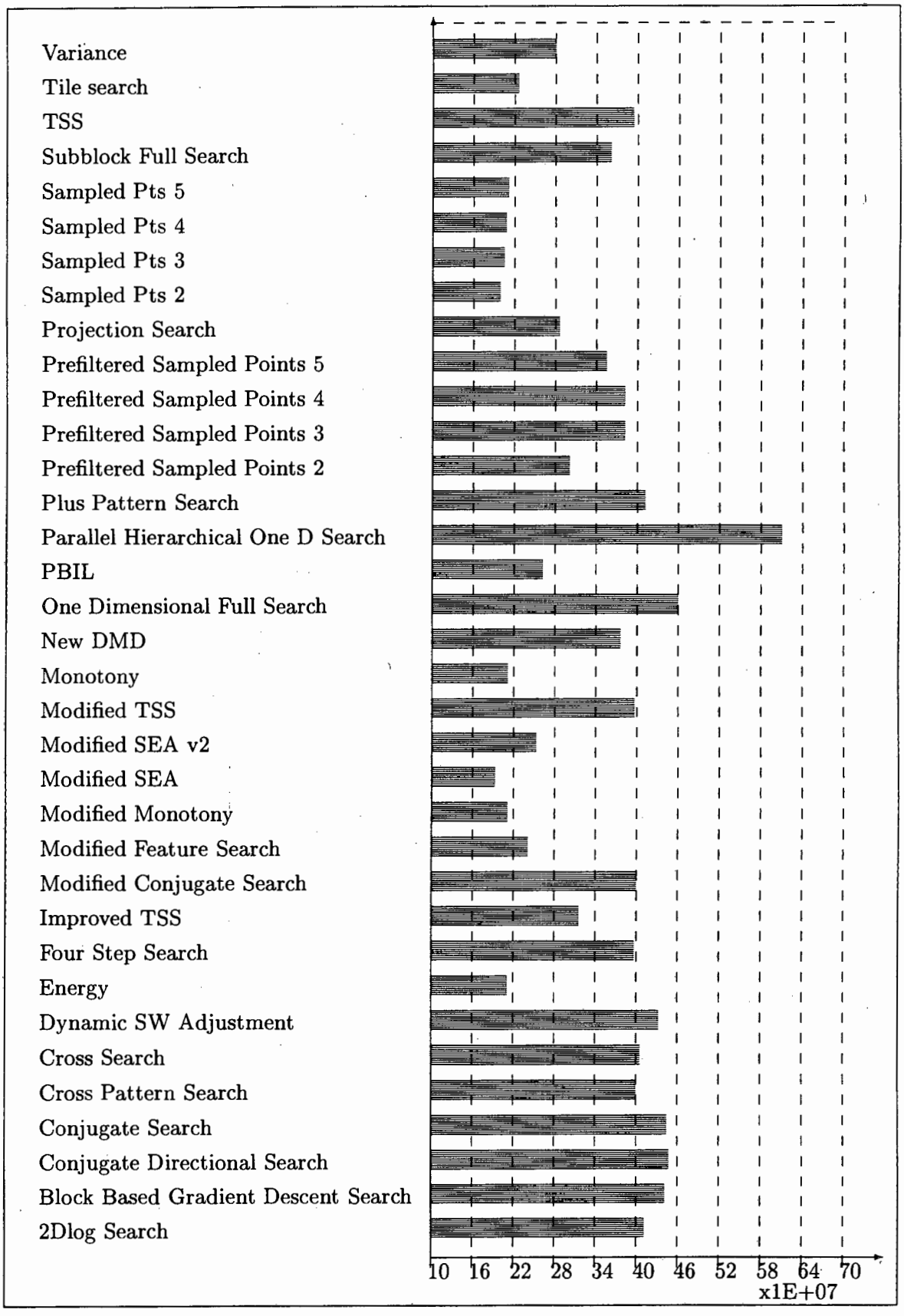


Figure B.19: **DFD** for sequence **cnn_news**

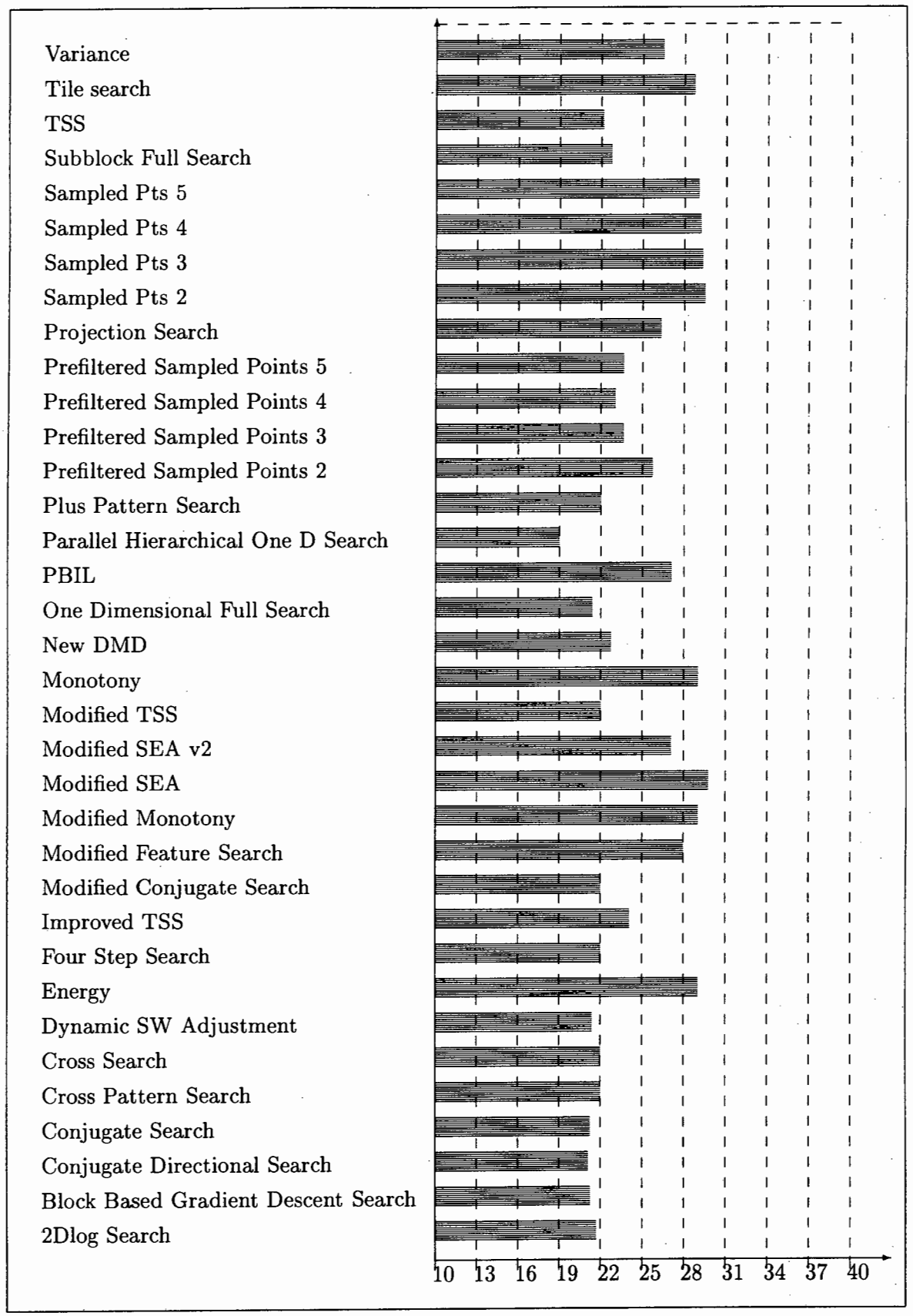


Figure B.20: **PSNR** for sequence **cnn_news**

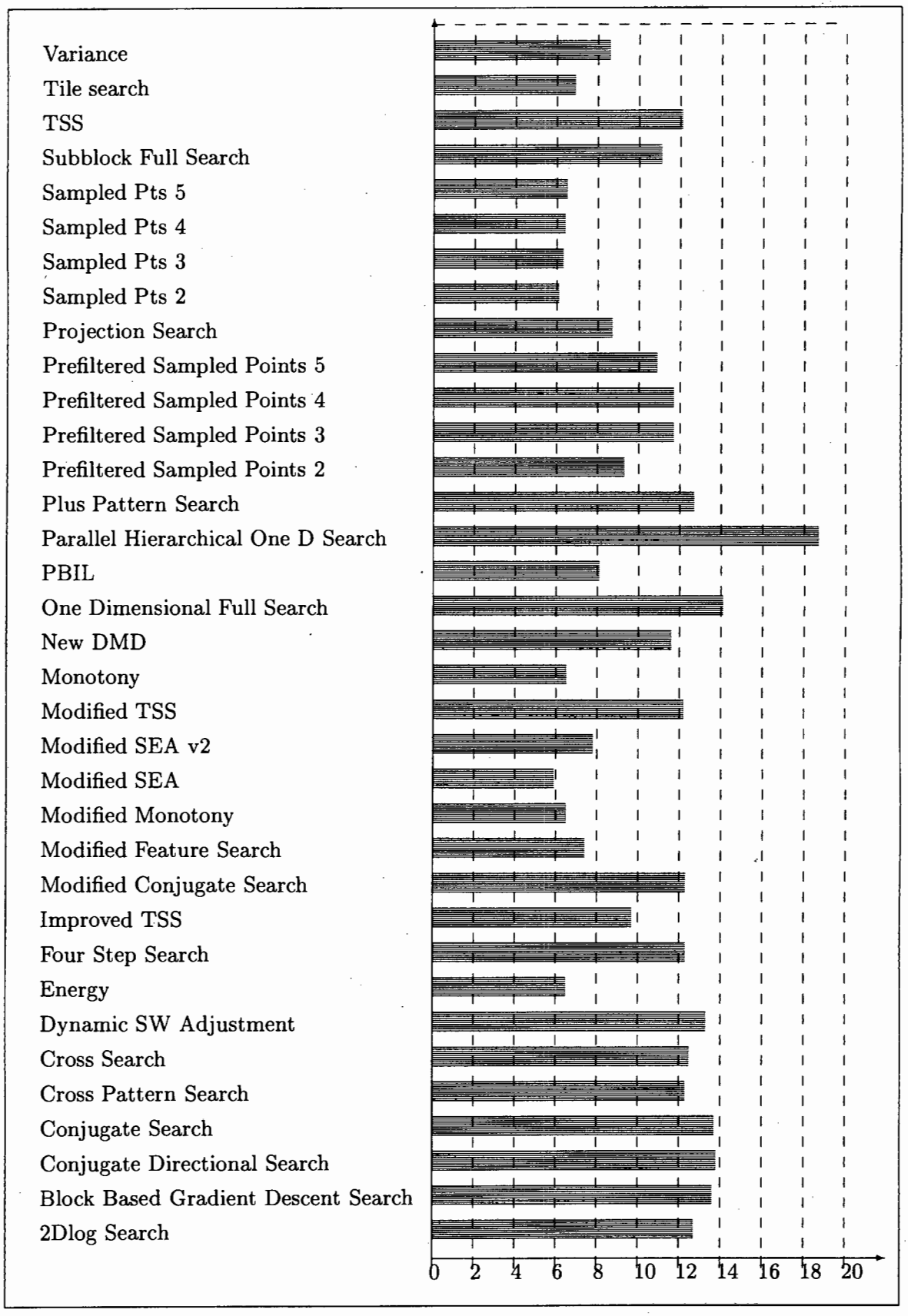


Figure B.21: **Di abs mean** for sequence **cnn_news**

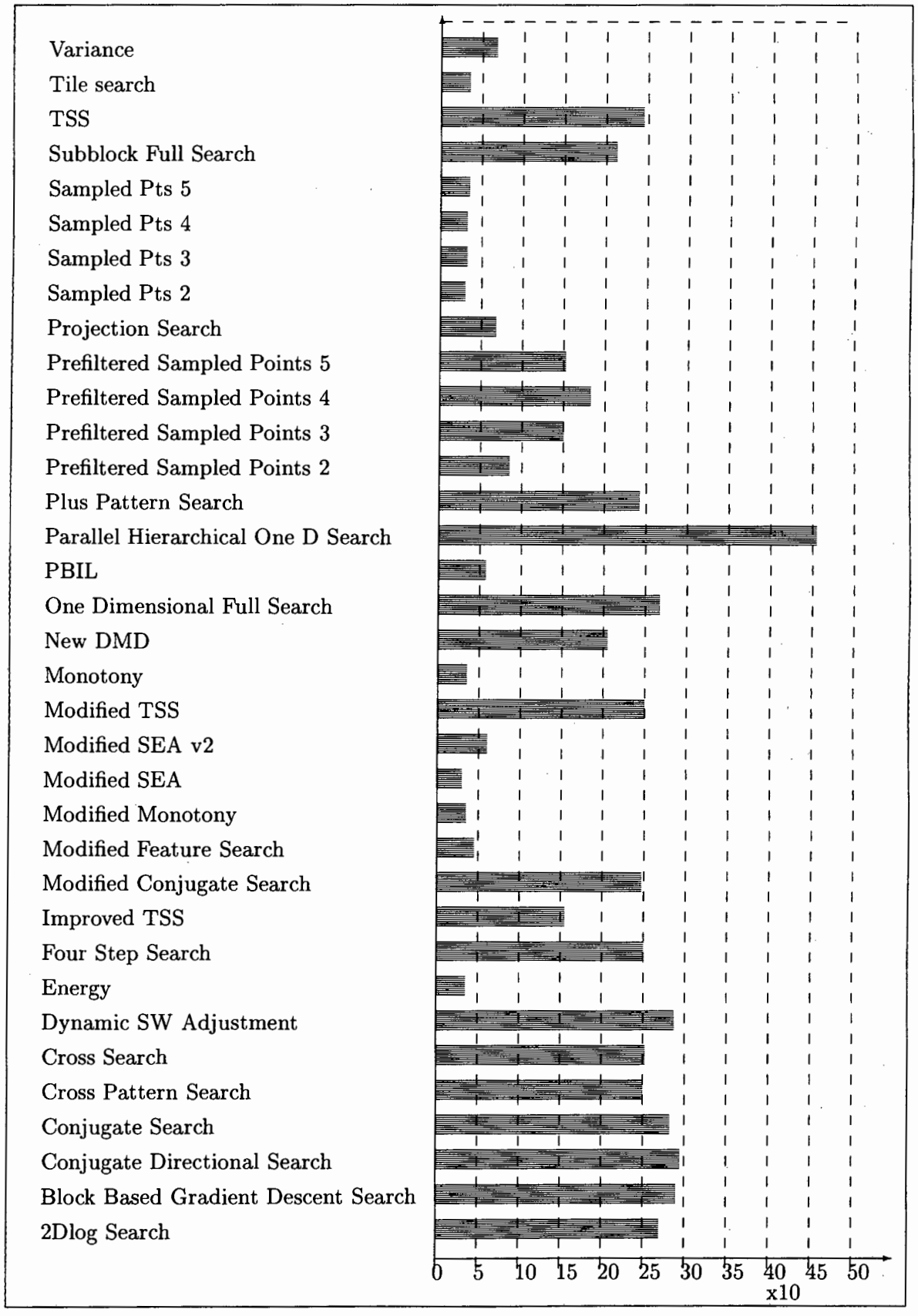


Figure B.22: **Di abs var** for sequence **cnn_news**

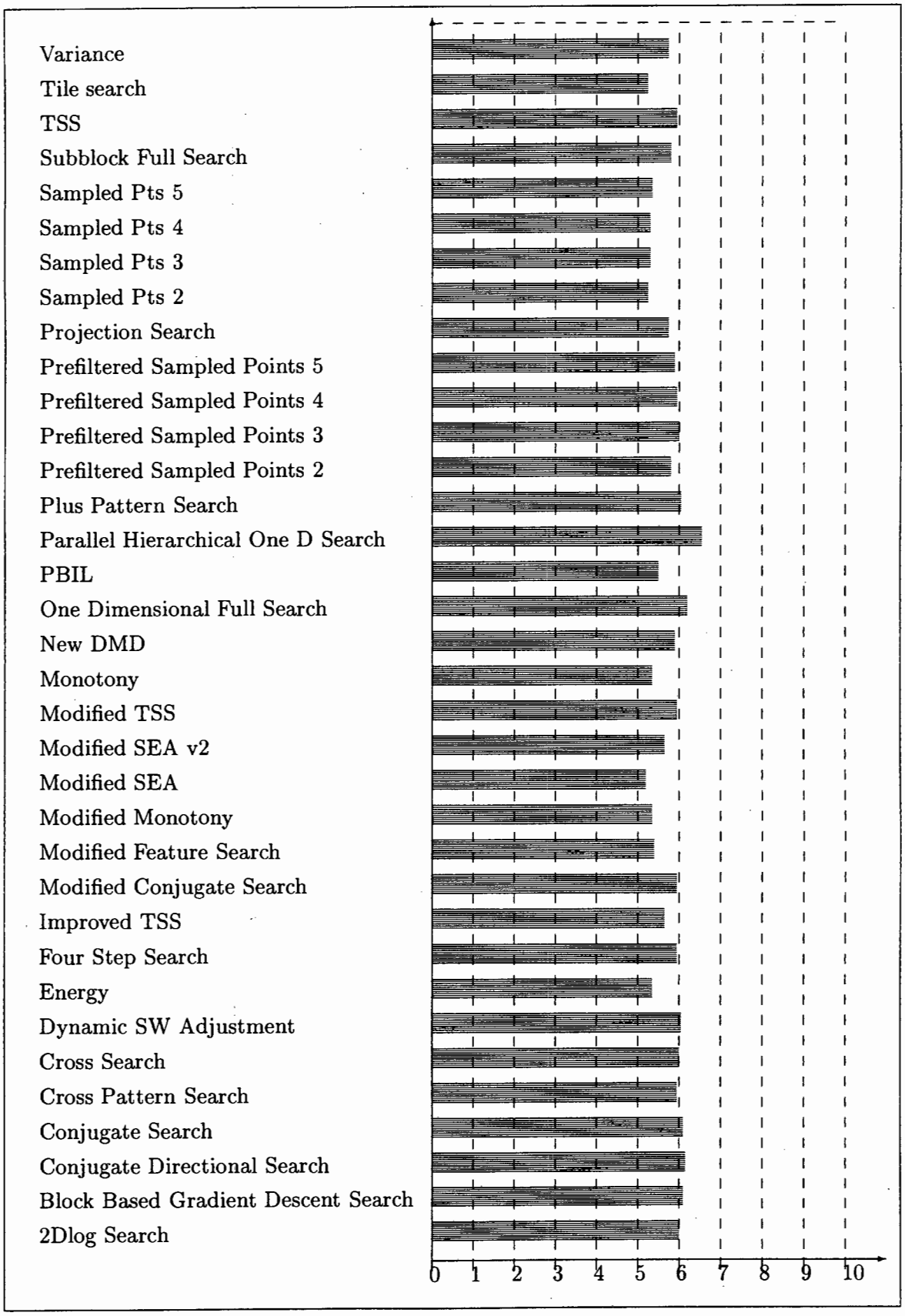


Figure B.23: **Entropy cumulative** for sequence **cnn_news**

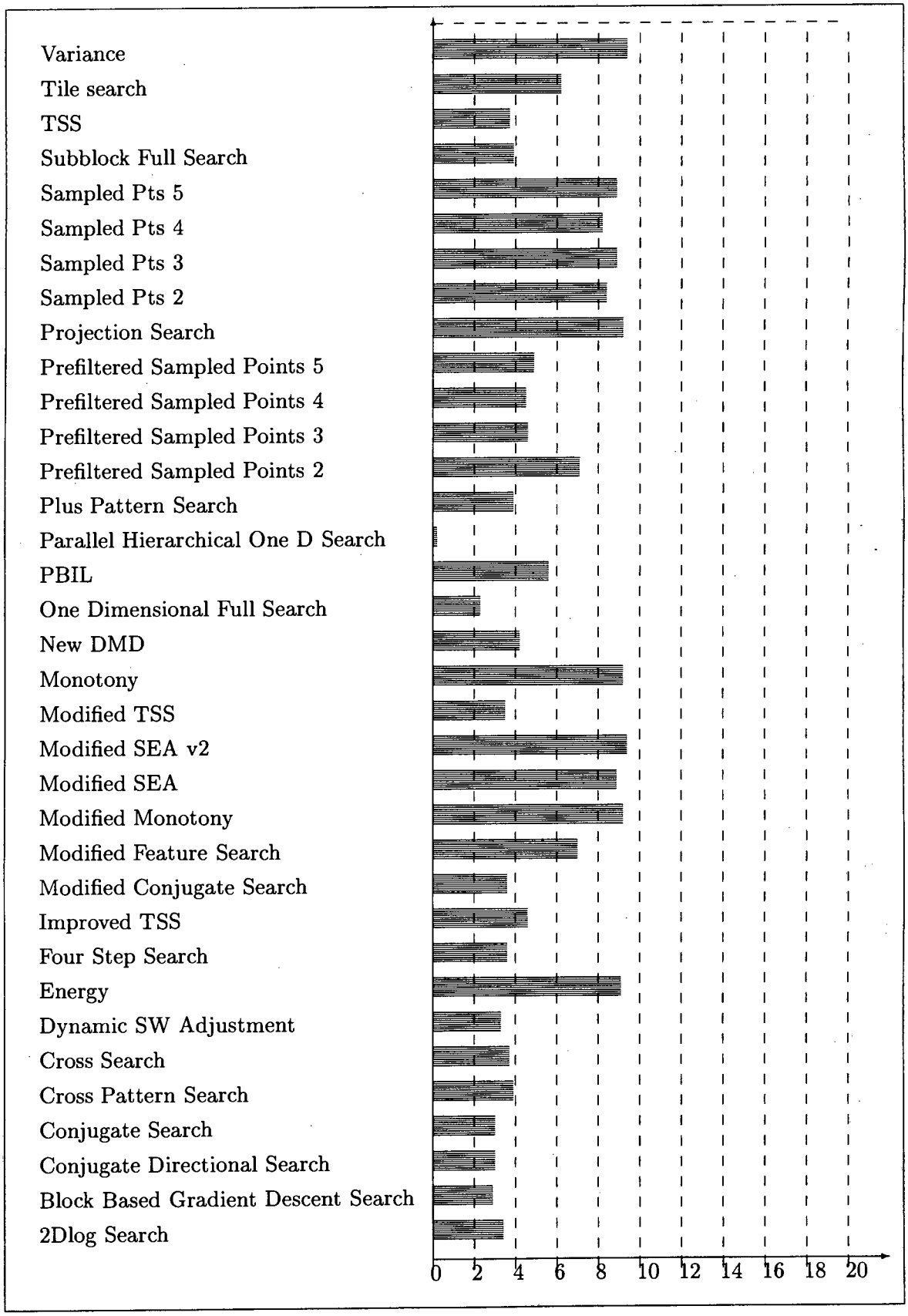


Figure B.24: **Entropy mtn vec** for sequence **cnn_news**

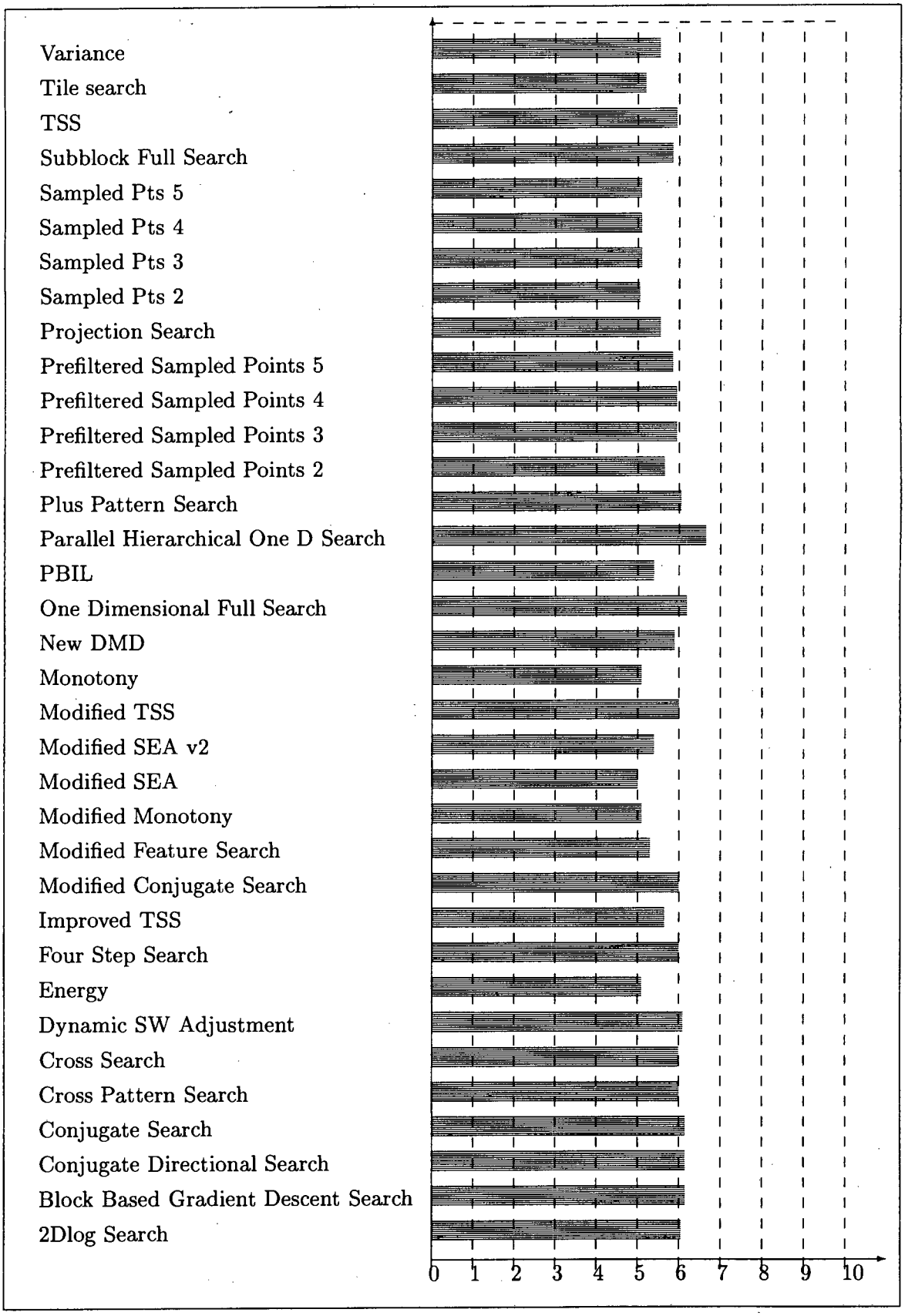


Figure B.25: **Entropy pixel** for sequence **cnn_news**

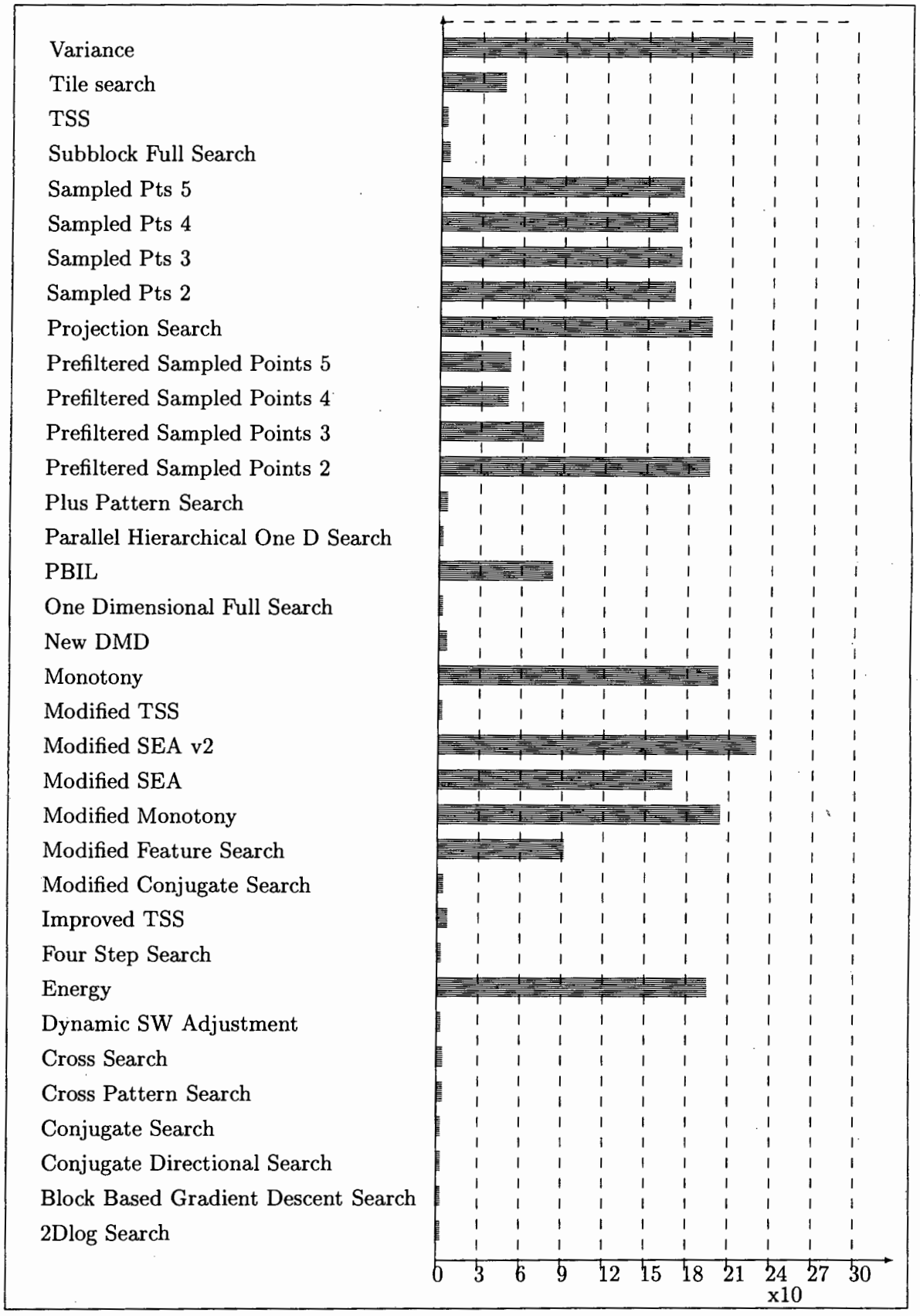


Figure B.26: **Motion Vector Means** for sequence **cnn_news**

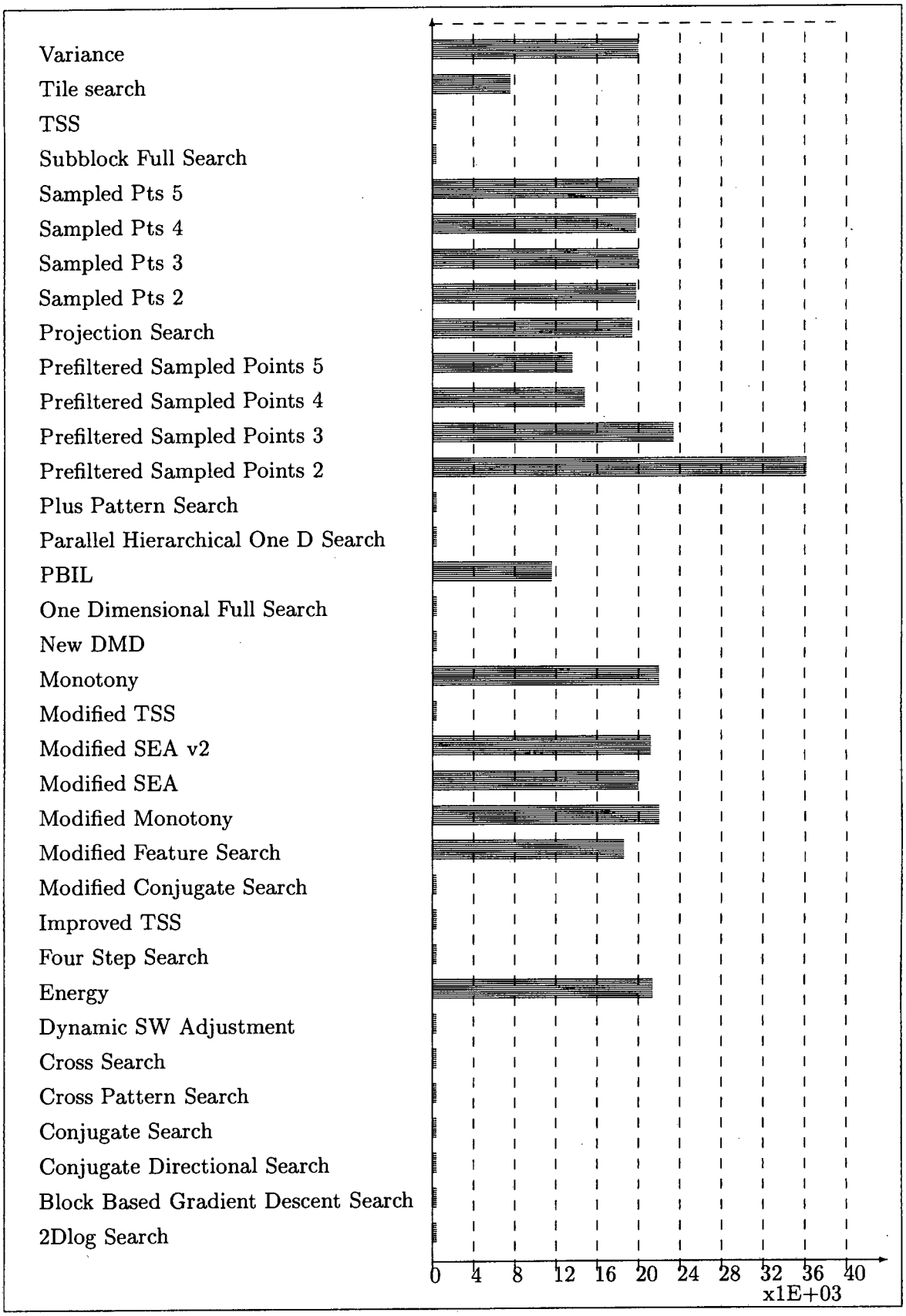


Figure B.27: **Motion Vector Var** for sequence **cnn_news**

Algorithm	Difference block			
			Absolute	
	Mean	Variance	Mean	Variance
2Dlog Search	0.28	394.7	10.2	289.8
Block Based Gradient Descent Search	-0.09	476.5	11.0	355.6
Conjugate Directional Search	0.01	484.2	11.1	361.0
Conjugate Search	-0.02	477.6	11.1	355.3
Cross Pattern Search	0.05	370.9	10.4	263.5
Cross Search	0.38	413.9	10.6	301.8
Dynamic SW Adjustment	0.06	457.6	10.8	341.6
Energy	-0.51	140.9	7.3	88.2
Four Step Search	0.30	384.7	10.1	282.3
Improved TSS	0.15	255.6	8.7	180.4
Modified Conjugate Search	0.21	366.1	10.0	266.0
Modified Feature Search	0.17	215.0	8.5	143.4
Modified Monotony	0.25	160.7	7.7	102.0
Modified SEA	0.14	110.6	6.5	68.5
Modified SEA v2	0.11	121.6	6.8	74.7
Modified TSS	0.23	383.2	10.1	281.6
Monotony	0.25	159.1	7.6	101.0
New DMD	0.31	359.1	10.4	250.6
One Dimensional Full Search	-0.06	567.7	12.5	412.3
PBIL	0.37	237.2	8.8	159.7
Parallel Hierarchical One D Search	-0.34	820.4	14.6	607.3
Plus Pattern Search	0.08	426.1	11.0	306.0
Prefiltered Sampled Points 2	1.37	247.5	8.9	170.3
Prefiltered Sampled Points 3	0.17	142.0	7.1	91.1
Prefiltered Sampled Points 4	0.80	138.8	7.3	86.4
Prefiltered Sampled Points 5	0.19	163.7	7.5	107.4
Projection Search	0.11	375.7	10.9	256.9
Sampled Pts 2	0.16	118.0	6.7	73.0
Sampled Pts 3	0.18	141.4	7.1	90.6
Sampled Pts 4	0.18	129.5	7.0	80.4
Sampled Pts 5	0.22	161.0	7.5	105.4
Subblock Full Search	0.12	285.7	9.0	204.1
TSS	0.32	369.5	10.0	269.7
Tile search	0.08	137.4	7.3	84.4
Variance	0.04	310.2	10.1	207.8

Table B.10: Comparison of Algorithms for the sequence **crawls**

Algorithm	Motion Vector		PSNR	DFD
	Mean	Variance		(x 10^8)
2Dlog Search	2.481	3.728	22.17	3.330
Block Based Gradient Descent Search	1.546	2.498	21.35	3.575
Conjugate Directional Search	1.824	3.186	21.28	3.609
Conjugate Search	1.793	2.939	21.34	3.595
Cross Pattern Search	5.014	8.428	22.44	3.369
Cross Search	3.755	7.986	21.96	3.443
Dynamic SW Adjustment	2.364	2.378	21.53	3.501
Energy	136.212	15651	26.64	2.365
Four Step Search	2.836	4.621	22.28	3.290
Improved TSS	6.864	27.469	24.06	2.819
Modified Conjugate Search	3.792	7.778	22.49	3.252
Modified Feature Search	42.593	9176	24.81	2.752
Modified Monotony	158.794	17835	26.07	2.492
Modified SEA	100.085	11006	27.69	2.108
Modified SEA v2	117.103	12982	27.28	2.226
Modified TSS	2.814	5.541	22.30	3.277
Monotony	154.662	17251	26.12	2.478
New DMD	4.926	9.501	22.58	3.388
One Dimensional Full Search	1.466	8.692	20.59	4.052
PBIL	60.012	7906	24.38	2.864
Parallel Hierarchical One D Search	0.001	0.004	18.99	4.747
Plus Pattern Search	5.199	8.045	21.84	3.562
Prefiltered Sampled Points 2	158.267	25657	24.19	2.891
Prefiltered Sampled Points 3	111.939	11974	26.61	2.321
Prefiltered Sampled Points 4	114.059	13214	26.71	2.366
Prefiltered Sampled Points 5	115.987	12578	25.99	2.438
Projection Search	144.166	14152	22.38	3.543
Sampled Pts 2	103.000	11238	27.41	2.182
Sampled Pts 3	111.632	11930	26.63	2.316
Sampled Pts 4	109.217	11862	27.01	2.279
Sampled Pts 5	115.620	12507	26.06	2.425
Subblock Full Search	4.953	10.824	23.57	2.935
TSS	3.761	6.299	22.45	3.248
Tile search	38.173	6991	26.75	2.365
Variance	176.331	17958	23.21	3.289

Table B.11: Comparison of Algorithms for the sequence **crawls**

Algorithm	Entropy		
	Pixel	Motion Vector	Cumulative
2Dlog Search	5.64	2.99	5.60
Block Based Gradient Descent Search	5.72	2.38	5.67
Conjugate Directional Search	5.73	2.41	5.68
Conjugate Search	5.73	2.39	5.68
Cross Pattern Search	5.70	3.86	5.66
Cross Search	5.70	3.56	5.66
Dynamic SW Adjustment	5.69	2.82	5.64
Energy	5.27	8.55	5.44
Four Step Search	5.63	3.18	5.58
Improved TSS	5.46	4.39	5.44
Modified Conjugate Search	5.62	3.33	5.58
Modified Feature Search	5.46	5.02	5.48
Modified Monotony	5.34	8.80	5.53
Modified SEA	5.11	7.88	5.26
Modified SEA v2	5.19	8.19	5.36
Modified TSS	5.62	3.14	5.58
Monotony	5.33	8.77	5.52
New DMD	5.72	3.97	5.68
One Dimensional Full Search	5.92	1.35	5.86
PBIL	5.51	5.10	5.56
Parallel Hierarchical One D Search	6.08	0.00	6.02
Plus Pattern Search	5.76	3.90	5.73
Prefiltered Sampled Points 2	5.51	7.74	5.66
Prefiltered Sampled Points 3	5.24	8.14	5.39
Prefiltered Sampled Points 4	5.27	7.58	5.42
Prefiltered Sampled Points 5	5.30	8.18	5.45
Projection Search	5.79	8.83	5.94
Sampled Pts 2	5.16	7.45	5.31
Sampled Pts 3	5.23	8.14	5.39
Sampled Pts 4	5.22	7.29	5.37
Sampled Pts 5	5.29	8.18	5.45
Subblock Full Search	5.50	3.80	5.47
TSS	5.62	3.48	5.58
Tile search	5.28	5.00	5.31
Variance	5.70	9.08	5.88

Table B.12: Comparison of Algorithms for the sequence **crawls**

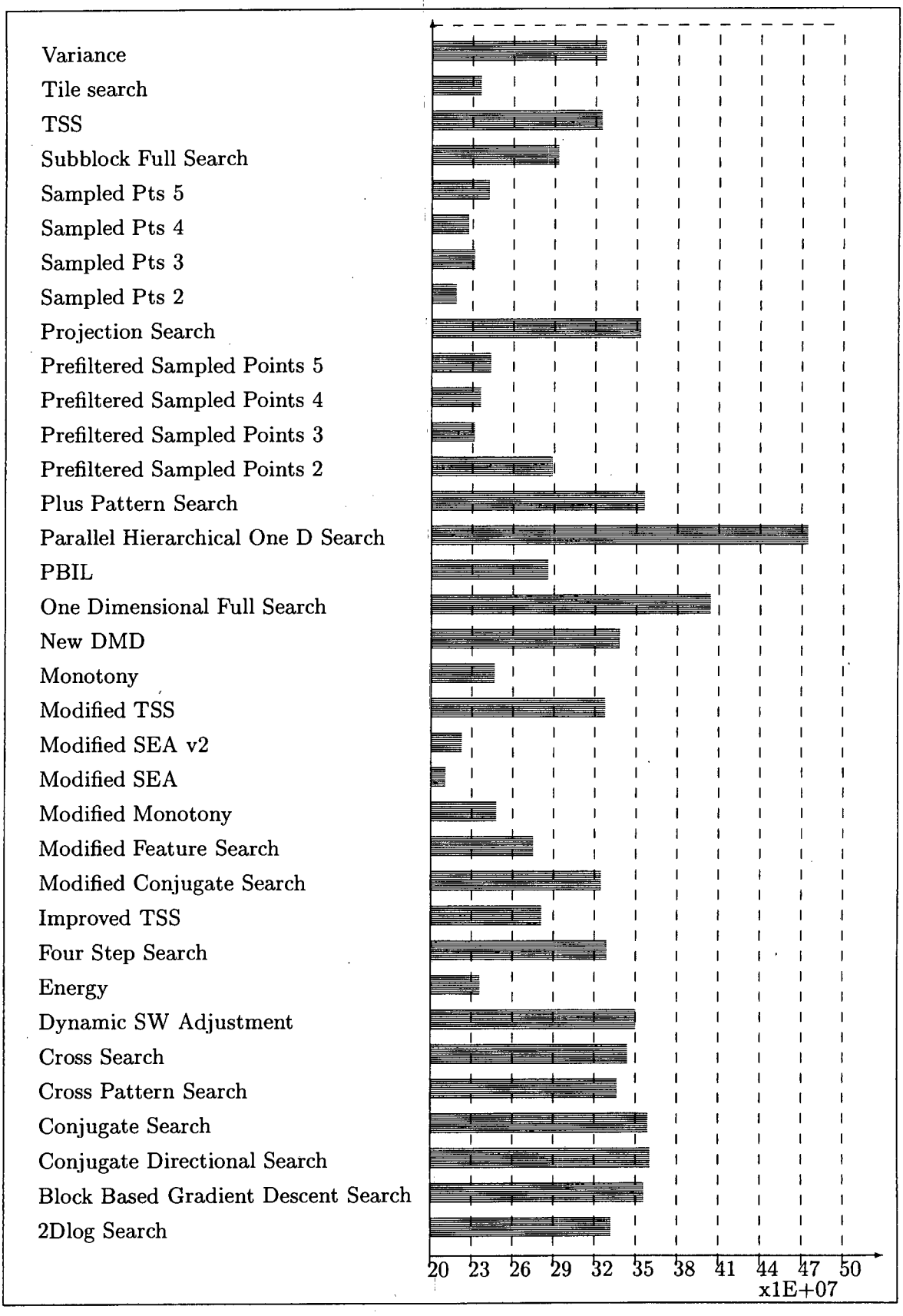


Figure B.28: **DFD** for sequence **crawls**

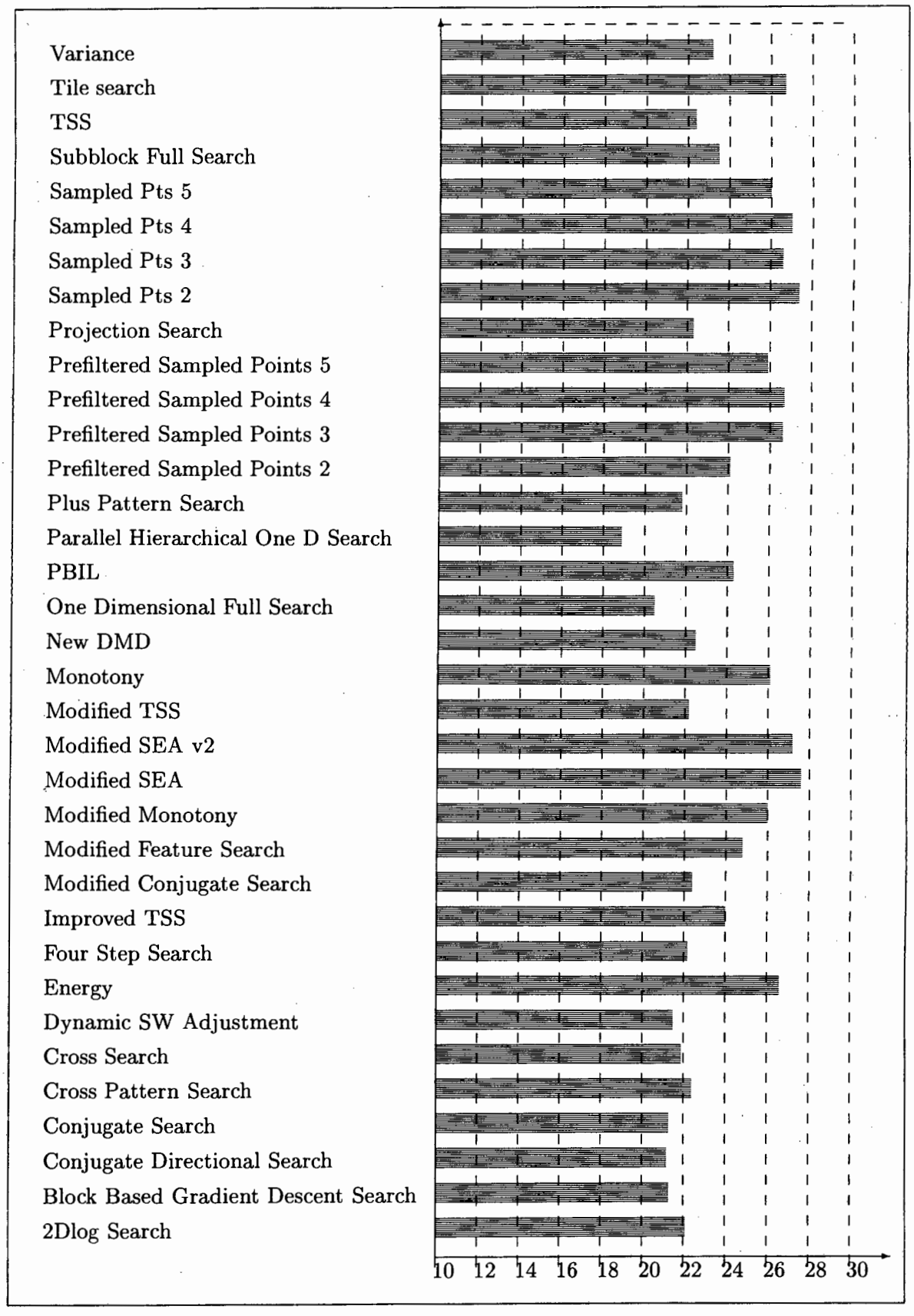


Figure B.29: **PSNR** for sequence **crawls**

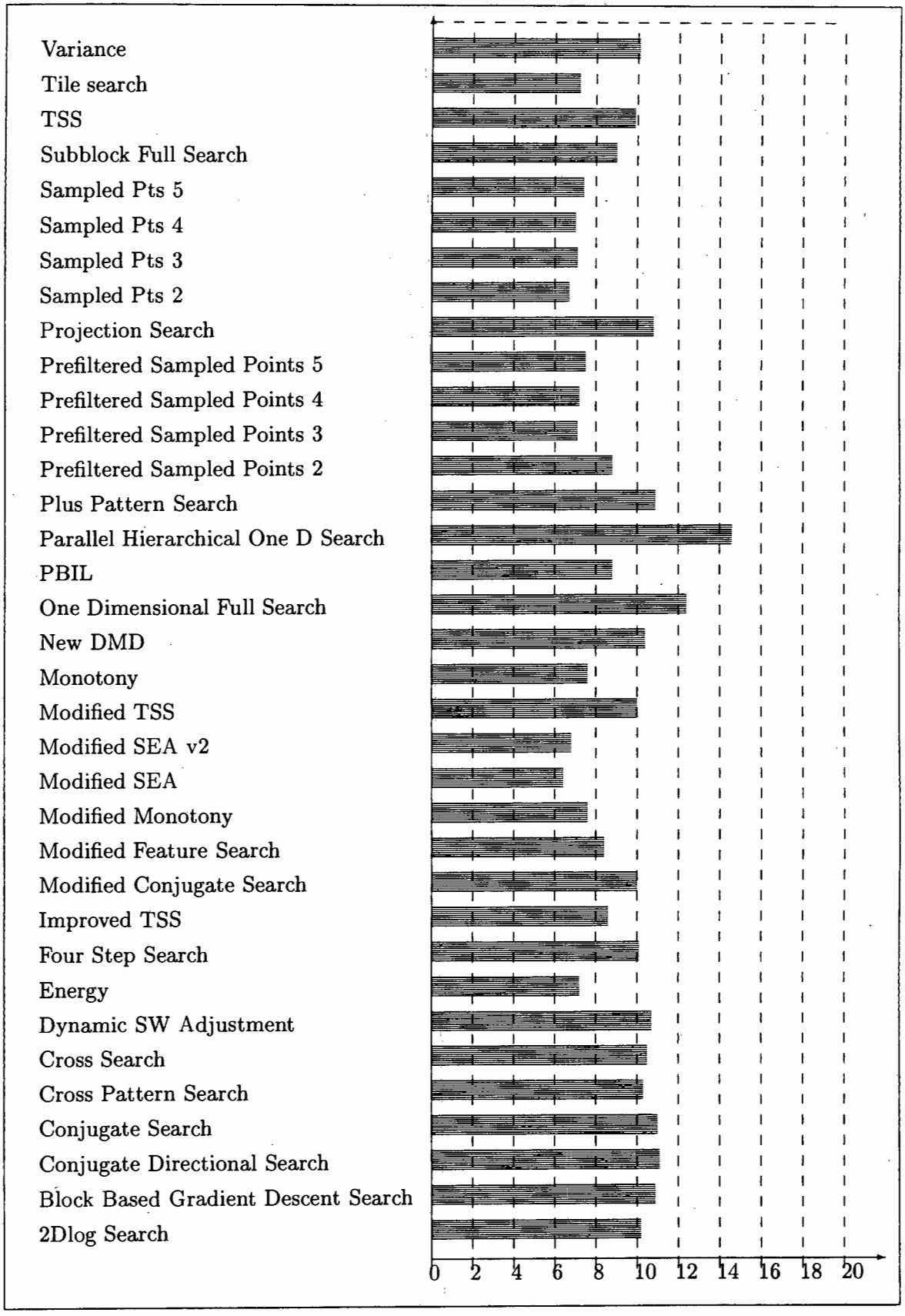


Figure B.30: **Di abs mean** for sequence **crawls**

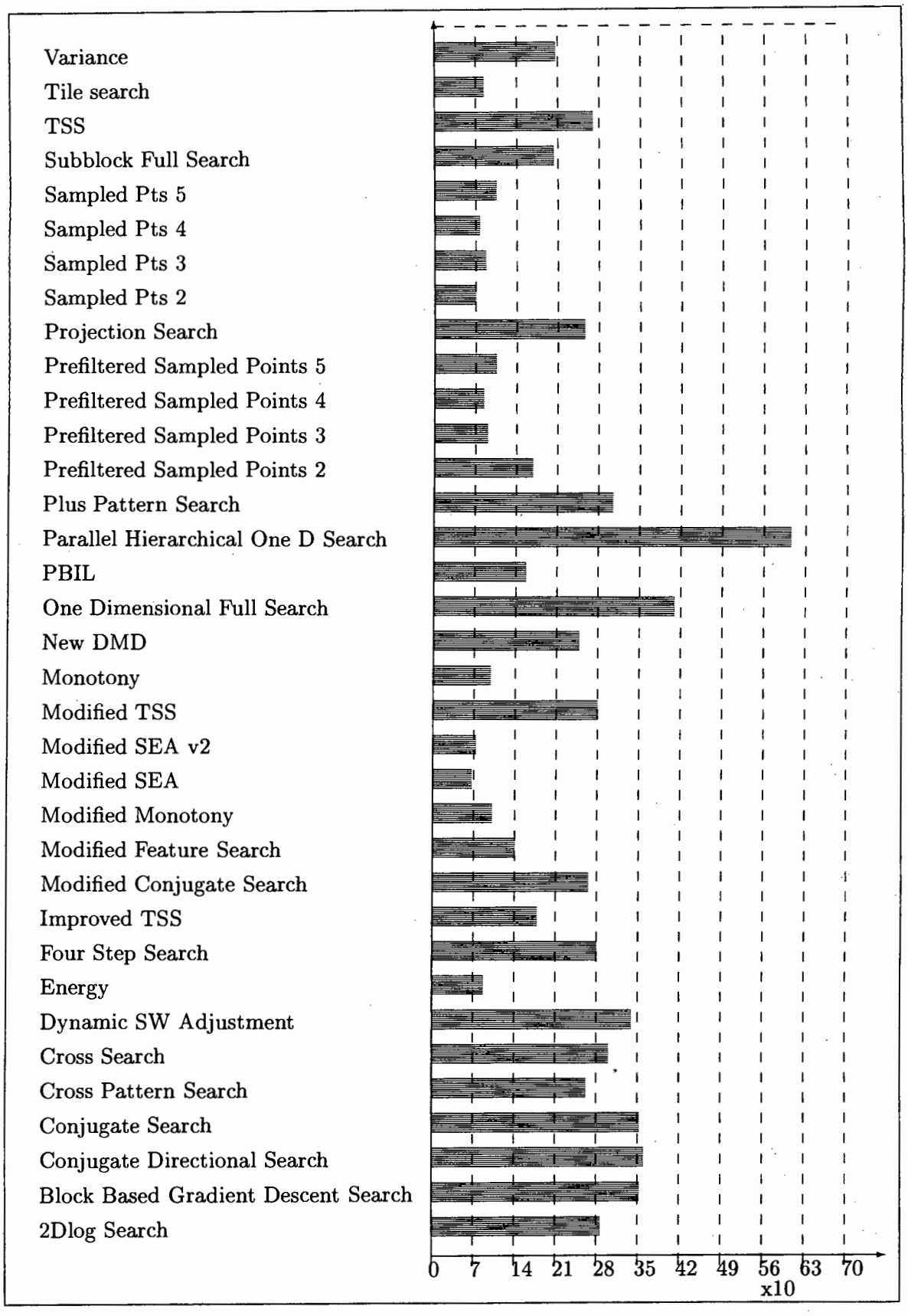


Figure B.31: **Di abs var** for sequence **crawls**

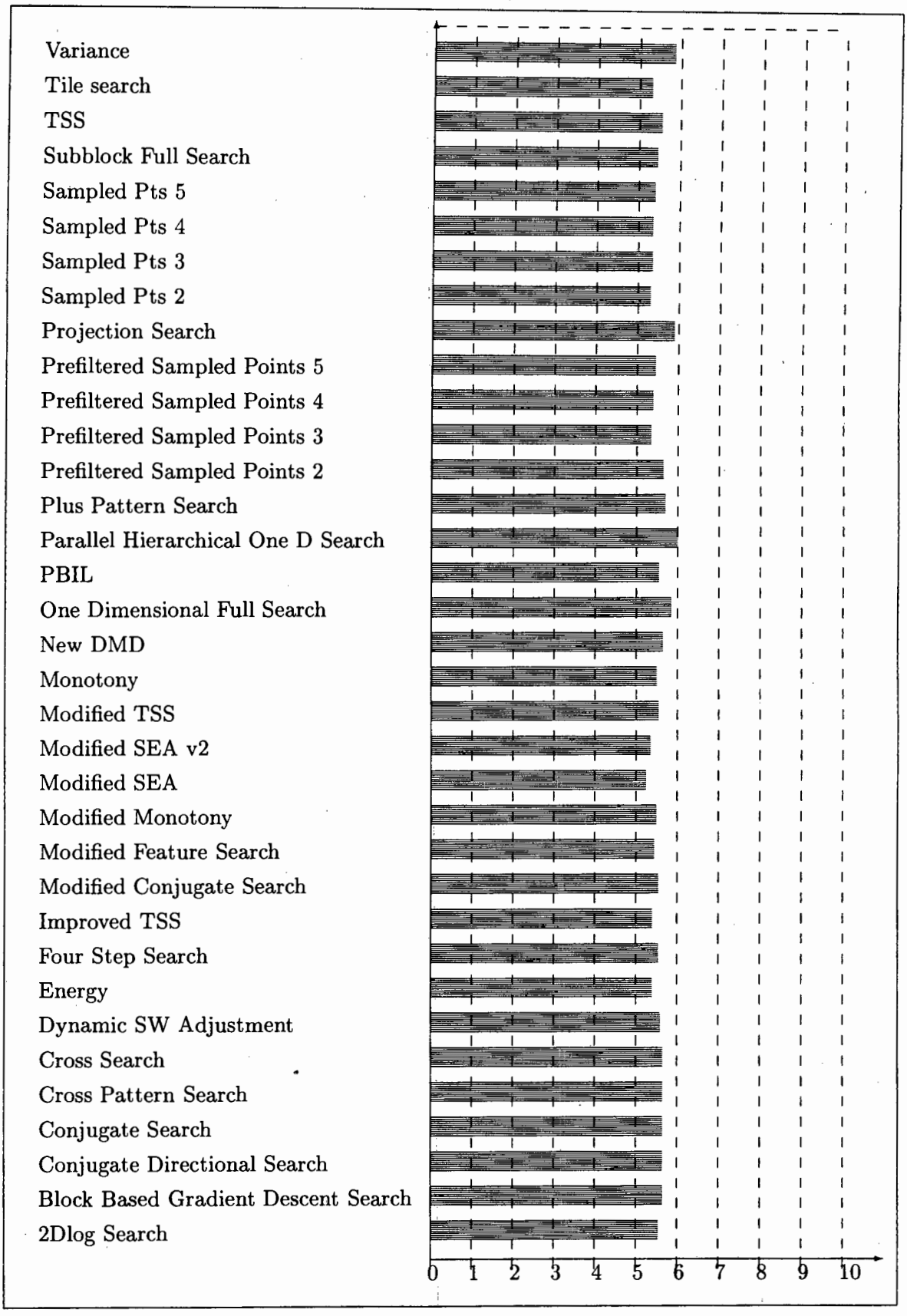


Figure B.32: **Entropy cumulative** for sequence **crawls**

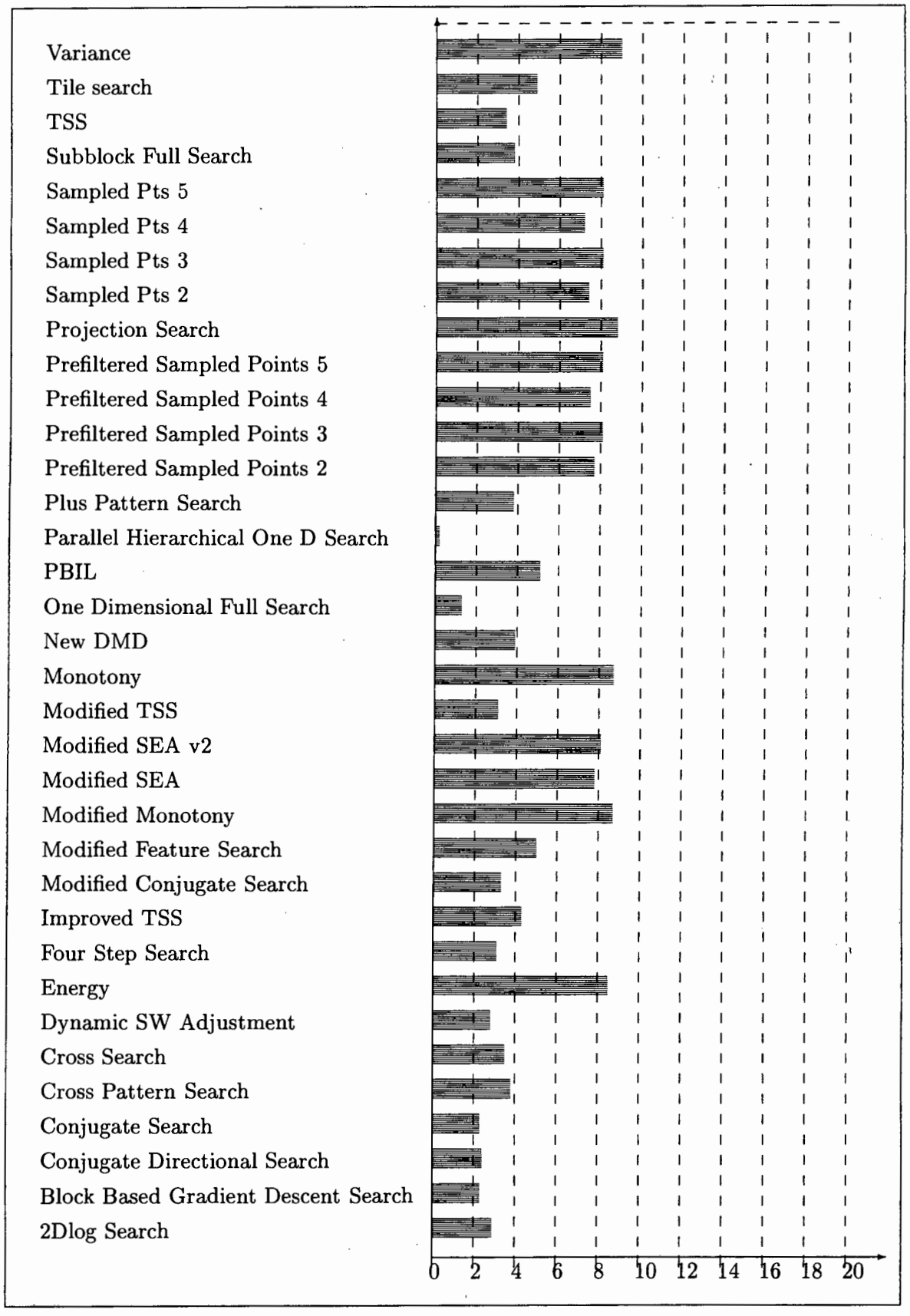


Figure B.33: **Entropy mtn vec** for sequence **crawls**

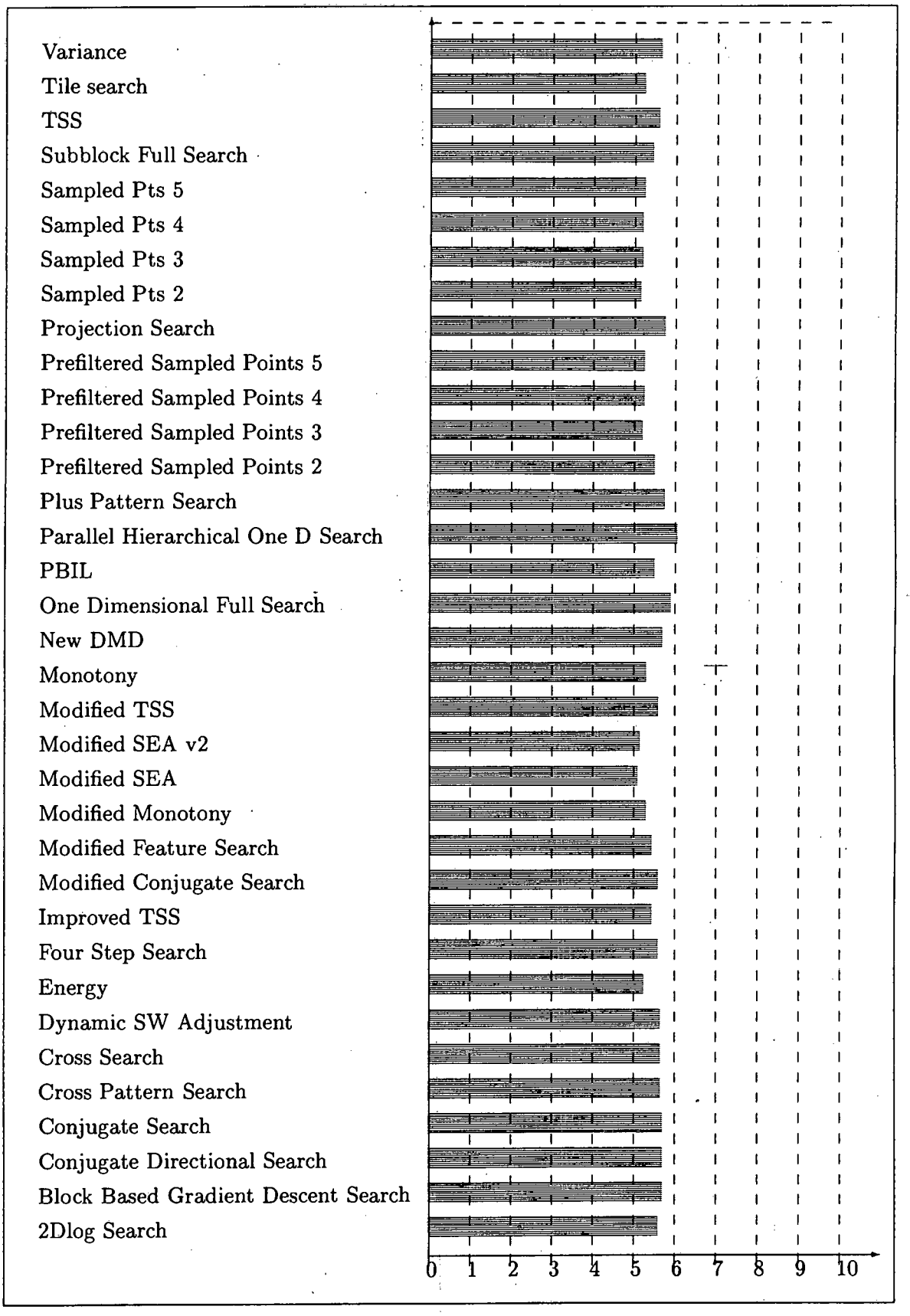


Figure B.34: **Entropy pixel** for sequence **crawls**

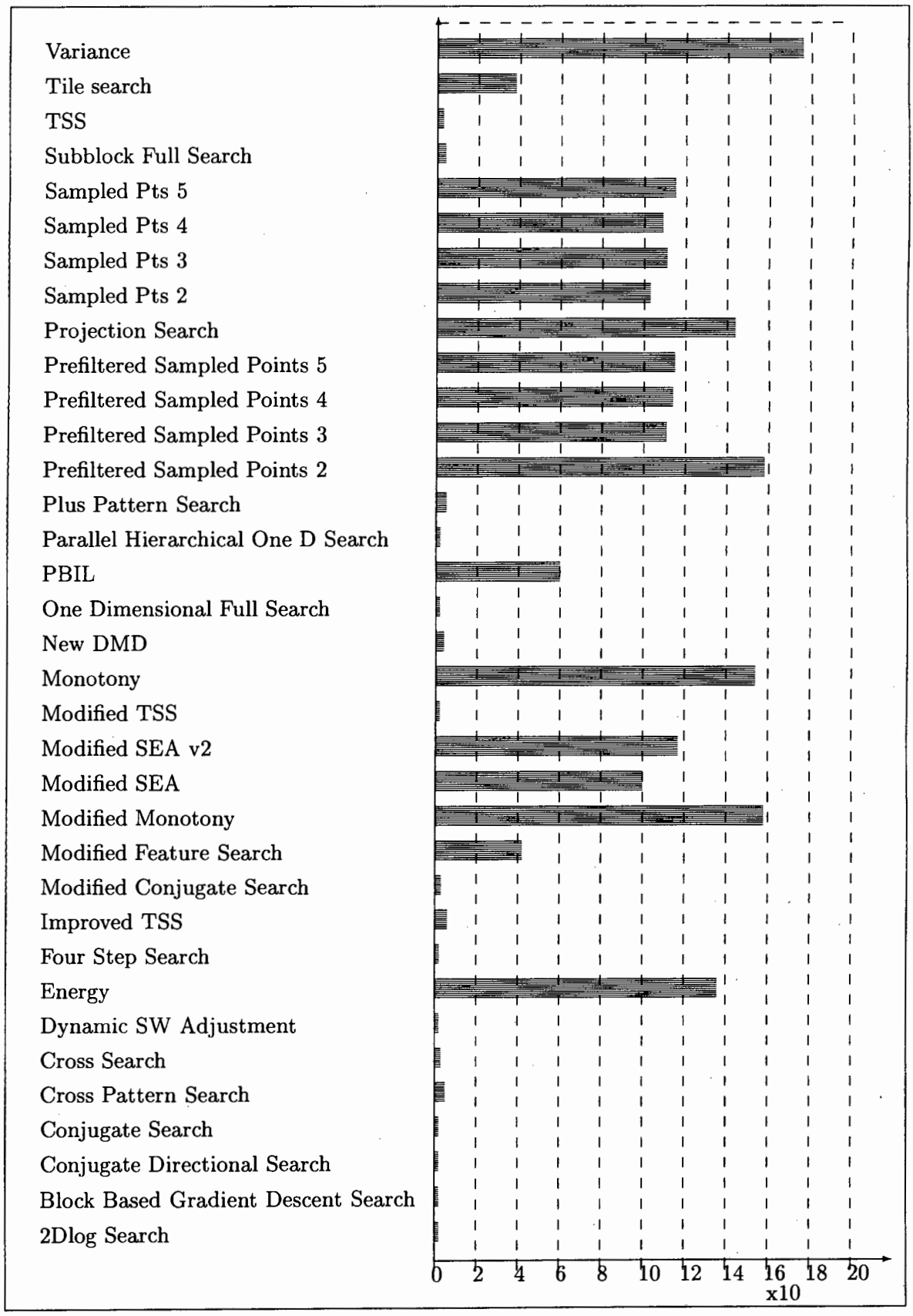


Figure B.35: **Motion Vector Means** for sequence **crawls**

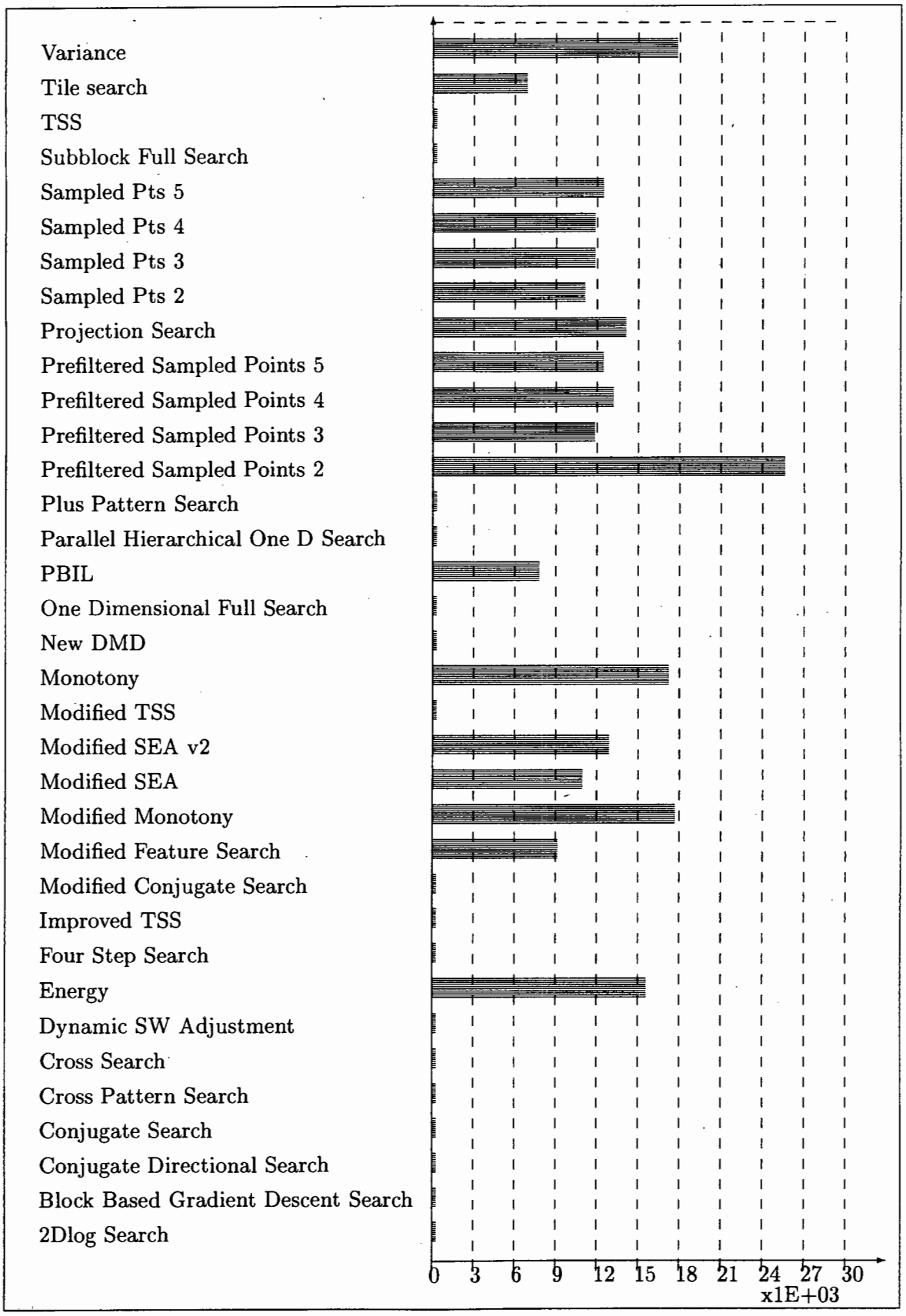


Figure B.36: **Motion Vector Var** for sequence **crawls**

Algorithm	Difference block			
			Absolute	
	Mean	Variance	Mean	Variance
2Dlog Search	0.04	834.6	15.9	581.5
Block Based Gradient Descent Search	-0.01	895.1	16.6	619.6
Conjugate Directional Search	0.11	932.4	17.1	639.5
Conjugate Search	0.02	912.2	17.0	624.8
Cross Pattern Search	0.07	756.4	14.9	535.2
Cross Search	0.01	805.8	15.7	560.1
Dynamic SW Adjustment	0.05	875.5	16.4	605.0
Energy	-0.23	95.5	5.2	68.3
Four Step Search	0.06	801.7	15.5	560.2
Improved TSS	0.15	523.6	12.1	376.7
Modified Conjugate Search	0.04	779.4	15.3	544.4
Modified Feature Search	-0.03	142.0	6.6	97.8
Modified Monotony	-0.03	95.9	5.4	66.7
Modified SEA	-0.03	68.7	4.5	48.4
Modified SEA v2	-0.15	116.6	6.1	78.9
Modified TSS	0.04	779.0	15.3	544.8
Monotony	-0.04	95.1	5.4	66.1
New DMD	0.02	739.3	15.2	509.0
One Dimensional Full Search	0.11	906.0	17.1	612.7
PBIL	0.08	163.8	7.0	115.1
Parallel Hierarchical One D Search	0.04	1425.5	22.5	920.3
Plus Pattern Search	0.03	799.2	15.5	559.6
Prefiltered Sampled Points 2	1.40	122.8	6.6	80.9
Prefiltered Sampled Points 3	-0.04	80.8	4.9	57.0
Prefiltered Sampled Points 4	0.44	91.1	5.3	62.7
Prefiltered Sampled Points 5	-0.04	89.9	5.1	63.8
Projection Search	0.04	215.3	8.2	148.8
Sampled Pts 2	-0.01	76.8	4.8	54.0
Sampled Pts 3	-0.04	79.8	4.8	56.3
Sampled Pts 4	-0.01	83.3	5.0	58.6
Sampled Pts 5	-0.03	87.0	5.0	61.7
Subblock Full Search	0.06	672.4	13.9	479.1
TSS	0.02	784.9	15.4	549.1
Tile search	-0.06	93.0	6.0	57.3
Variance	0.12	219.1	8.0	154.5

Table B.13: Comparison of Algorithms for the sequence **cross_country**

Algorithm	Motion Vector		PSNR	DFD
	Mean	Variance		(x 10^8)
2Dlog Search	3.951	8.011	18.92	5.171
Block Based Gradient Descent Search	2.621	8.004	18.61	5.395
Conjugate Directional Search	3.229	9.458	18.43	5.563
Conjugate Search	3.111	8.324	18.53	5.511
Cross Pattern Search	5.772	11.717	19.34	4.834
Cross Search	4.881	11.948	19.07	5.095
Dynamic SW Adjustment	3.062	4.310	18.71	5.346
Energy	171.139	20194	28.33	1.698
Four Step Search	4.492	10.590	19.09	5.051
Improved TSS	9.001	43.397	20.94	3.940
Modified Conjugate Search	5.297	12.597	19.21	4.983
Modified Feature Search	107.245	19731	26.61	2.160
Modified Monotony	183.199	21631	28.31	1.756
Modified SEA	137.393	18682	29.76	1.464
Modified SEA v2	192.902	19632	27.46	1.996
Modified TSS	3.982	11.168	19.22	4.974
Monotony	182.054	21505	28.35	1.751
New DMD	6.763	16.913	19.44	4.932
One Dimensional Full Search	4.554	24.686	18.56	5.567
PBIL	96.049	13133	25.99	2.269
Parallel Hierarchical One D Search	0.001	0.006	16.59	7.306
Plus Pattern Search	5.893	11.191	19.10	5.031
Prefiltered Sampled Points 2	177.053	27533	27.24	2.155
Prefiltered Sampled Points 3	146.778	18961	29.05	1.588
Prefiltered Sampled Points 4	159.706	21415	28.54	1.736
Prefiltered Sampled Points 5	150.504	19184	28.59	1.662
Projection Search	179.255	18750	24.80	2.652
Sampled Pts 2	146.868	18990	29.28	1.550
Sampled Pts 3	146.375	18974	29.11	1.576
Sampled Pts 4	150.199	19180	28.93	1.615
Sampled Pts 5	149.107	19088	28.74	1.634
Subblock Full Search	6.153	15.806	19.85	4.518
TSS	5.007	10.970	19.18	4.992
Tile search	52.392	8424	28.44	1.944
Variance	208.405	19199	24.72	2.614

Table B.14: Comparison of Algorithms for the sequence **cross_country**

Algorithm	Entropy		
	Pixel	Motion Vector	Cumulative
2Dlog Search	6.20	3.44	6.14
Block Based Gradient Descent Search	6.26	2.81	6.20
Conjugate Directional Search	6.32	2.80	6.26
Conjugate Search	6.31	2.81	6.25
Cross Pattern Search	6.11	3.77	6.07
Cross Search	6.19	3.62	6.14
Dynamic SW Adjustment	6.25	3.19	6.19
Energy	4.73	8.82	4.95
Four Step Search	6.16	3.58	6.12
Improved TSS	5.82	4.38	5.80
Modified Conjugate Search	6.15	3.41	6.11
Modified Feature Search	5.09	6.91	5.20
Modified Monotony	4.79	8.90	5.01
Modified SEA	4.53	8.06	4.72
Modified SEA v2	4.99	9.13	5.22
Modified TSS	6.14	3.44	6.09
Monotony	4.79	8.89	5.01
New DMD	6.18	4.34	6.14
One Dimensional Full Search	6.35	2.60	6.29
PBIL	5.15	5.58	5.23
Parallel Hierarchical One D Search	6.73	0.00	6.64
Plus Pattern Search	6.17	3.81	6.13
Prefiltered Sampled Points 2	5.11	7.79	5.28
Prefiltered Sampled Points 3	4.64	8.21	4.84
Prefiltered Sampled Points 4	4.78	8.04	4.97
Prefiltered Sampled Points 5	4.70	8.23	4.90
Projection Search	5.37	9.01	5.56
Sampled Pts 2	4.61	7.90	4.81
Sampled Pts 3	4.63	8.21	4.83
Sampled Pts 4	4.67	7.89	4.86
Sampled Pts 5	4.68	8.22	4.87
Subblock Full Search	6.00	3.67	5.96
TSS	6.15	3.62	6.10
Tile search	5.00	6.16	5.07
Variance	5.33	9.35	5.56

Table B.15: Comparison of Algorithms for the sequence **cross_country**

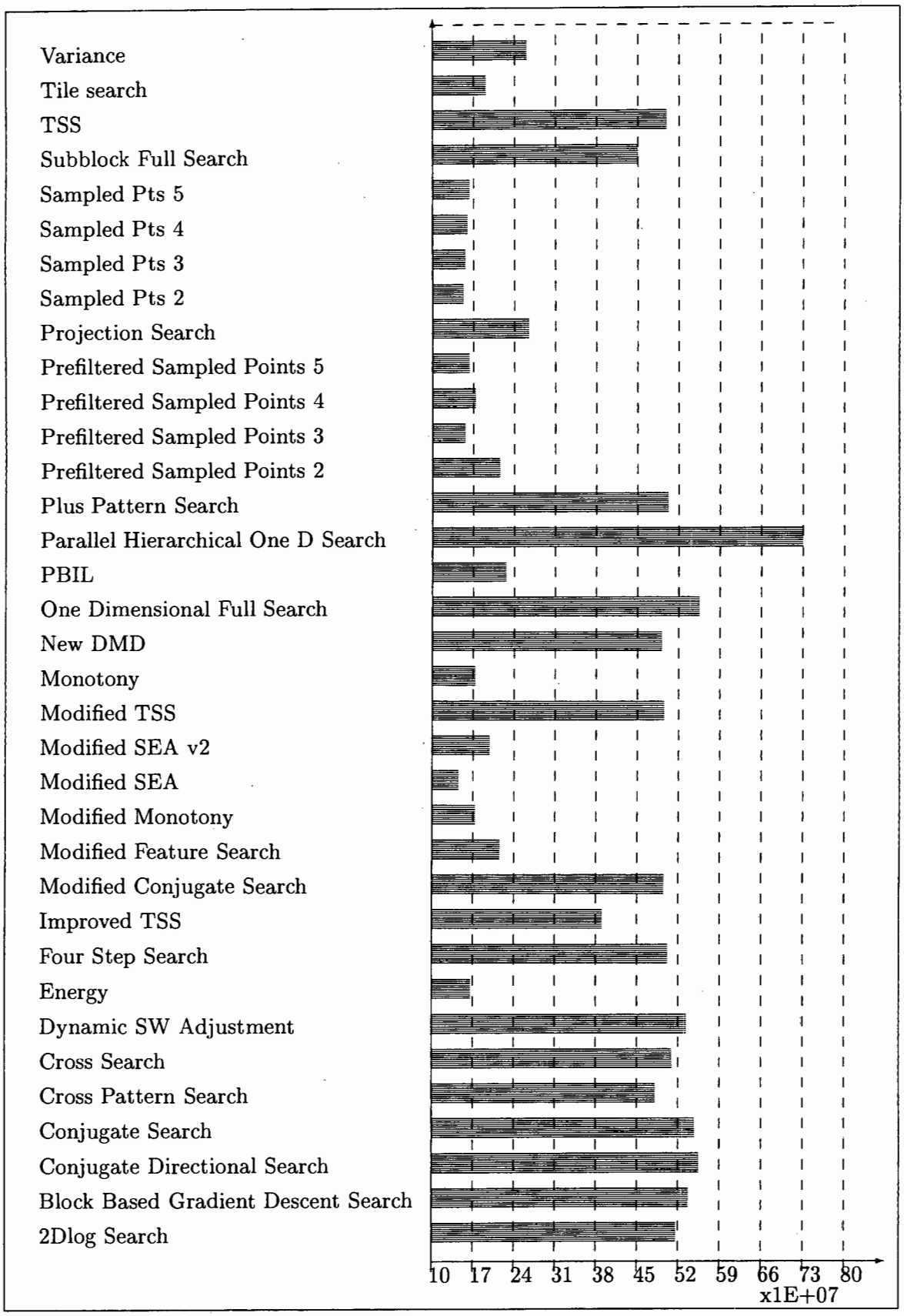


Figure B.37: **DFD** for sequence **cross_country**

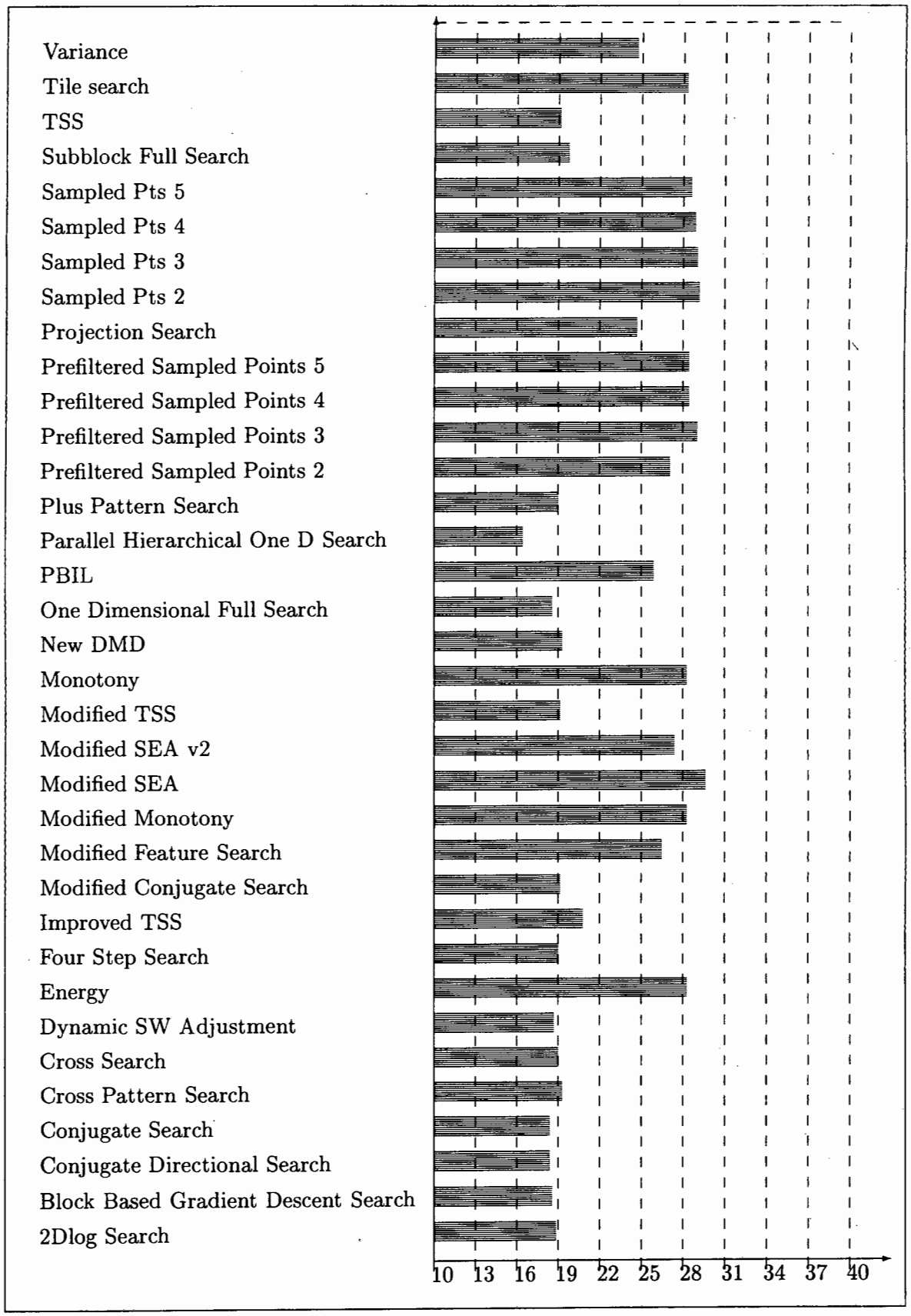


Figure B.38: **PSNR** for sequence **cross_country**

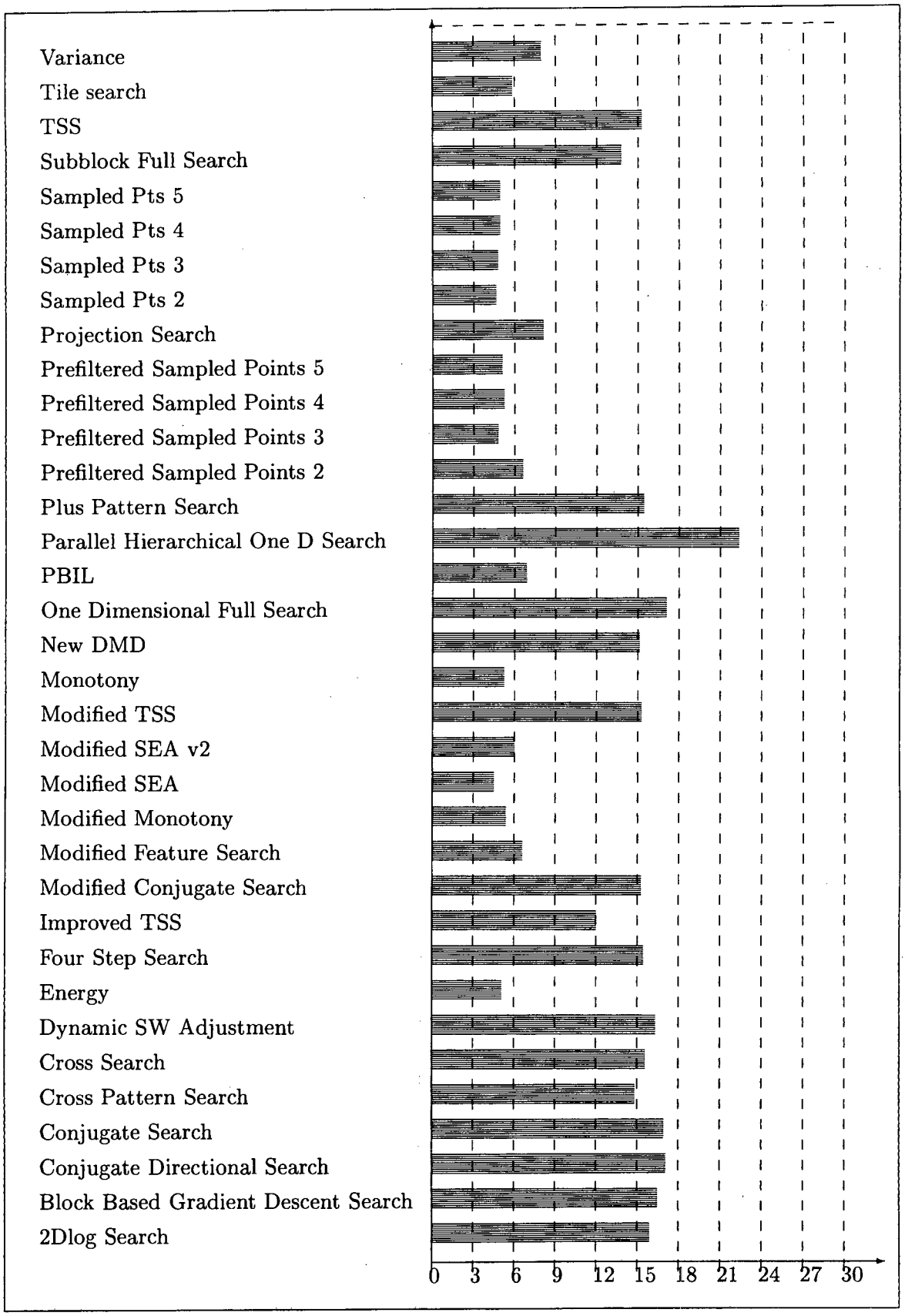


Figure B.39: **Di abs mean** for sequence **cross_country**

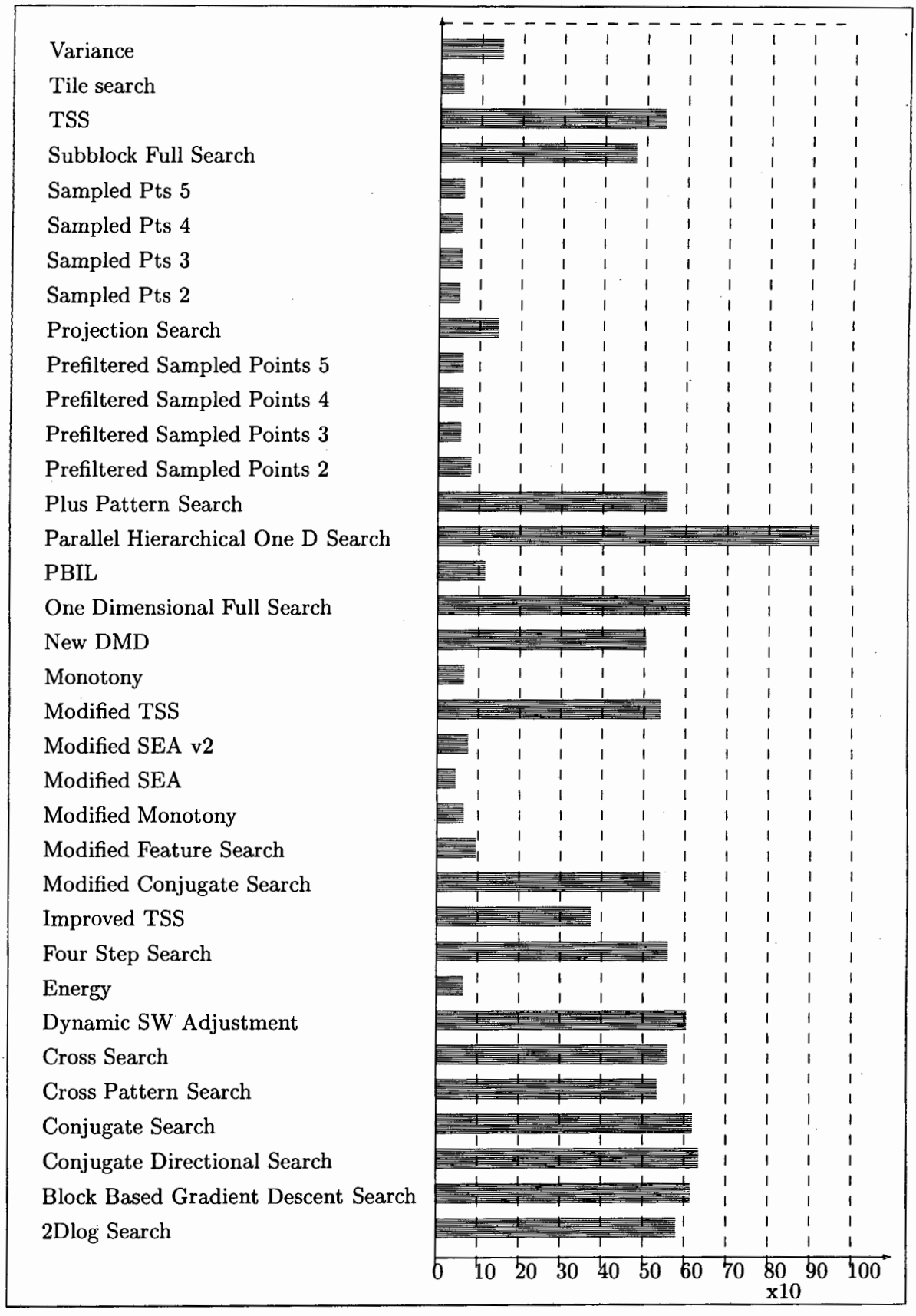


Figure B.40: **Di abs var** for sequence **cross_country**

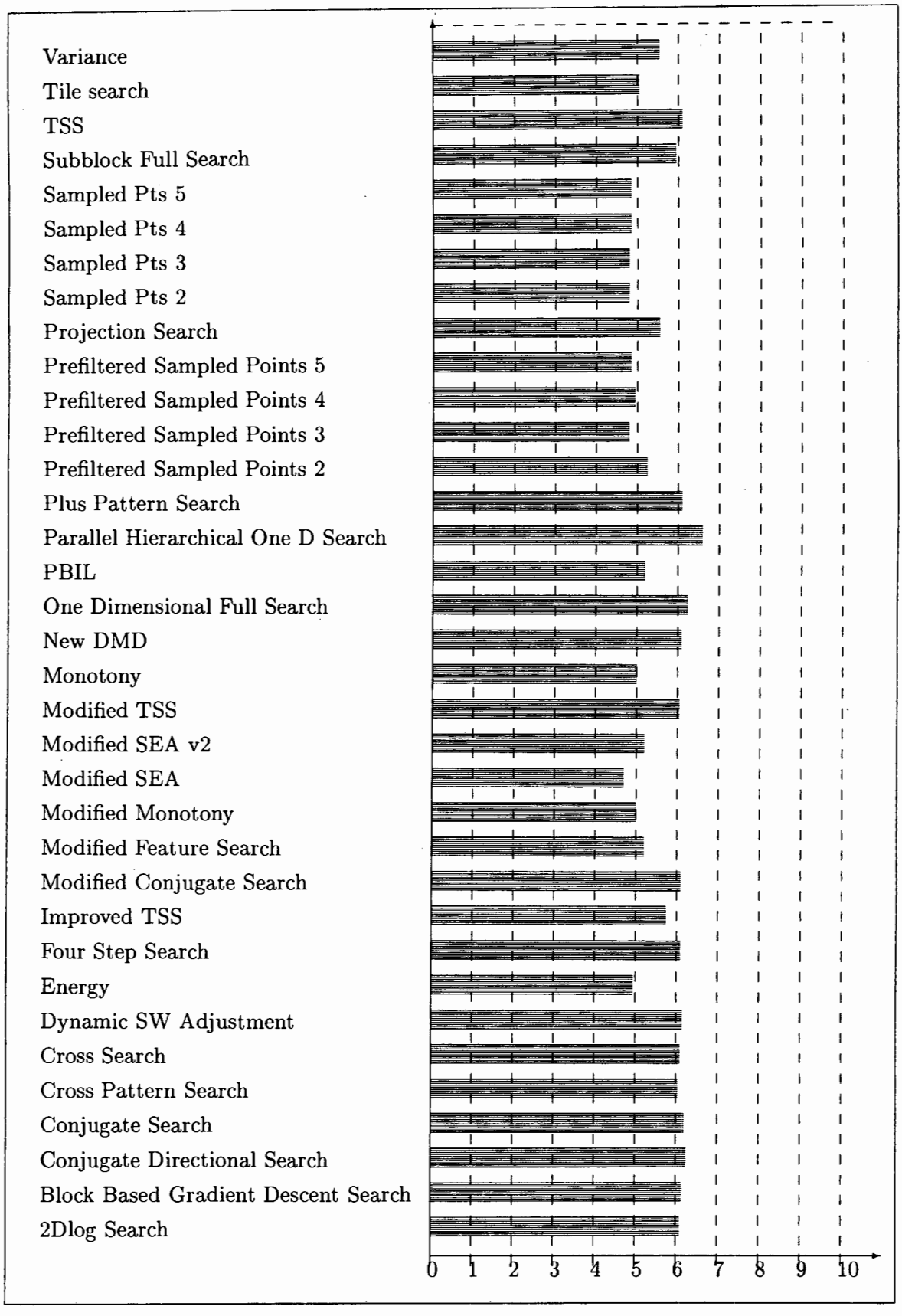


Figure B.41: **Entropy cumulative** for sequence **cross_country**

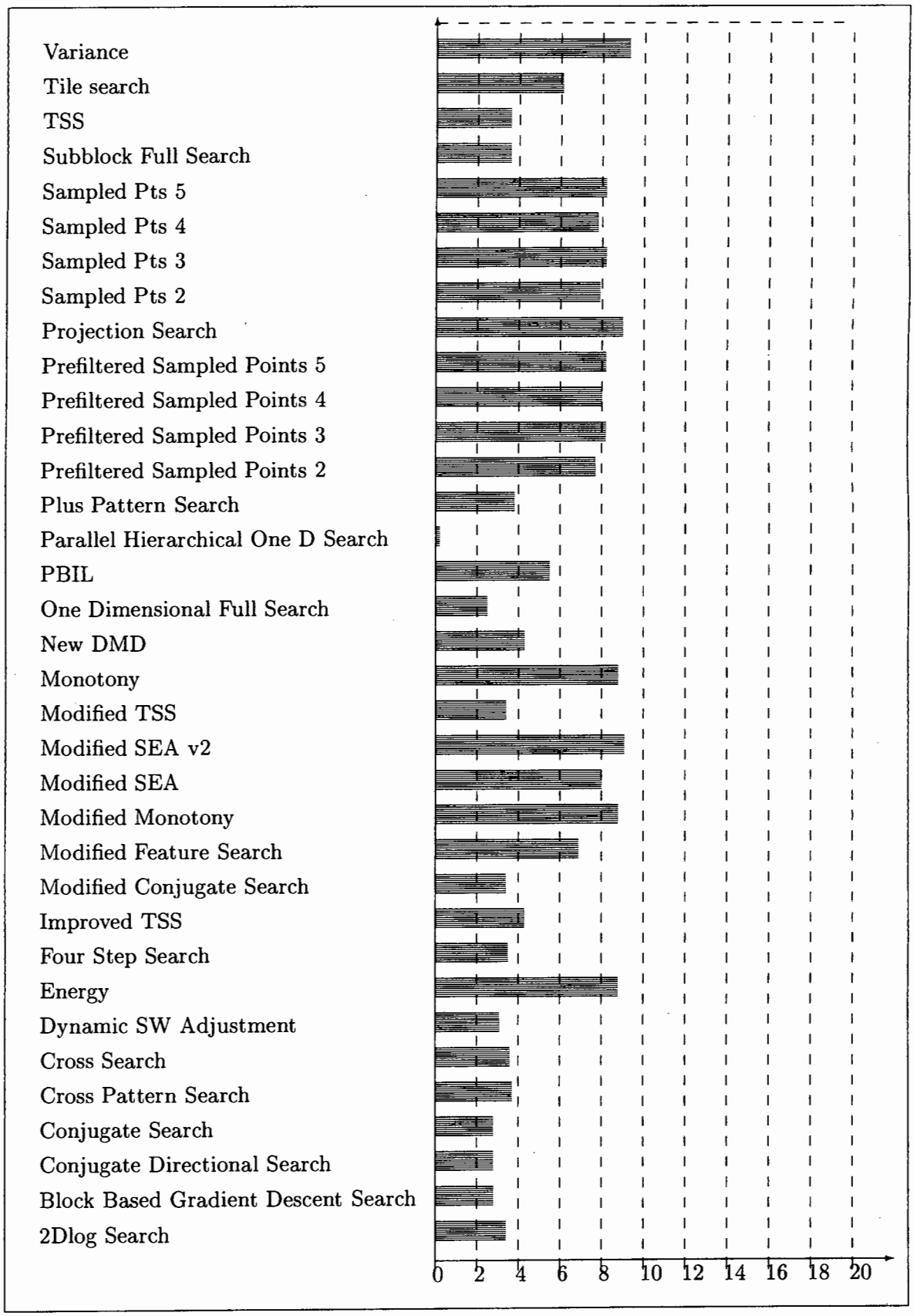


Figure B.42: **Entropy mtn vec** for sequence **cross_country**

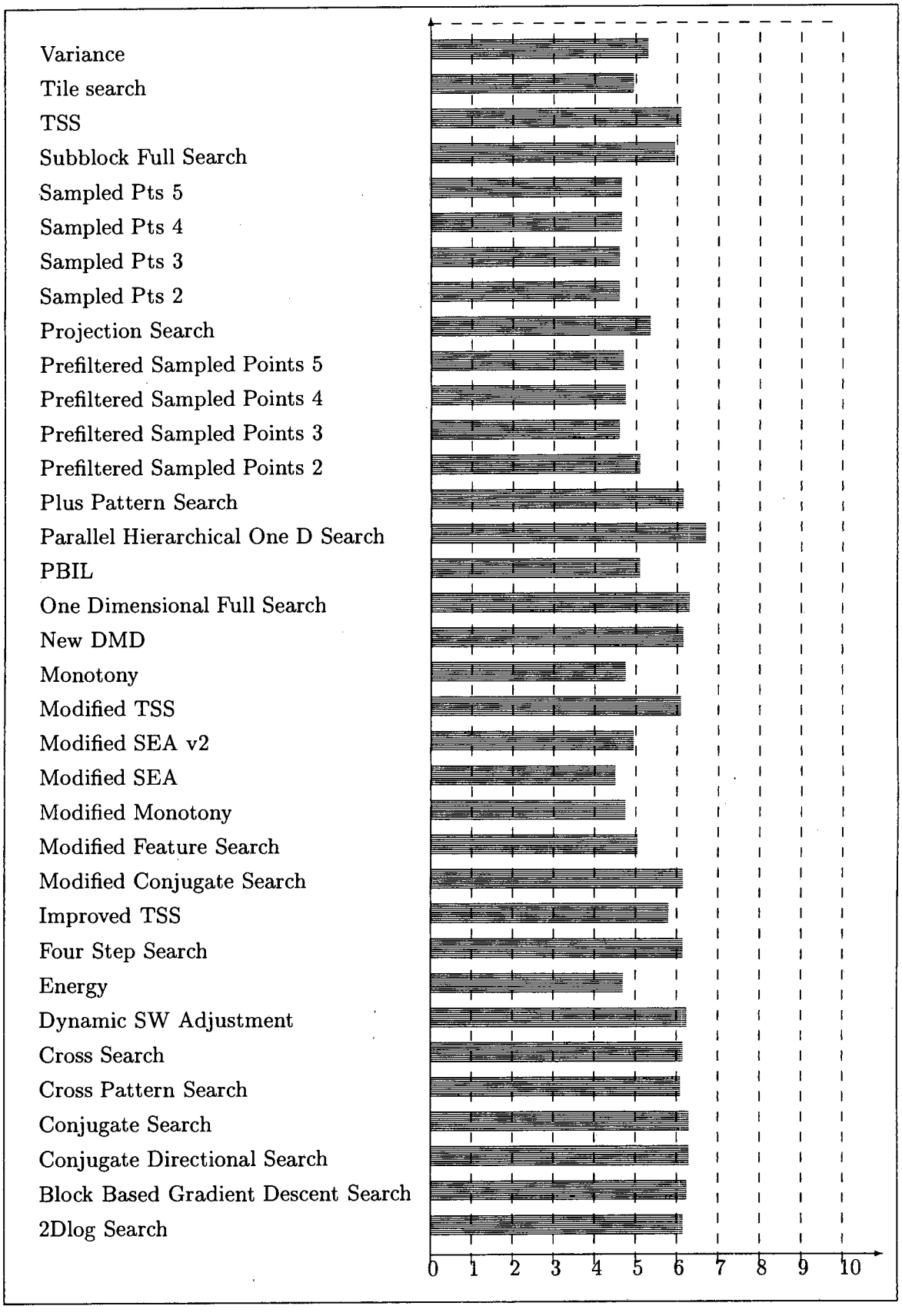


Figure B.43: **Entropy pixel** for sequence **cross_country**

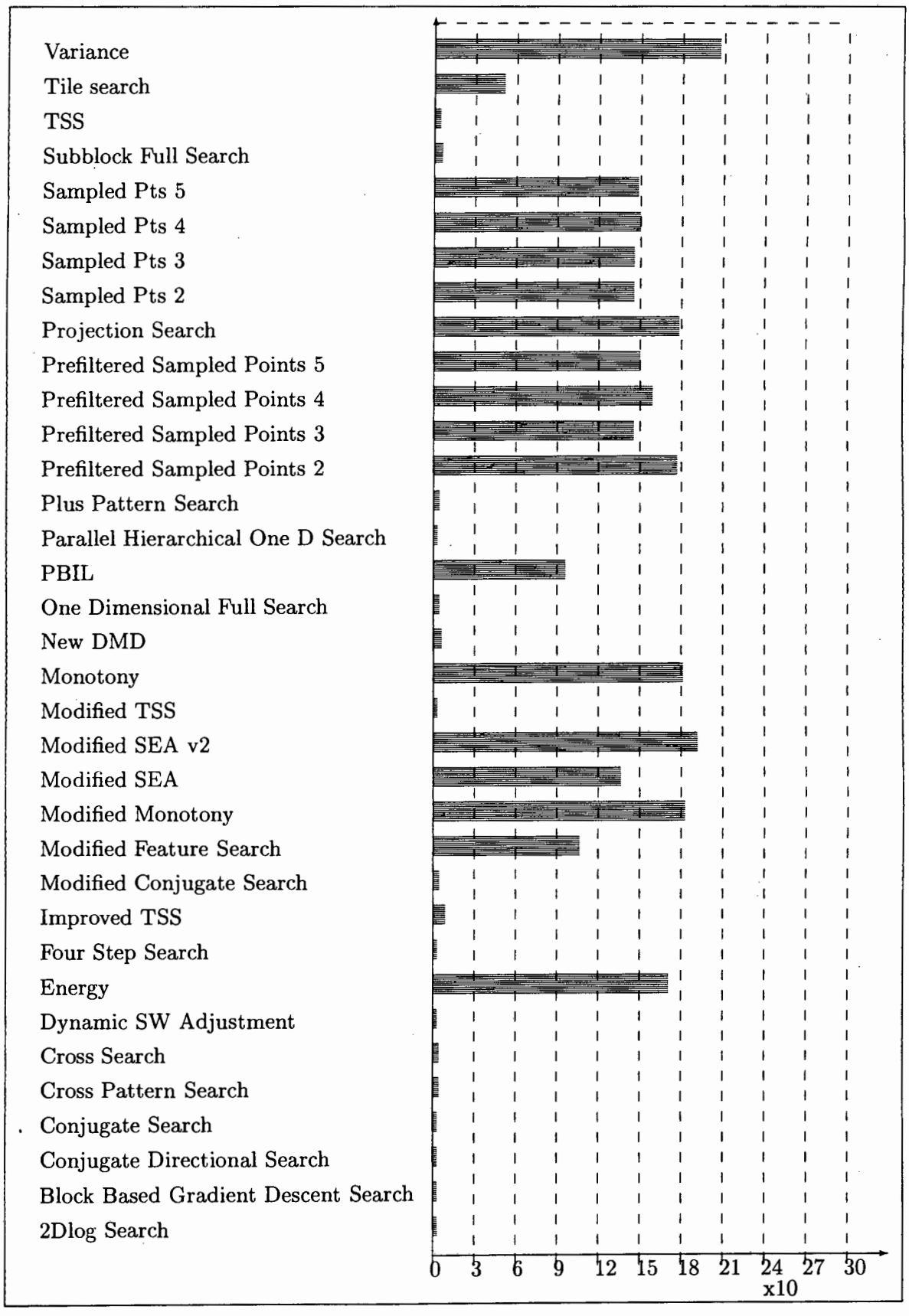


Figure B.44: **Motion Vector Means** for sequence **cross_country**

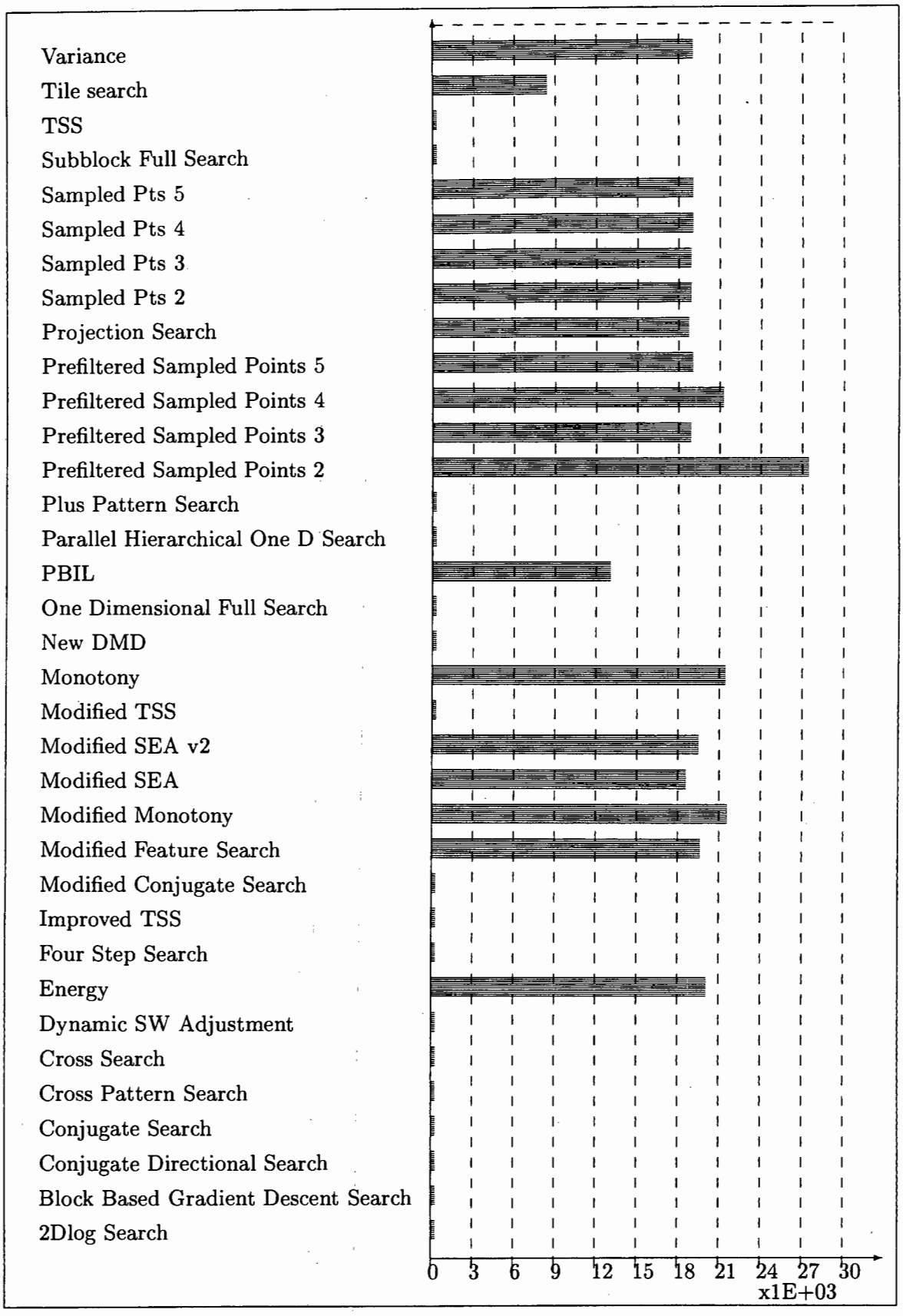


Figure B.45: **Motion Vector Var** for sequence **cross_country**

Algorithm	Difference block			
			Absolute	
	Mean	Variance	Mean	Variance
2Dlog Search	-0.04	701.7	15.2	470.5
Block Based Gradient Descent Search	-0.07	865.1	16.7	584.6
Conjugate Directional Search	-0.10	930.7	17.6	622.0
Conjugate Search	-0.11	931.0	17.6	621.6
Cross Pattern Search	-0.04	674.9	15.0	450.0
Cross Search	-0.13	784.1	16.3	518.9
Dynamic SW Adjustment	-0.10	767.3	15.9	514.8
Energy	-0.99	428.1	12.4	276.6
Four Step Search	-0.03	660.9	14.8	442.2
Improved TSS	-0.06	490.6	12.8	326.1
Modified Conjugate Search	-0.06	732.5	15.7	486.9
Modified Feature Search	-0.06	516.2	13.4	336.3
Modified Monotony	0.09	429.8	12.5	273.1
Modified SEA	0.07	320.5	10.7	206.4
Modified SEA v2	0.58	512.9	13.7	325.3
Modified TSS	-0.05	645.1	14.7	430.2
Monotony	0.09	427.0	12.5	271.4
New DMD	-0.16	761.0	16.2	500.0
One Dimensional Full Search	-0.03	1421.6	22.1	934.9
PBIL	-0.12	662.4	15.2	432.5
Parallel Hierarchical One D Search	-0.01	1874.4	25.9	1204.8
Plus Pattern Search	-0.07	772.5	15.9	519.0
Prefiltered Sampled Points 2	1.54	433.0	12.9	268.1
Prefiltered Sampled Points 3	0.11	403.3	12.0	260.4
Prefiltered Sampled Points 4	0.69	432.8	12.6	275.7
Prefiltered Sampled Points 5	0.12	451.4	12.6	292.2
Projection Search	-0.01	883.2	18.1	553.9
Sampled Pts 2	0.09	356.3	11.2	230.2
Sampled Pts 3	0.12	401.5	11.9	259.4
Sampled Pts 4	0.14	422.6	12.2	273.4
Sampled Pts 5	0.14	444.6	12.5	287.8
Subblock Full Search	-0.07	508.6	13.0	338.8
TSS	-0.09	658.8	14.8	439.1
Tile search	0.07	355.1	11.5	222.3
Variance	0.07	780.7	17.0	492.1

Table B.16: Comparison of Algorithms for the sequence **cycling**

Algorithm	Motion Vector		PSNR	DFD
	Mean	Variance		(x 10^8)
2Dlog Search	3.117	3.751	19.67	4.943
Block Based Gradient Descent Search	2.060	3.220	18.76	5.444
Conjugate Directional Search	2.575	4.029	18.44	5.711
Conjugate Search	2.537	3.771	18.44	5.718
Cross Pattern Search	5.098	7.008	19.84	4.875
Cross Search	4.411	6.890	19.19	5.294
Dynamic SW Adjustment	2.907	2.051	19.28	5.165
Energy	149.507	19081	21.81	4.015
Four Step Search	3.476	4.463	19.93	4.808
Improved TSS	6.765	22.201	21.22	4.169
Modified Conjugate Search	4.655	7.142	19.48	5.094
Modified Feature Search	49.672	11172	21.00	4.360
Modified Monotony	173.715	21545	21.80	4.070
Modified SEA	111.942	16530	23.07	3.471
Modified SEA v2	183.989	18882	21.03	4.457
Modified TSS	3.465	5.370	20.03	4.765
Monotony	171.261	21264	21.83	4.055
New DMD	5.508	8.902	19.32	5.252
One Dimensional Full Search	2.321	10.982	16.60	7.171
PBIL	78.245	11300	19.92	4.929
Parallel Hierarchical One D Search	0.001	0.005	15.40	8.412
Plus Pattern Search	5.343	6.661	19.25	5.176
Prefiltered Sampled Points 2	141.183	23225	21.77	4.203
Prefiltered Sampled Points 3	143.160	17541	22.07	3.885
Prefiltered Sampled Points 4	146.572	18293	21.77	4.080
Prefiltered Sampled Points 5	155.198	17680	21.59	4.101
Projection Search	181.956	17162	18.67	5.899
Sampled Pts 2	121.595	16976	22.61	3.650
Sampled Pts 3	143.040	17611	22.09	3.875
Sampled Pts 4	147.437	17424	21.87	3.970
Sampled Pts 5	154.165	17672	21.65	4.072
Subblock Full Search	5.298	8.682	21.07	4.237
TSS	4.384	5.224	19.94	4.819
Tile search	85.319	15991	22.63	3.747
Variance	199.511	18131	19.21	5.523

Table B.17: Comparison of Algorithms for the sequence **cycling**

Algorithm	Entropy		
	Pixel	Motion Vector	Cumulative
2Dlog Search	6.28	3.33	6.23
Block Based Gradient Descent Search	6.40	2.83	6.35
Conjugate Directional Search	6.48	2.95	6.42
Conjugate Search	6.48	2.94	6.42
Cross Pattern Search	6.26	3.91	6.22
Cross Search	6.38	3.75	6.33
Dynamic SW Adjustment	6.34	3.18	6.28
Energy	6.00	8.55	6.13
Four Step Search	6.24	3.48	6.19
Improved TSS	6.04	4.47	6.01
Modified Conjugate Search	6.32	3.65	6.28
Modified Feature Search	6.12	5.27	6.13
Modified Monotony	6.03	8.82	6.18
Modified SEA	5.80	7.84	5.90
Modified SEA v2	6.16	9.07	6.32
Modified TSS	6.23	3.46	6.18
Monotony	6.03	8.80	6.17
New DMD	6.37	4.11	6.33
One Dimensional Full Search	6.80	1.96	6.73
PBIL	6.28	5.54	6.32
Parallel Hierarchical One D Search	7.03	0.00	6.95
Plus Pattern Search	6.34	3.94	6.30
Prefiltered Sampled Points 2	6.09	7.51	6.20
Prefiltered Sampled Points 3	5.96	8.52	6.09
Prefiltered Sampled Points 4	6.03	8.15	6.16
Prefiltered Sampled Points 5	5.96	8.60	6.09
Projection Search	6.55	9.14	6.69
Sampled Pts 2	5.87	7.70	5.98
Sampled Pts 3	5.95	8.51	6.08
Sampled Pts 4	5.98	8.01	6.11
Sampled Pts 5	6.02	8.70	6.16
Subblock Full Search	6.07	3.95	6.03
TSS	6.24	3.71	6.20
Tile search	5.93	6.60	5.99
Variance	6.46	9.30	6.61

Table B.18: Comparison of Algorithms for the sequence **cycling**

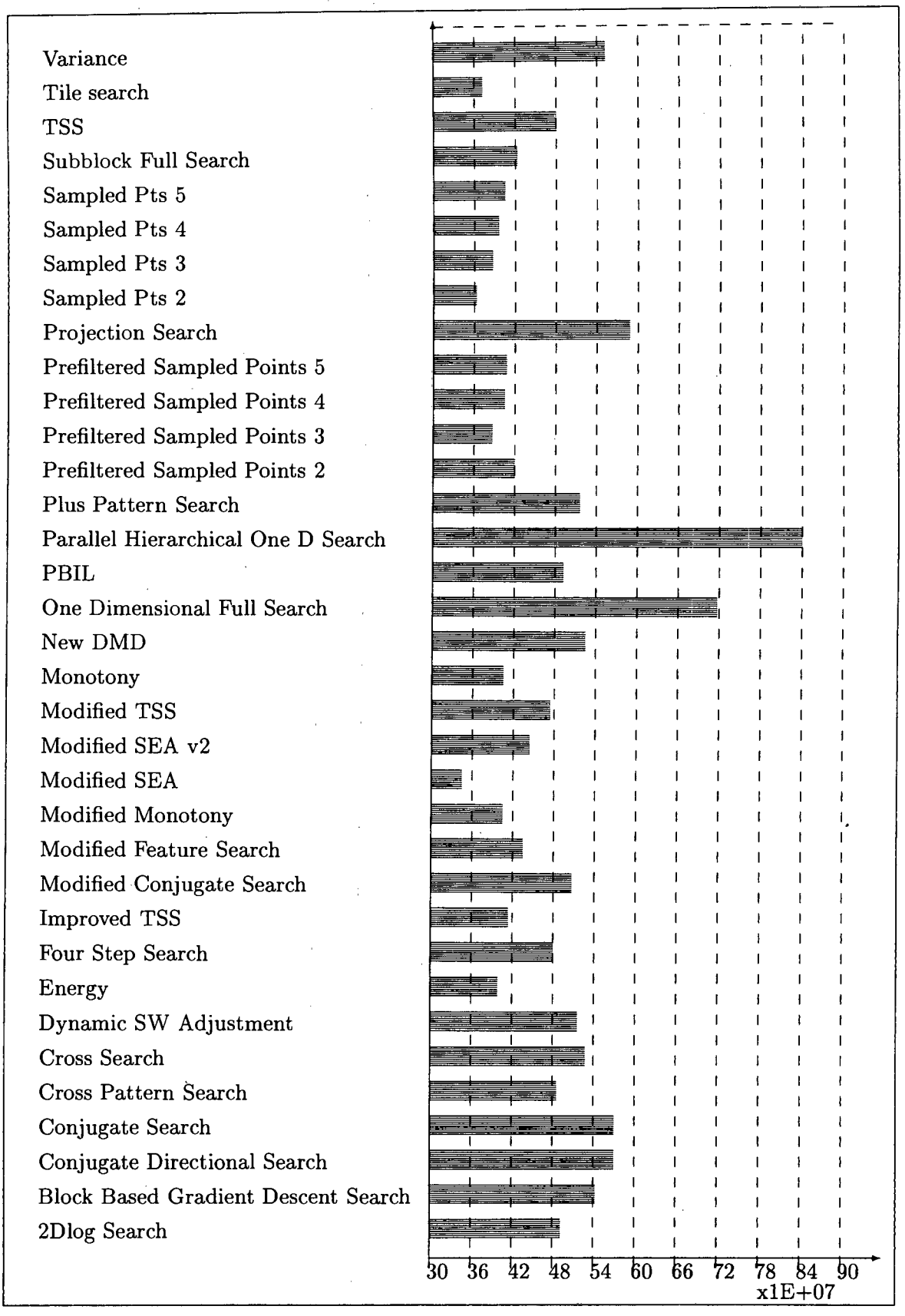


Figure B.46: **DFD** for sequence **cycling**

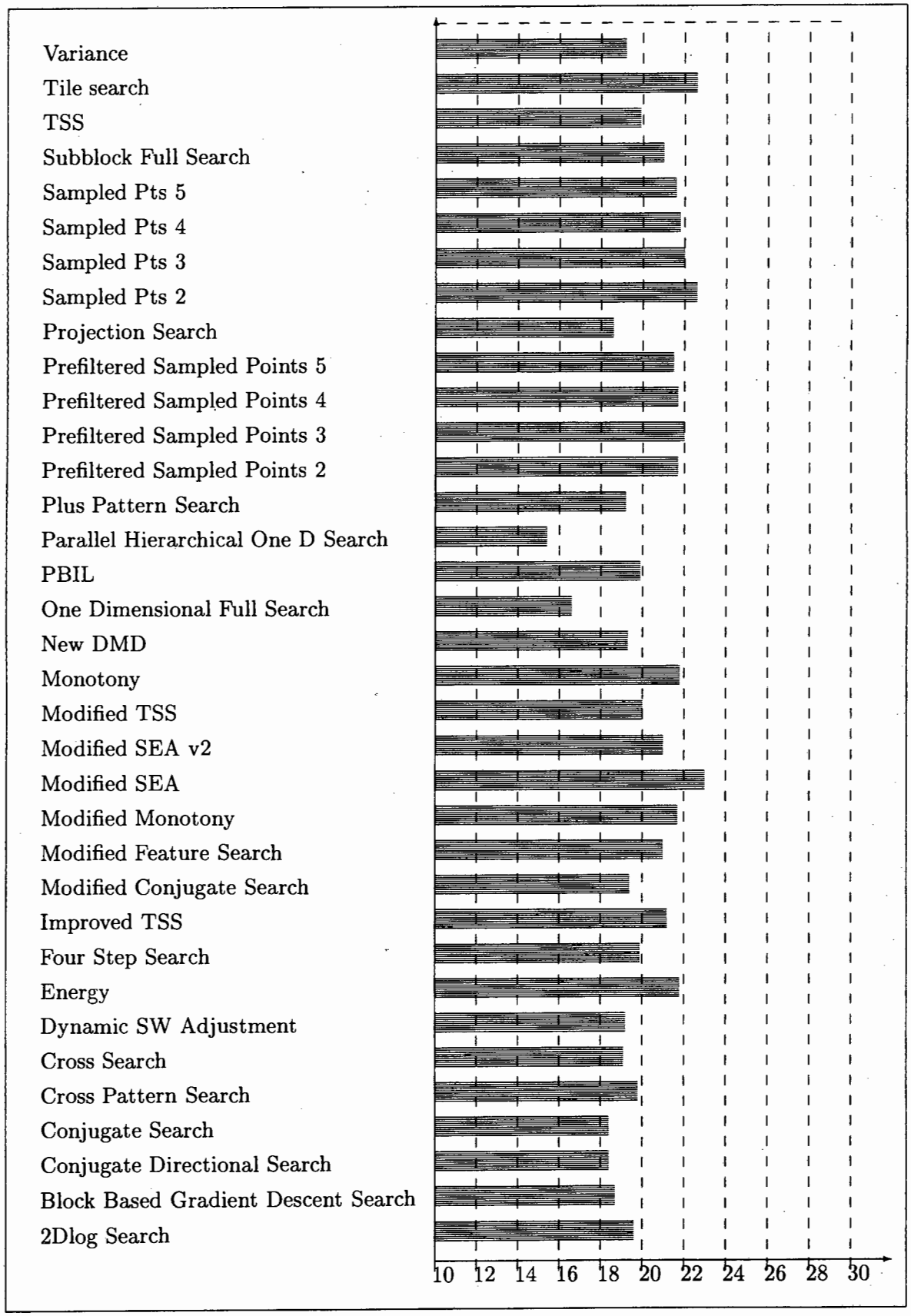


Figure B.47: **PSNR** for sequence **cycling**

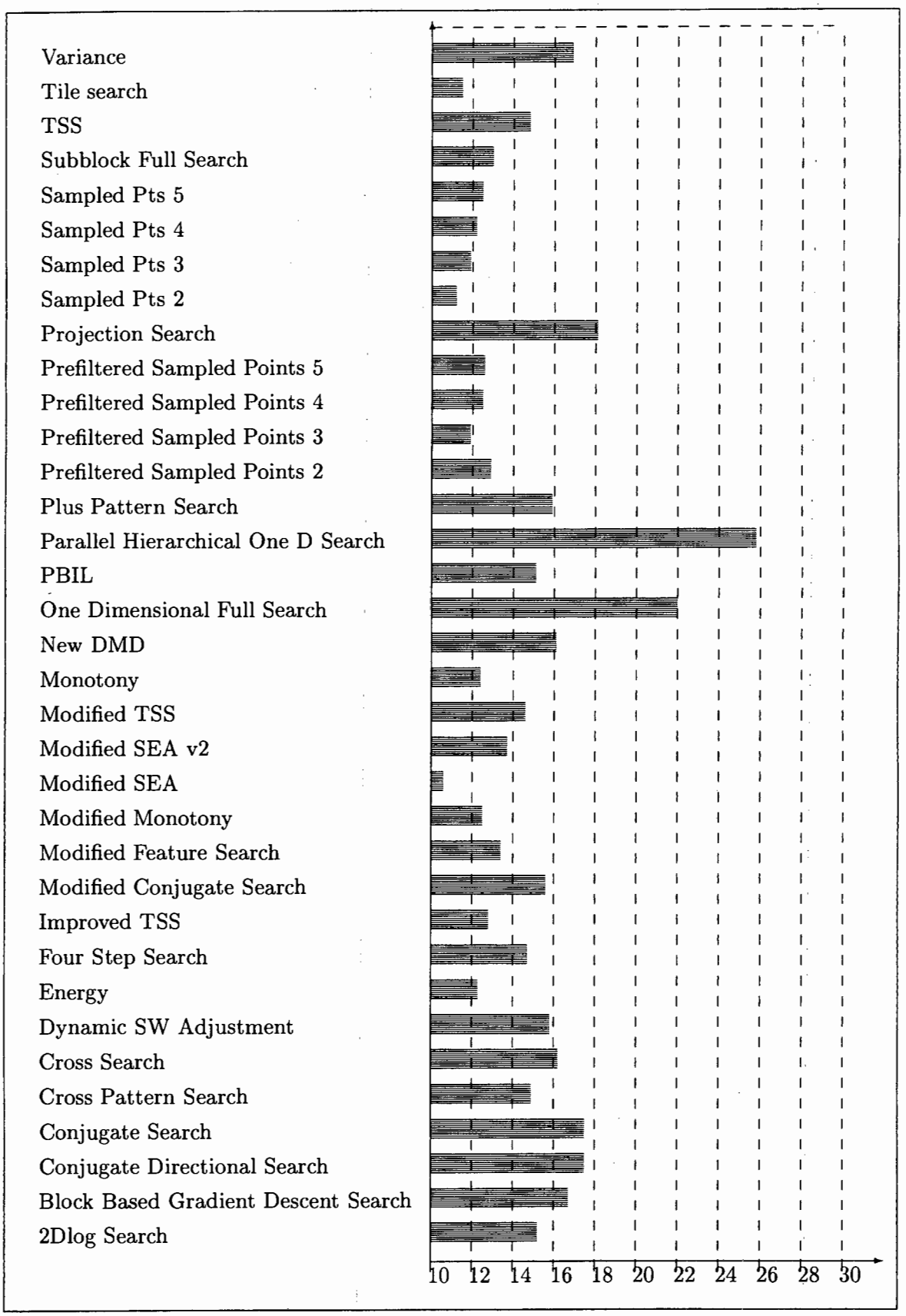


Figure B.48: **Di abs mean** for sequence **cycling**

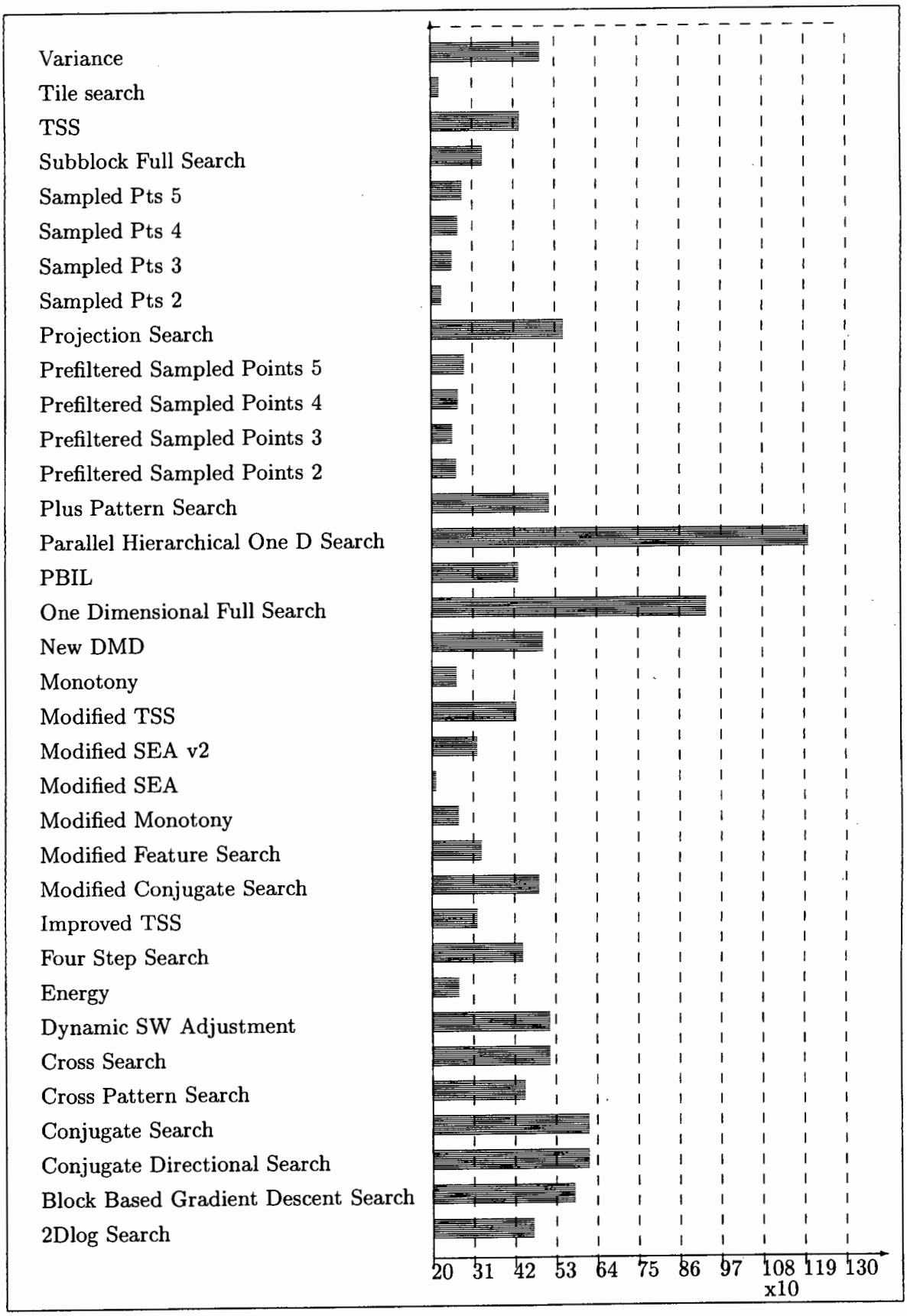


Figure B.49: **Di abs var** for sequence **cycling**

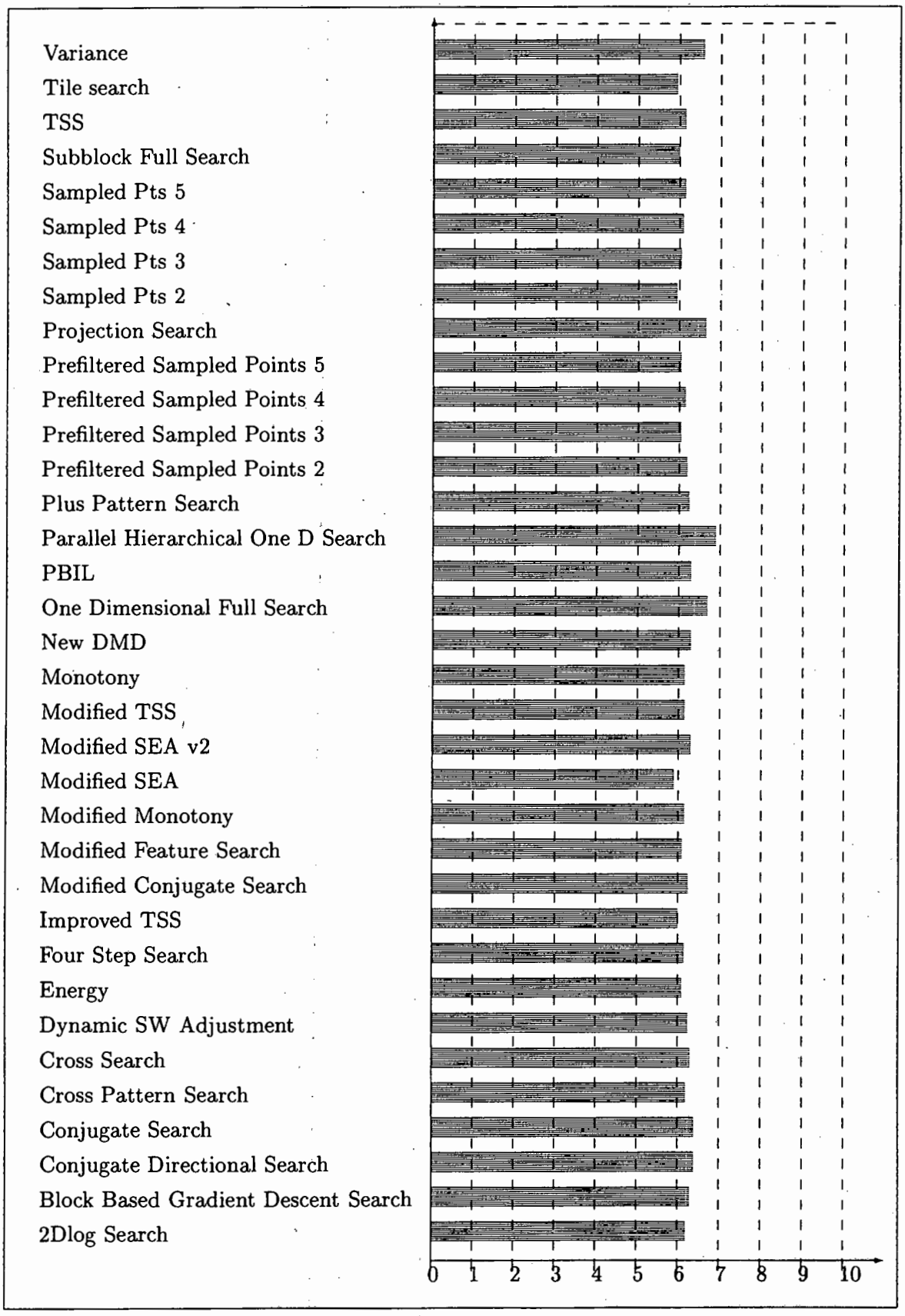


Figure B.50: **Entropy cumulative** for sequence **cycling**

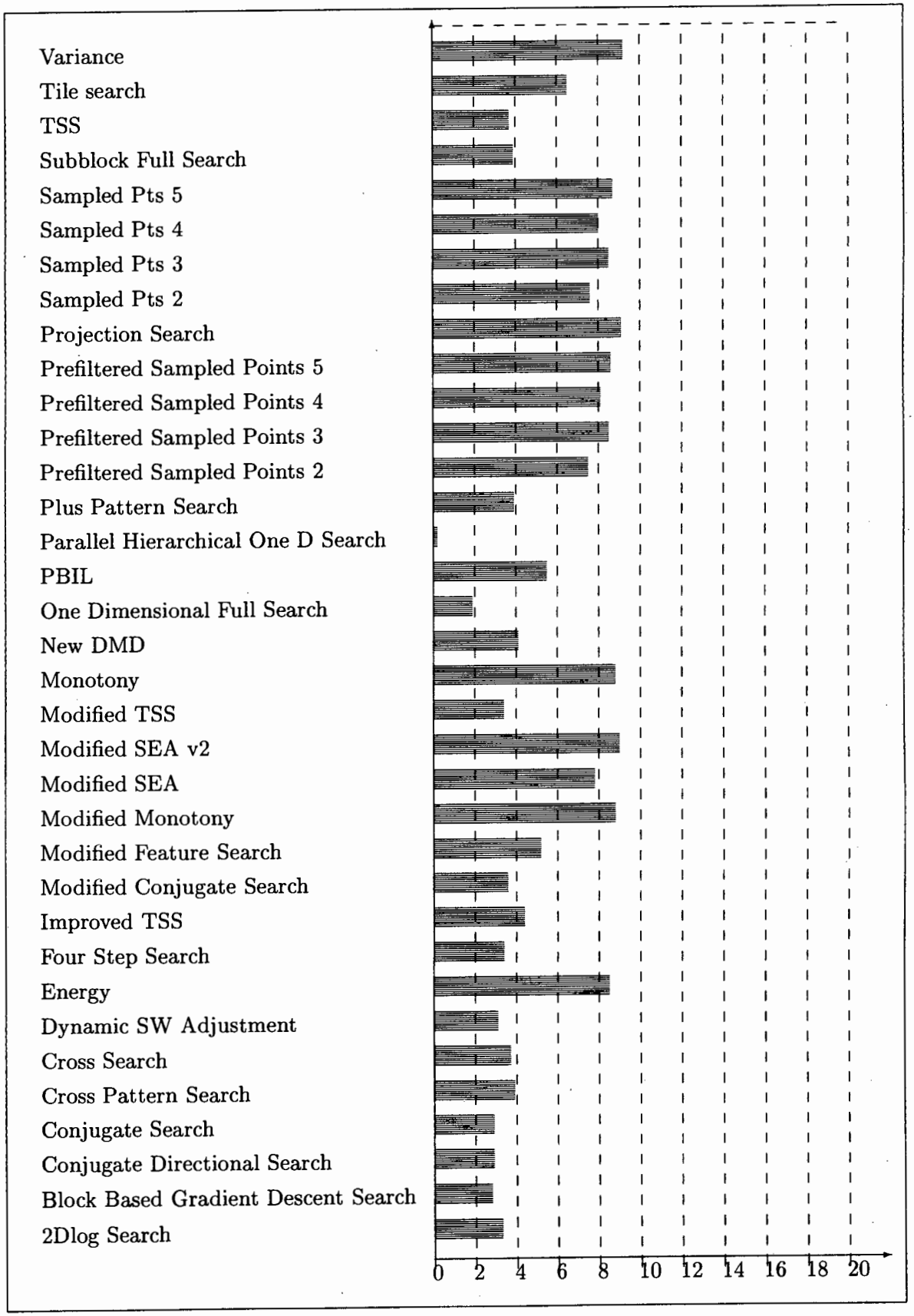


Figure B.51: **Entropy mtn vec** for sequence **cycling**

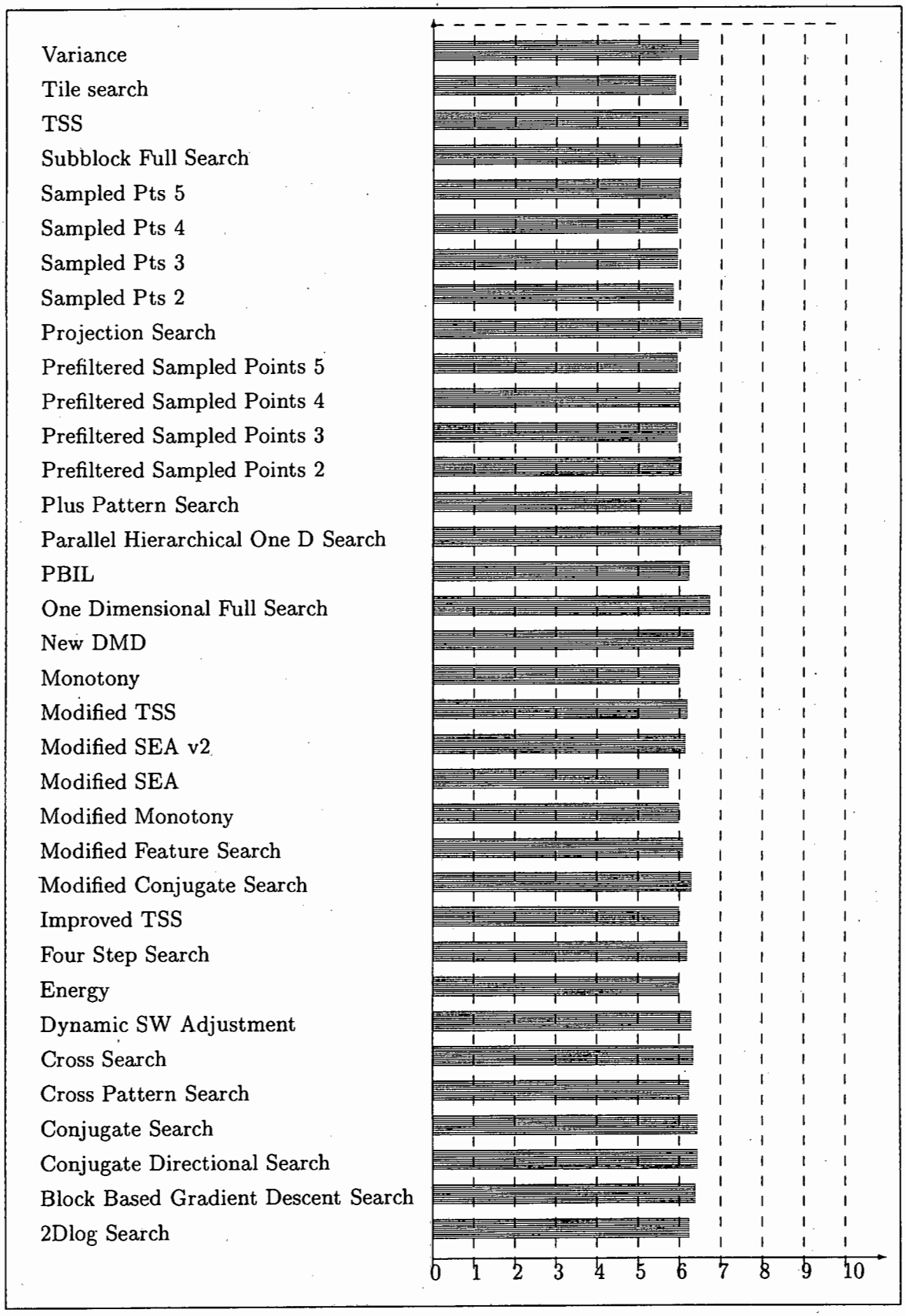


Figure B.52: **Entropy pixel** for sequence **cycling**

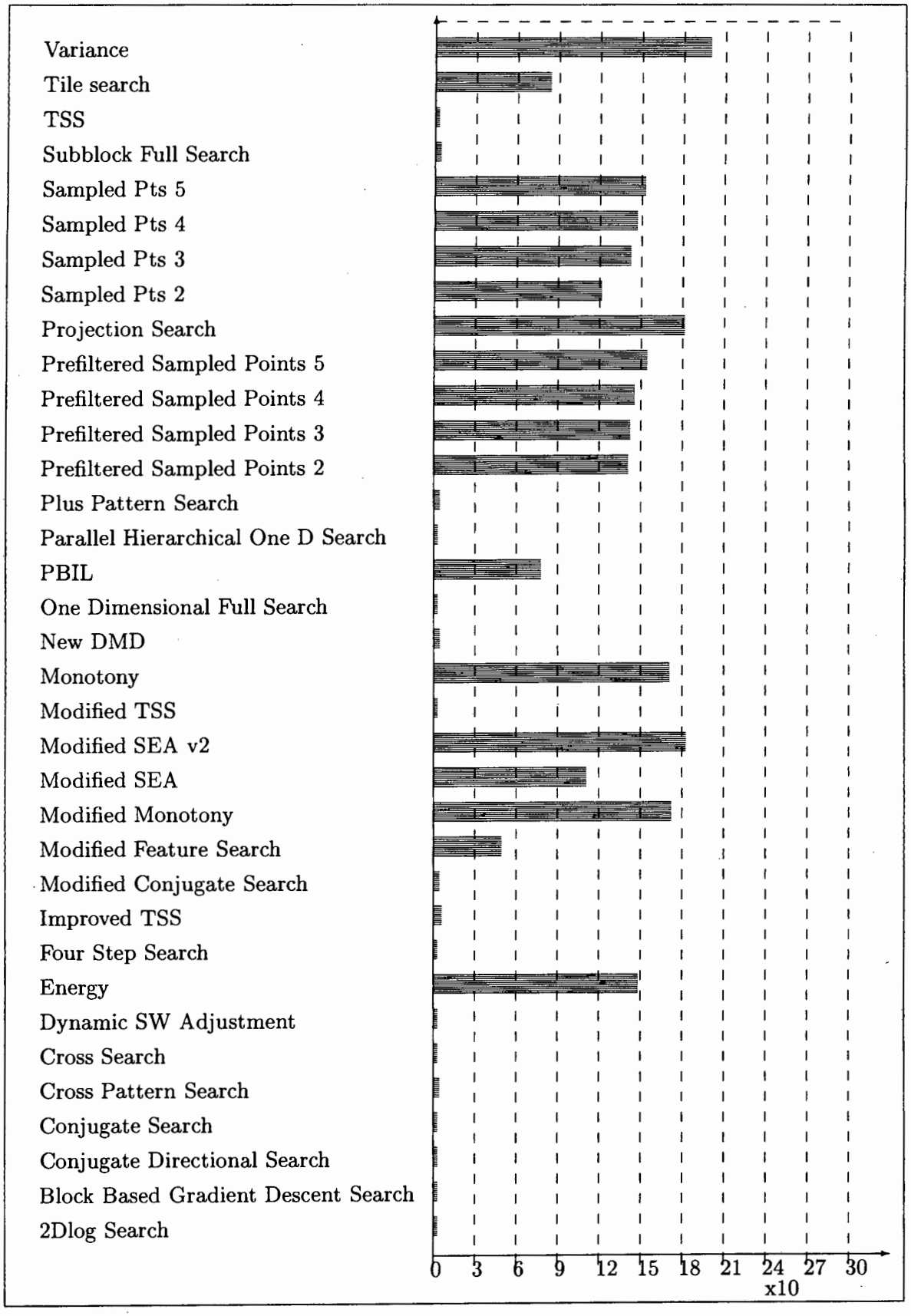


Figure B.53: **Motion Vector Means** for sequence **cycling**

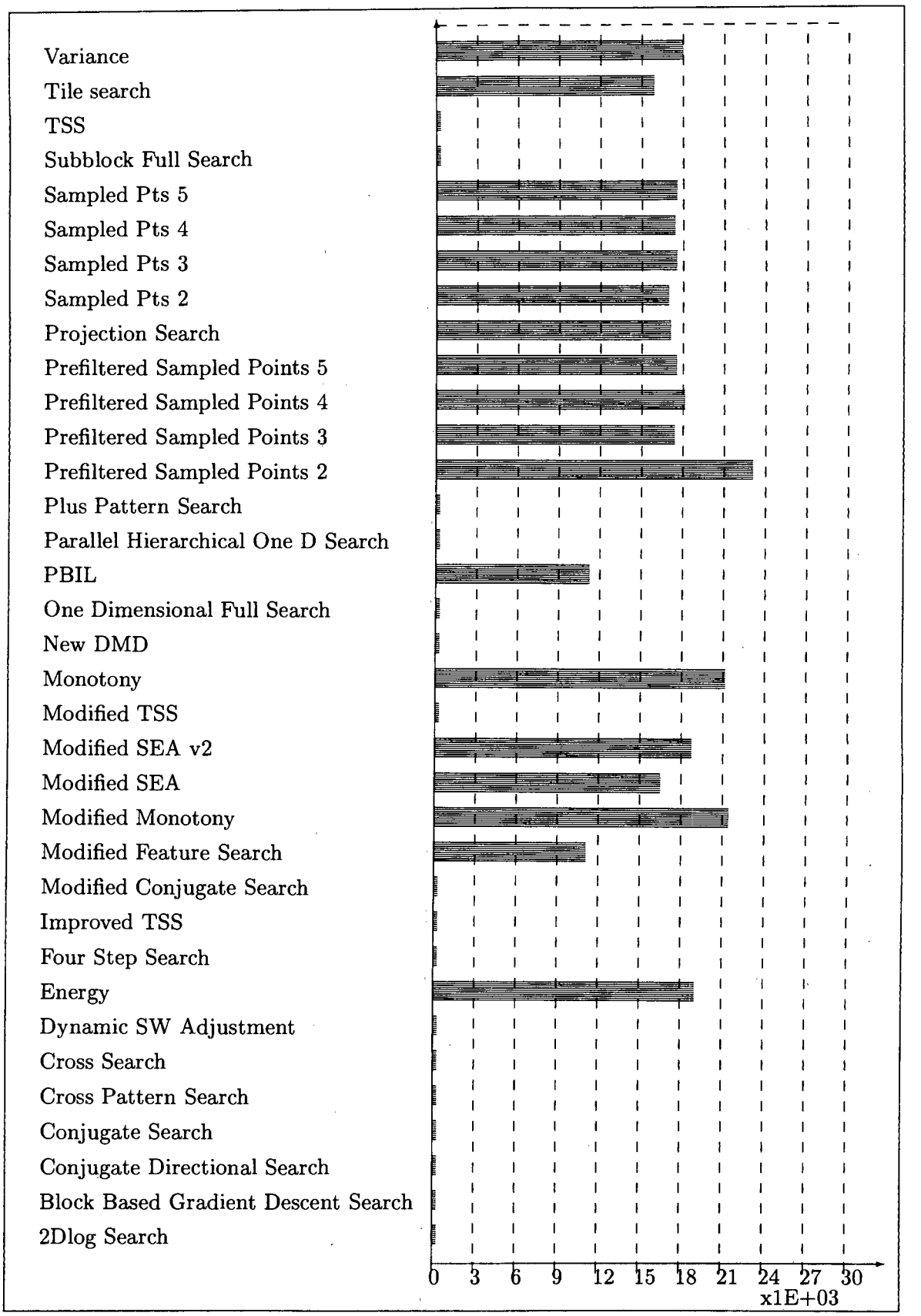


Figure B.54: **Motion Vector Var** for sequence **cycling**

Algorithm	Difference block			
			Absolute	
	Mean	Variance	Mean	Variance
2Dlog Search	-0.88	1965.3	21.6	1497.7
Block Based Gradient Descent Search	-0.43	3903.5	32.5	2849.3
Conjugate Directional Search	-0.79	2644.8	26.4	1946.7
Conjugate Search	-0.88	2692.9	26.8	1975.2
Cross Pattern Search	0.04	1261.8	16.5	989.4
Cross Search	-0.48	1429.7	18.3	1094.5
Dynamic SW Adjustment	-0.52	2431.2	25.2	1798.5
Energy	-0.18	159.4	6.5	117.8
Four Step Search	-0.74	1564.5	18.8	1212.0
Improved TSS	0.21	626.7	11.1	504.3
Modified Conjugate Search	-0.52	1460.3	18.5	1117.7
Modified Feature Search	-0.02	396.5	9.5	305.4
Modified Monotony	0.08	179.2	6.9	132.1
Modified SEA	0.09	115.2	5.4	86.3
Modified SEA v2	0.11	598.6	12.5	443.2
Modified TSS	-0.40	1388.0	17.8	1069.8
Monotony	0.08	179.8	6.9	132.5
New DMD	-0.66	1100.5	15.7	855.6
One Dimensional Full Search	-0.54	3024.5	28.9	2190.1
PBIL	-0.14	499.4	11.0	377.9
Parallel Hierarchical One D Search	-0.02	5869.1	45.8	3771.1
Plus Pattern Search	0.04	1354.4	17.6	1044.8
Prefiltered Sampled Points 2	-0.55	800.0	15.0	576.6
Prefiltered Sampled Points 3	4.64	1174.8	18.6	850.5
Prefiltered Sampled Points 4	0.94	1266.5	17.7	952.5
Prefiltered Sampled Points 5	-1.07	1005.3	16.0	750.2
Projection Search	-0.16	1213.0	18.2	883.0
Sampled Pts 2	0.08	127.8	5.7	95.4
Sampled Pts 3	0.06	162.0	6.5	120.3
Sampled Pts 4	0.05	159.8	6.4	119.0
Sampled Pts 5	0.04	199.0	7.2	147.8
Subblock Full Search	0.23	821.3	12.8	657.1
TSS	-0.48	1354.7	17.5	1050.2
Tile search	0.01	151.3	6.6	107.3
Variance	0.07	591.6	12.6	432.2

Table B.19: Comparison of Algorithms for the sequence **rot_disc3**

Algorithm	Motion Vector		PSNR	DFD
	Mean	Variance		(x 10^8)
2Dlog Search	4.680	5.615	15.20	7.035
Block Based Gradient Descent Search	2.497	6.855	12.22	10.555
Conjugate Directional Search	3.942	9.575	13.91	8.592
Conjugate Search	3.747	7.999	13.83	8.713
Cross Pattern Search	6.967	5.068	17.12	5.365
Cross Search	6.281	7.091	16.58	5.953
Dynamic SW Adjustment	3.748	2.911	14.27	8.178
Energy	103.754	15015	26.11	2.097
Four Step Search	5.541	7.483	16.19	6.107
Improved TSS	10.507	17.782	20.16	3.597
Modified Conjugate Search	6.487	7.290	16.49	6.020
Modified Feature Search	45.193	9646	22.15	3.104
Modified Monotony	113.939	15664	25.60	2.231
Modified SEA	75.425	10817	27.52	1.748
Modified SEA v2	189.296	21368	20.36	4.052
Modified TSS	5.366	9.036	16.71	5.799
Monotony	113.780	15602	25.58	2.236
New DMD	7.593	10.858	17.72	5.091
One Dimensional Full Search	4.517	21.522	13.32	9.391
PBIL	72.513	8683	21.15	3.582
Parallel Hierarchical One D Search	0.001	0.011	10.45	14.889
Plus Pattern Search	7.027	4.623	16.81	5.719
Prefiltered Sampled Points 2	215.347	32254	19.10	4.862
Prefiltered Sampled Points 3	151.457	27361	17.43	6.045
Prefiltered Sampled Points 4	129.254	23923	17.10	5.768
Prefiltered Sampled Points 5	127.426	21360	18.11	5.203
Projection Search	179.886	18759	17.29	5.906
Sampled Pts 2	78.135	10903	27.07	1.850
Sampled Pts 3	96.632	12512	26.04	2.099
Sampled Pts 4	93.319	12066	26.09	2.078
Sampled Pts 5	111.795	13592	25.14	2.325
Subblock Full Search	8.162	6.260	18.99	4.166
TSS	6.280	6.056	16.81	5.674
Tile search	33.279	4557	26.33	2.155
Variance	169.555	17734	20.41	4.104

Table B.20: Comparison of Algorithms for the sequence **rot_disc3**

Algorithm	Entropy		
	Pixel	Motion Vector	Cumulative
2Dlog Search	6.37	3.57	6.31
Block Based Gradient Descent Search	6.83	2.93	6.75
Conjugate Directional Search	6.66	3.05	6.60
Conjugate Search	6.68	3.07	6.62
Cross Pattern Search	6.04	3.64	6.01
Cross Search	6.20	3.55	6.16
Dynamic SW Adjustment	6.61	3.35	6.54
Energy	5.00	7.97	5.16
Four Step Search	6.19	3.67	6.14
Improved TSS	5.54	4.42	5.54
Modified Conjugate Search	6.22	3.52	6.18
Modified Feature Search	5.45	4.79	5.49
Modified Monotony	5.09	8.24	5.25
Modified SEA	4.76	7.27	4.90
Modified SEA v2	5.88	9.20	6.05
Modified TSS	6.15	3.60	6.11
Monotony	5.10	8.25	5.26
New DMD	6.02	4.32	5.99
One Dimensional Full Search	6.80	2.55	6.73
PBIL	5.67	5.84	5.73
Parallel Hierarchical One D Search	7.44	0.00	7.34
Plus Pattern Search	6.15	3.66	6.11
Prefiltered Sampled Points 2	6.12	8.10	6.26
Prefiltered Sampled Points 3	6.35	7.70	6.45
Prefiltered Sampled Points 4	6.23	7.47	6.31
Prefiltered Sampled Points 5	6.11	7.70	6.20
Projection Search	6.30	9.21	6.45
Sampled Pts 2	4.84	6.89	4.97
Sampled Pts 3	5.01	7.97	5.16
Sampled Pts 4	4.99	7.34	5.14
Sampled Pts 5	5.14	8.33	5.30
Subblock Full Search	5.71	3.53	5.69
TSS	6.11	3.62	6.07
Tile search	5.09	5.79	5.14
Variance	5.86	9.11	6.03

Table B.21: Comparison of Algorithms for the sequence **rot_disc3**

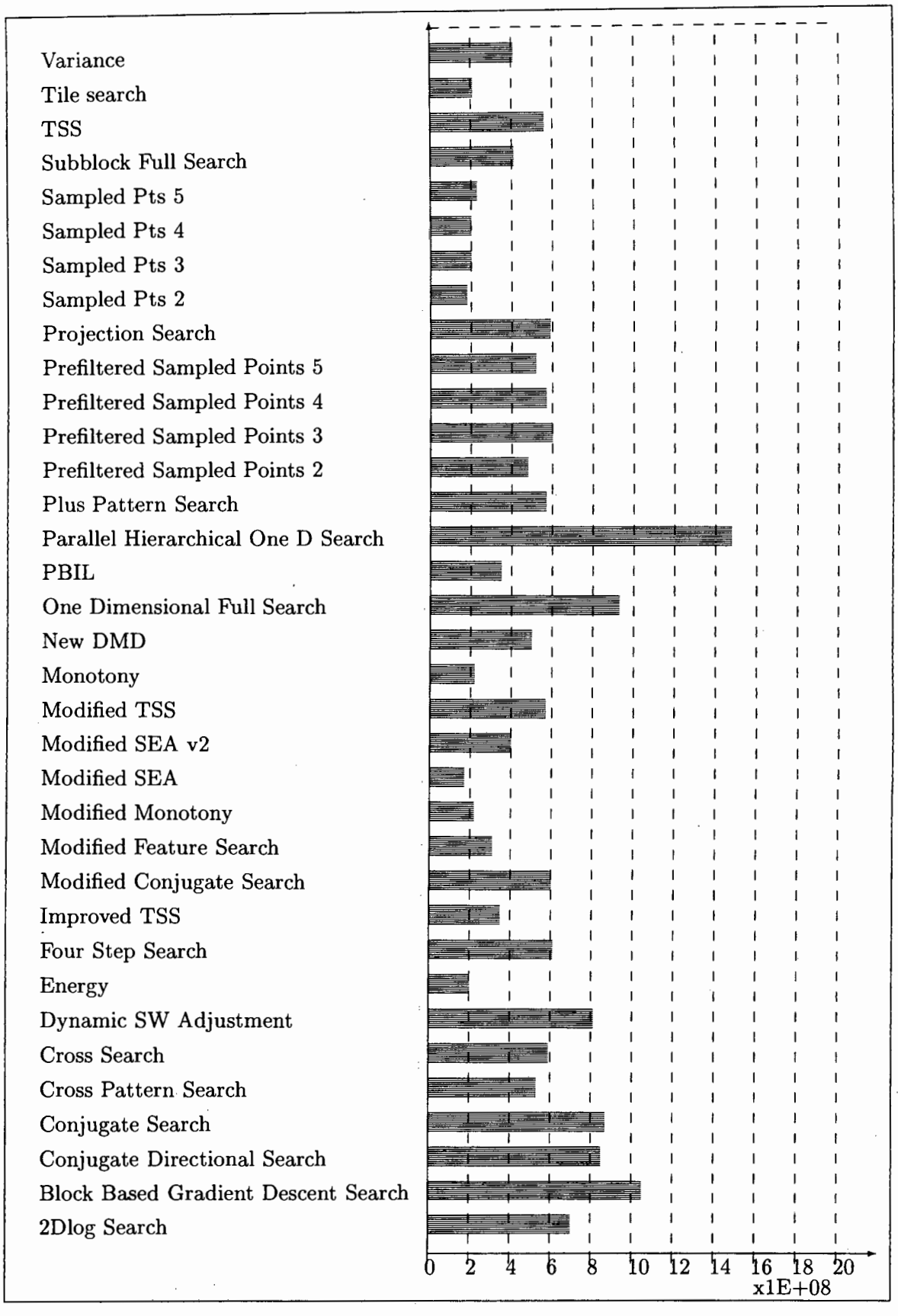


Figure B.55: **DFD** for sequence **rot_disc3**

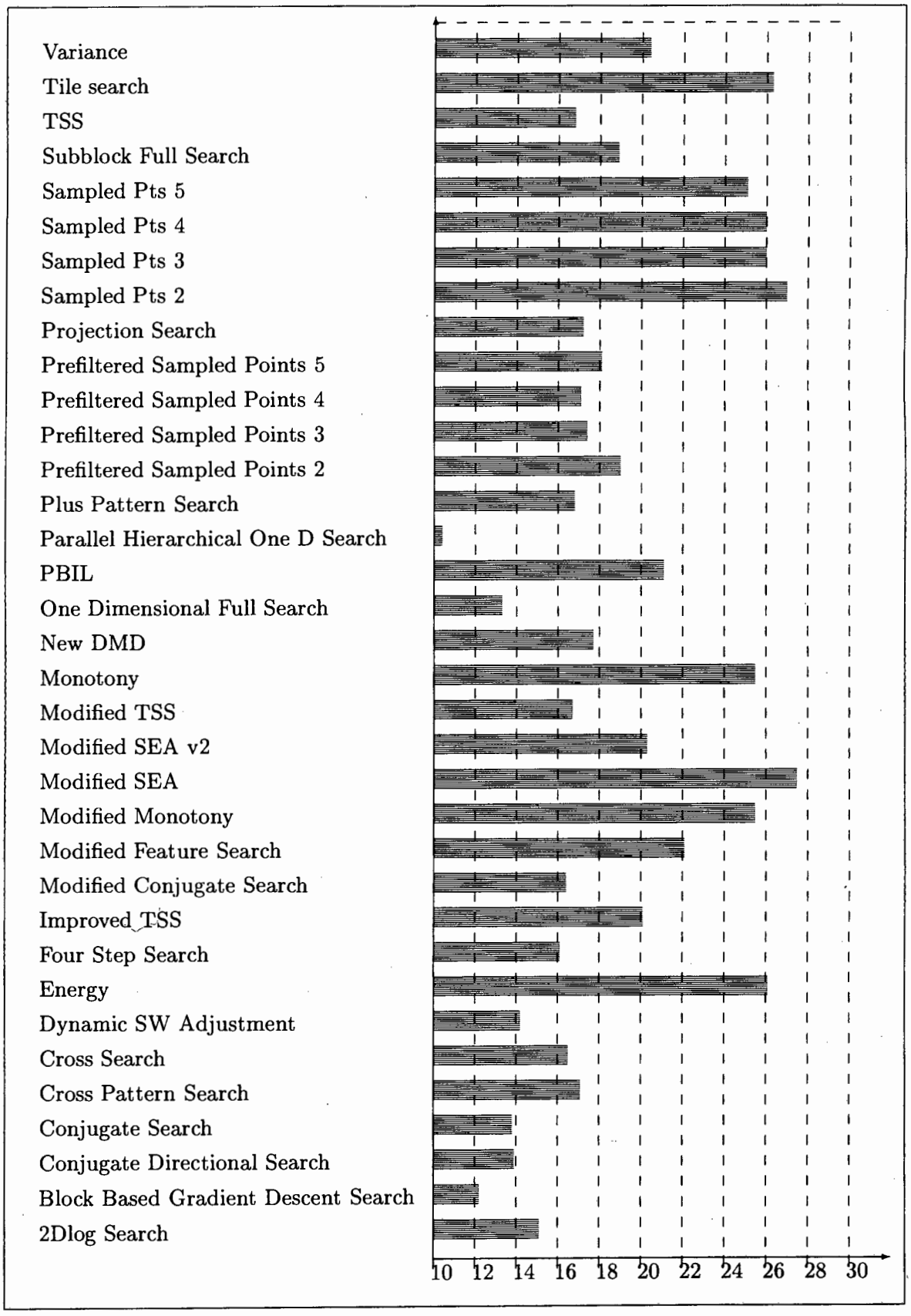


Figure B.56: **PSNR** for sequence **rot_disc3**

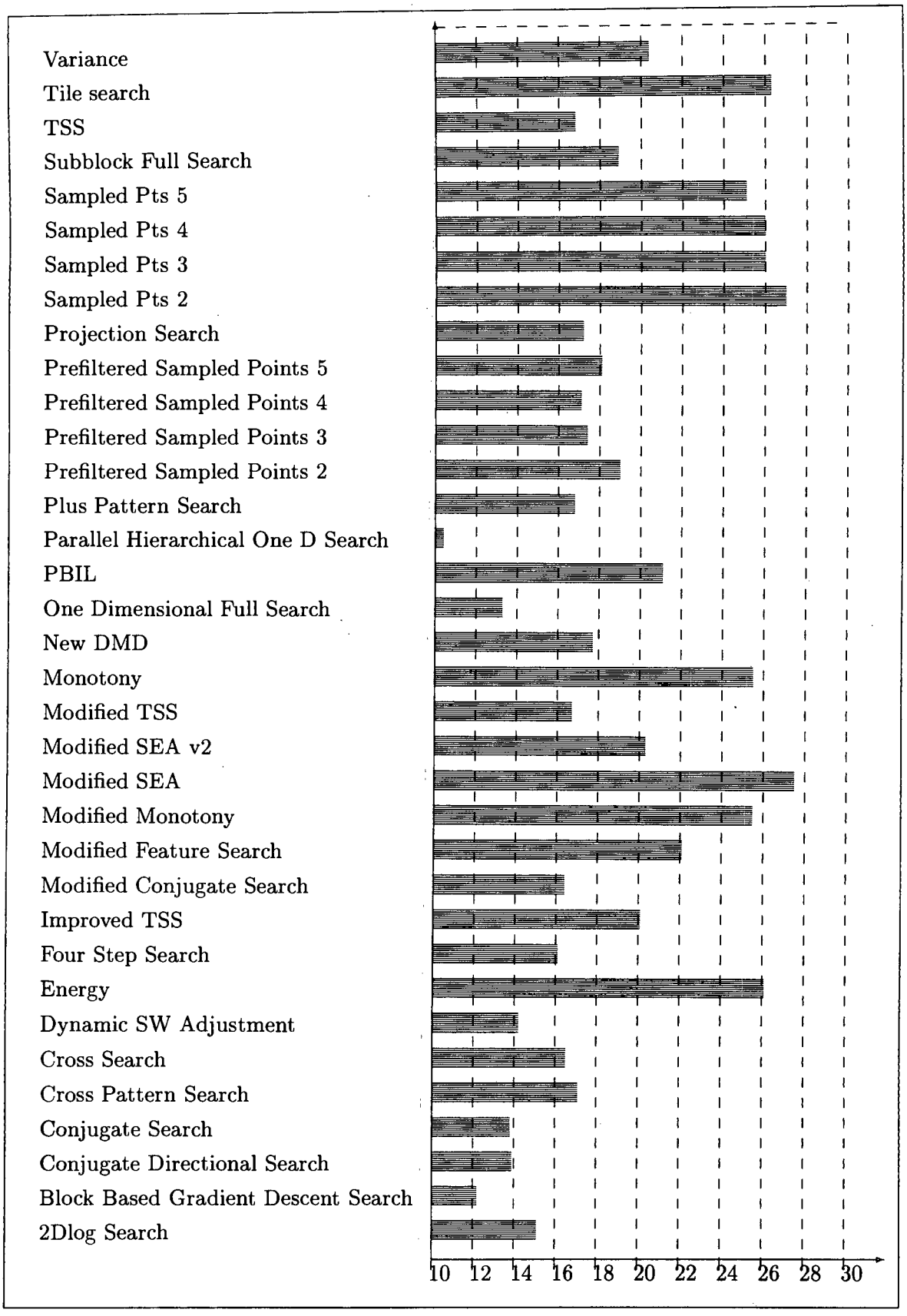


Figure B.56: **PSNR** for sequence **rot_disc3**

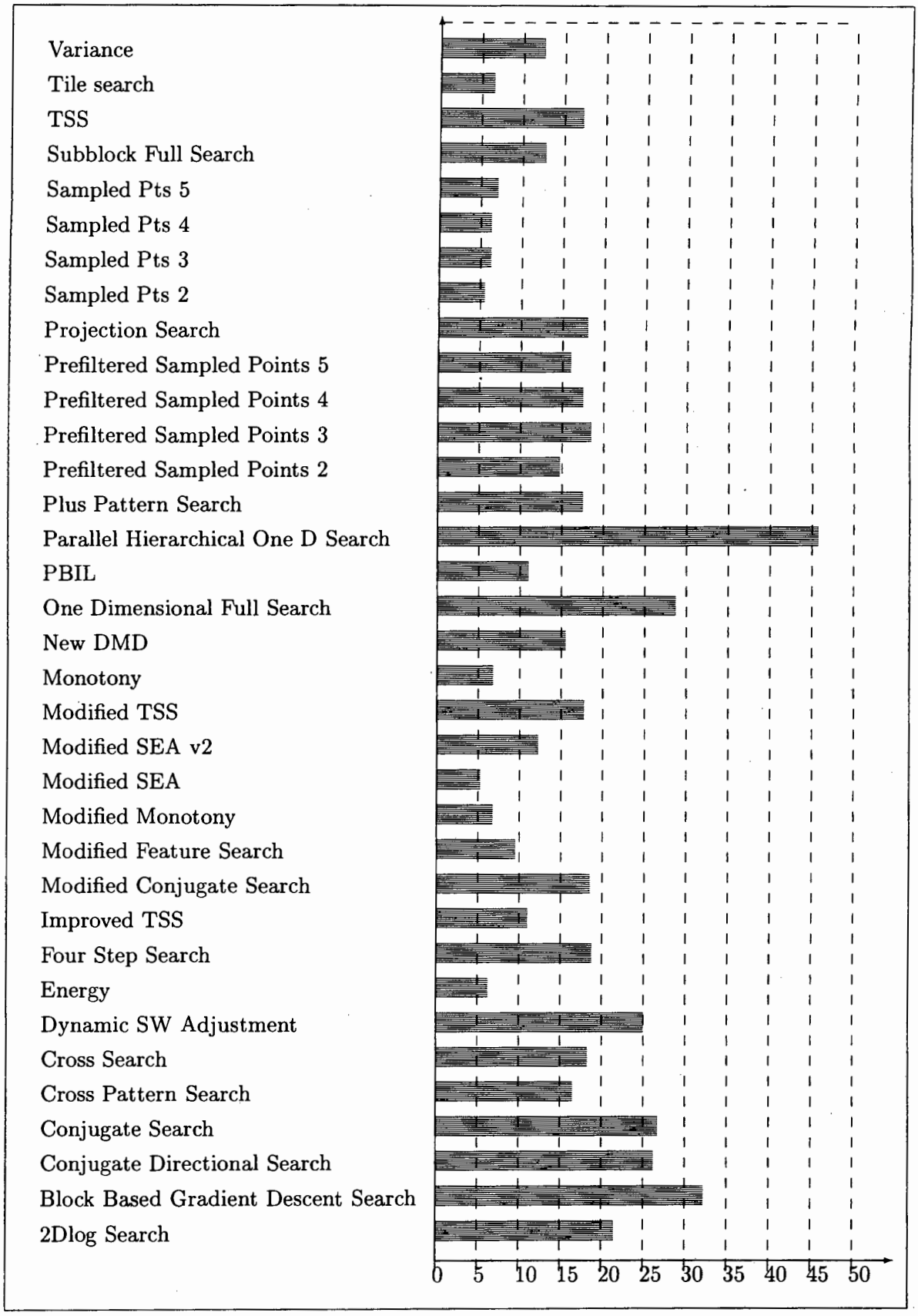


Figure B.57: **Di abs mean** for sequence **rot_disc3**

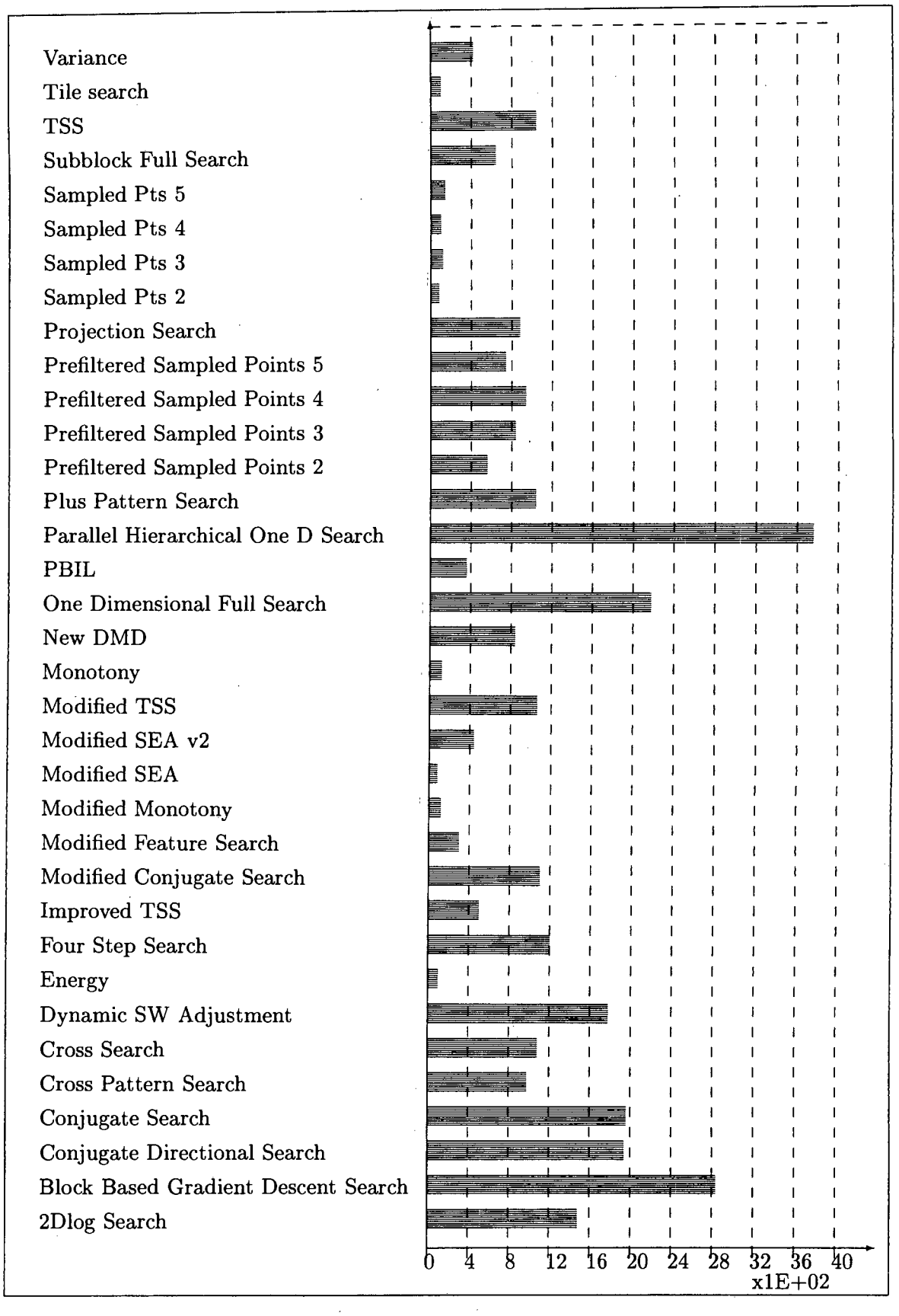


Figure B.58: **Di abs var** for sequence **rot_disc3**

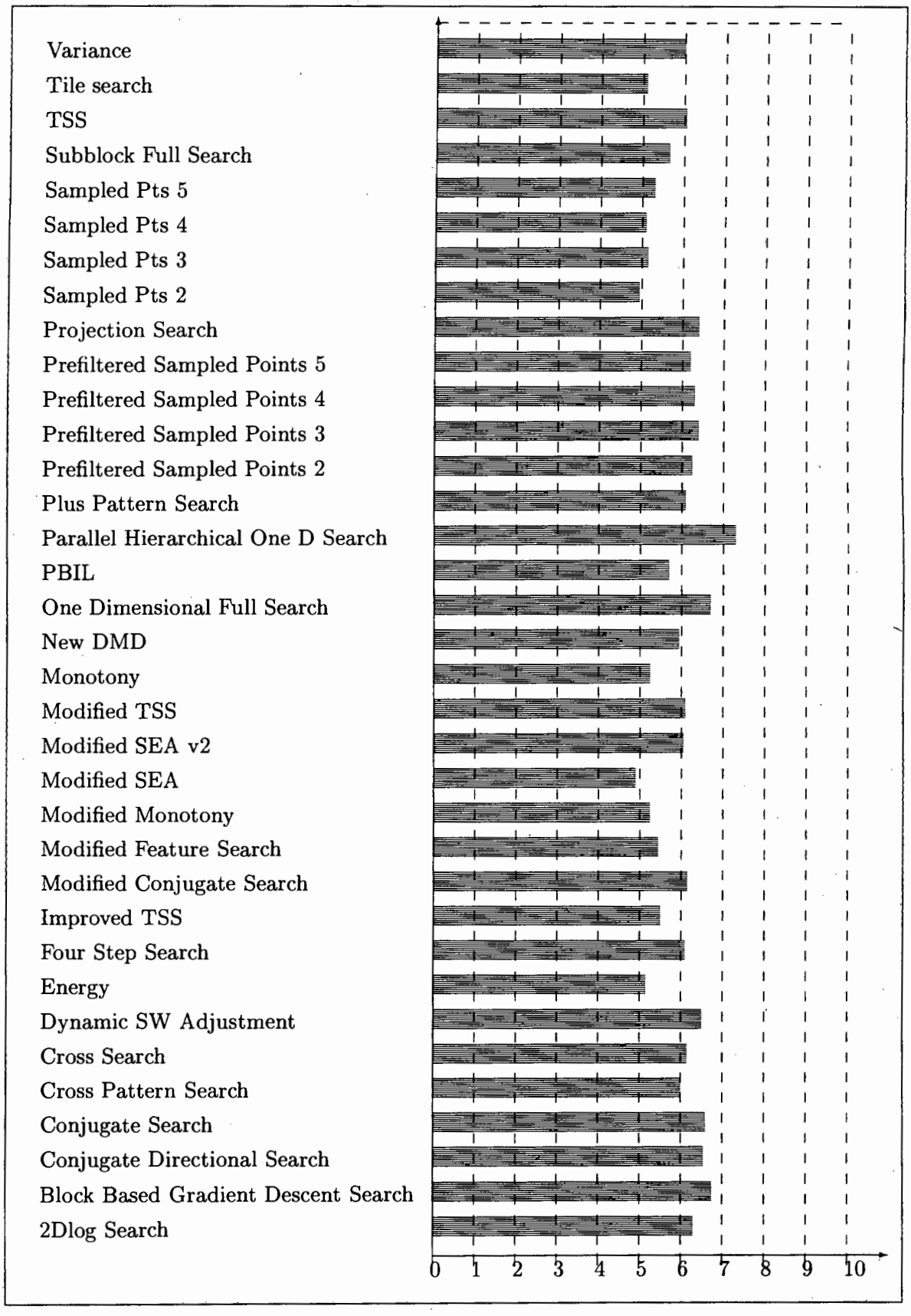


Figure B.59: **entropy cumulative** for sequence **rot_disc3**

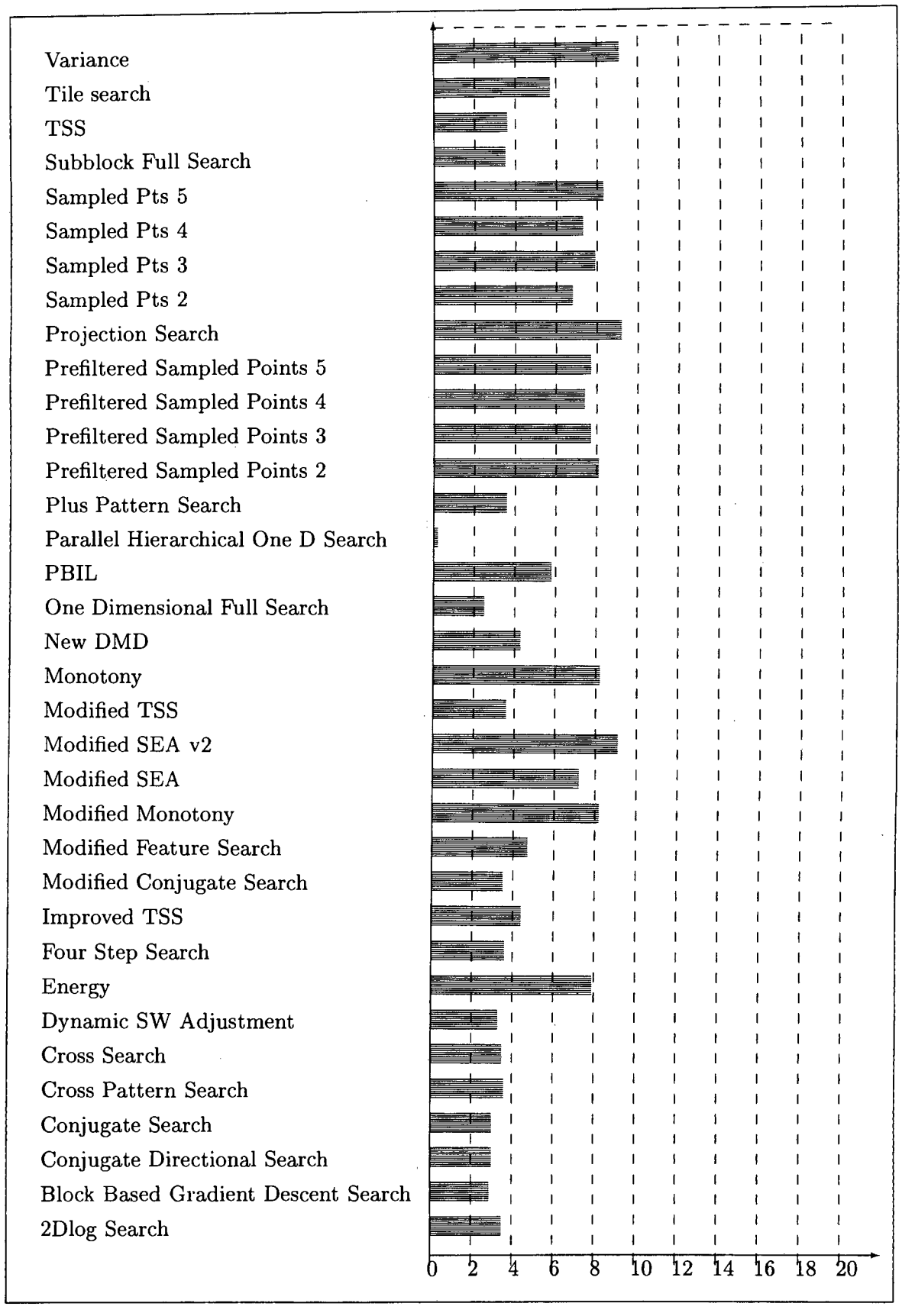


Figure B.60: **entropy mtn vec** for sequence **rot_disc3**

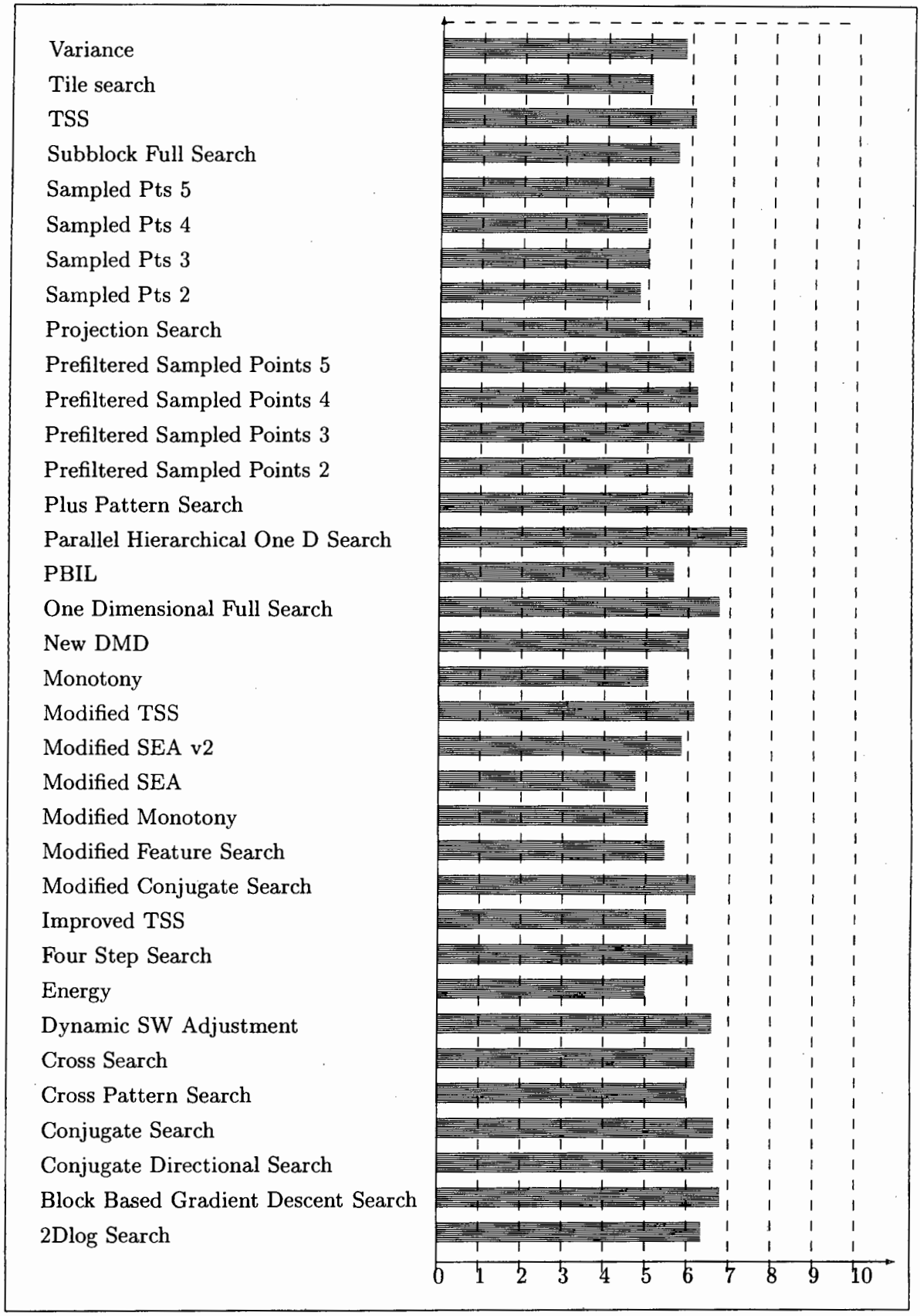


Figure B.61: **entropy pixel** for sequence **rot_disc3**

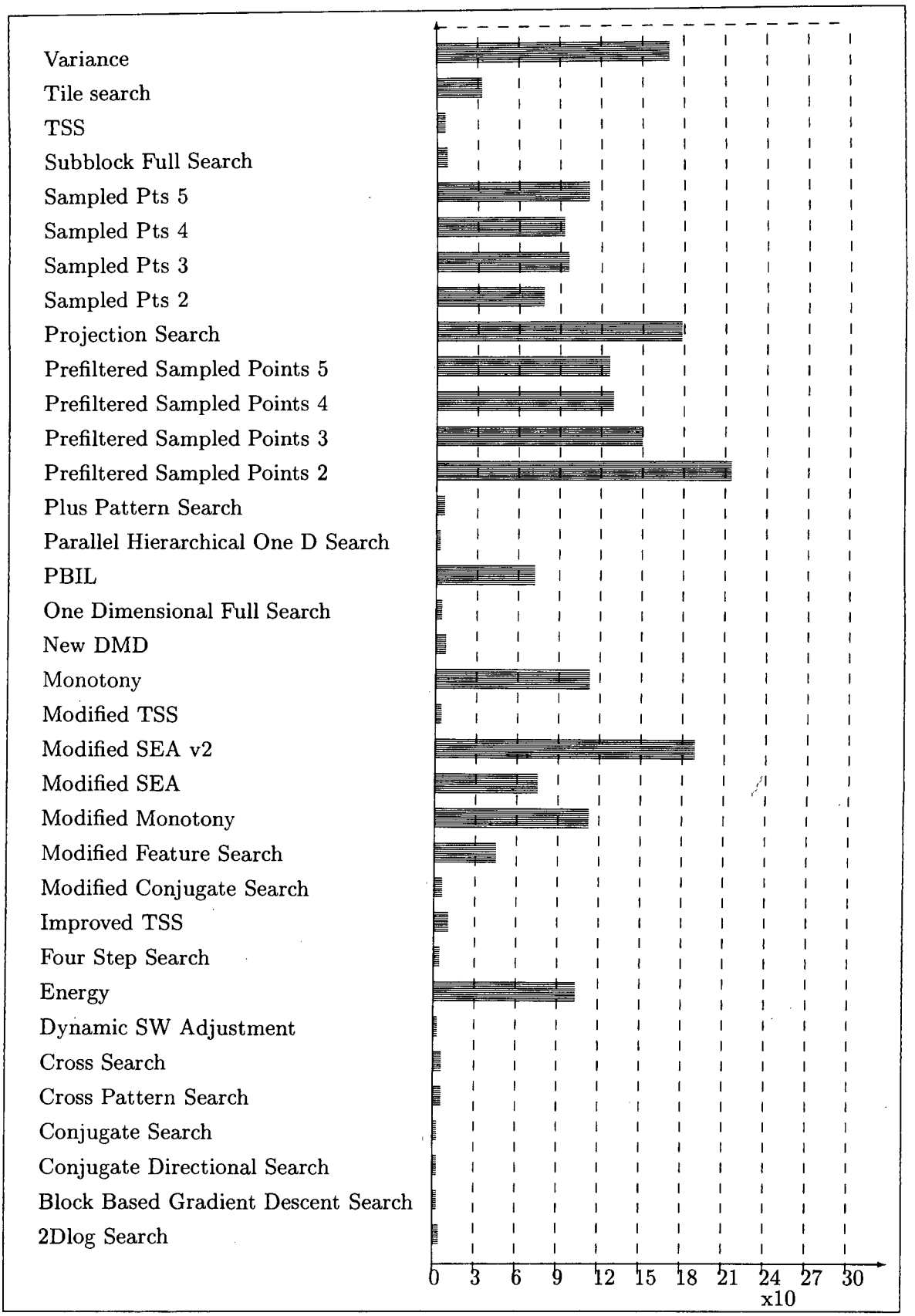


Figure B.62: **Motion Vector Means** for sequence **rot_disc3**

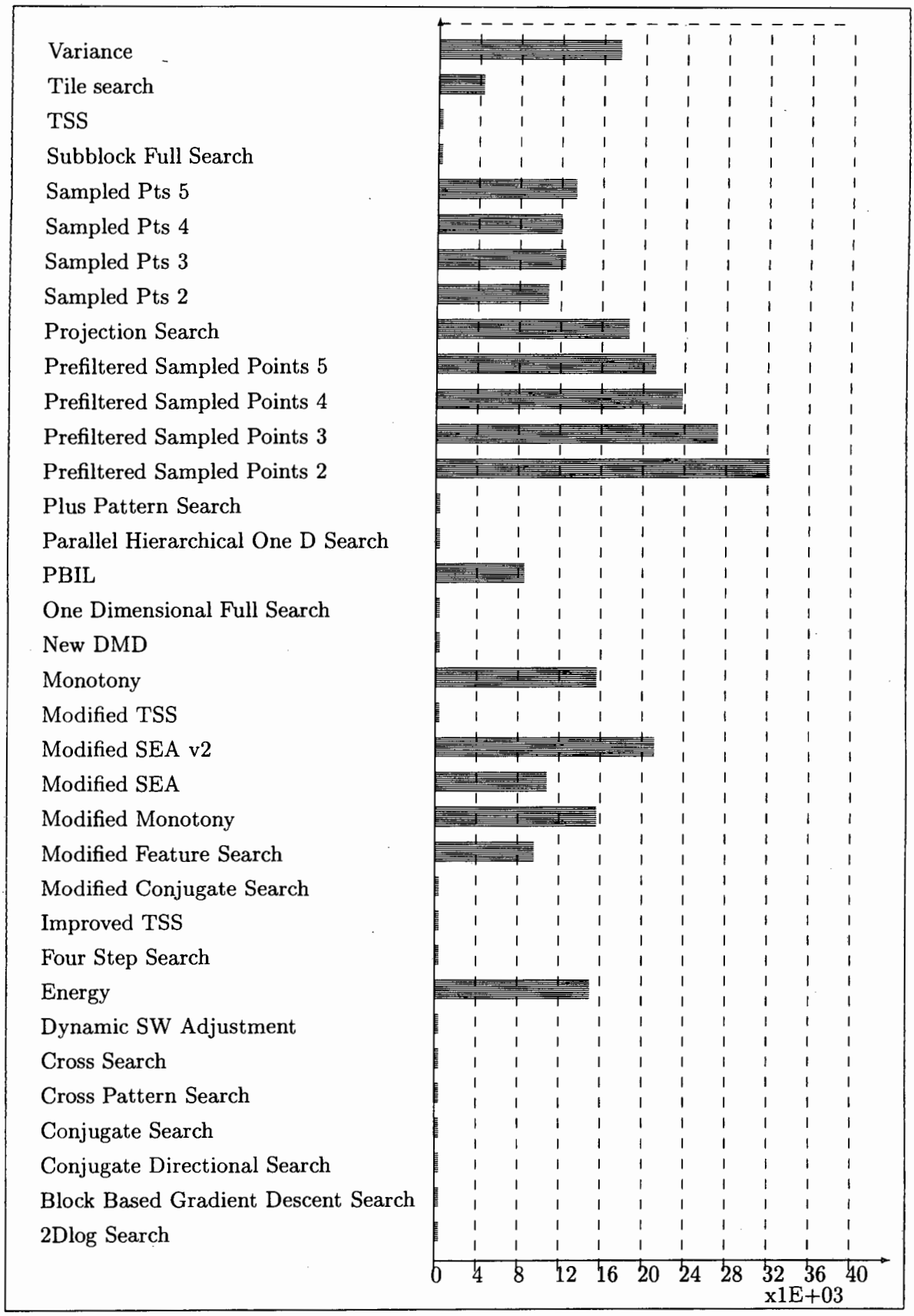


Figure B.63: **Motion Vector Var** for sequence **rot_disc3**

Algorithm	Difference block			
			Absolute	
	Mean	Variance	Mean	Variance
2Dlog Search	-0.03	454.9	9.2	370.2
Block Based Gradient Descent Search	-0.12	615.7	11.0	495.7
Conjugate Directional Search	-0.16	597.7	10.7	483.8
Conjugate Search	-0.16	591.2	10.6	478.0
Cross Pattern Search	0.05	357.9	8.5	286.0
Cross Search	-0.33	629.3	11.5	497.6
Dynamic SW Adjustment	0.03	470.0	9.4	381.3
Energy	-0.42	204.0	7.2	152.3
Four Step Search	-0.01	427.7	8.9	347.6
Improved TSS	0.10	285.4	7.5	228.7
Modified Conjugate Search	0.09	342.8	8.0	278.9
Modified Feature Search	0.01	160.3	6.3	120.0
Modified Monotony	-0.15	221.0	7.6	164.0
Modified SEA	-0.01	115.2	5.7	83.2
Modified SEA v2	0.18	1498.0	23.1	965.1
Modified TSS	0.05	383.9	8.5	311.7
Monotony	-0.13	216.1	7.5	160.0
New DMD	-0.19	524.3	10.8	408.1
One Dimensional Full Search	0.06	1301.6	17.3	1001.4
PBIL	-0.12	326.3	8.6	252.0
Parallel Hierarchical One D Search	-0.05	1464.9	18.7	1117.0
Plus Pattern Search	0.05	388.2	8.8	311.2
Prefiltered Sampled Points 2	-0.57	482.1	10.7	367.9
Prefiltered Sampled Points 3	1.61	541.6	11.1	421.2
Prefiltered Sampled Points 4	1.05	475.5	10.1	375.6
Prefiltered Sampled Points 5	0.54	409.6	9.2	325.2
Projection Search	-0.03	621.2	12.5	465.0
Sampled Pts 2	-0.03	135.0	6.0	99.1
Sampled Pts 3	-0.17	214.9	7.2	163.3
Sampled Pts 4	-0.10	182.4	6.8	136.8
Sampled Pts 5	-0.24	255.7	7.8	195.6
Subblock Full Search	0.09	299.7	7.5	243.3
TSS	0.04	375.9	8.5	304.5
Tile search	-0.20	227.0	7.8	166.5
Variance	0.06	611.4	12.5	456.1

Table B.22: Comparison of Algorithms for the sequence **ttennis**

Algorithm	Motion Vector		PSNR	DFD
	Mean	Variance		(x 10^8)
2Dlog Search	3.161	4.268	21.55	2.992
Block Based Gradient Descent Search	2.080	3.899	20.24	3.561
Conjugate Directional Search	2.588	4.545	20.37	3.470
Conjugate Search	2.559	4.320	20.41	3.459
Cross Pattern Search	4.509	6.726	22.59	2.756
Cross Search	3.986	7.735	20.14	3.732
Dynamic SW Adjustment	3.028	2.795	21.41	3.063
Energy	140.420	24461	25.03	2.342
Four Step Search	3.372	4.860	21.82	2.909
Improved TSS	4.930	15.163	23.58	2.449
Modified Conjugate Search	3.919	5.627	22.78	2.599
Modified Feature Search	26.974	7532	26.08	2.062
Modified Monotony	151.057	26166	24.69	2.455
Modified SEA	57.413	12247	27.52	1.837
Modified SEA v2	241.331	22658	16.38	7.504
Modified TSS	3.511	5.243	22.29	2.762
Monotony	149.922	26049	24.78	2.435
New DMD	5.057	7.601	20.94	3.504
One Dimensional Full Search	0.768	5.775	16.99	5.632
PBIL	41.282	7188	23.00	2.802
Parallel Hierarchical One D Search	0.001	0.006	16.47	6.063
Plus Pattern Search	4.679	6.718	22.24	2.852
Prefiltered Sampled Points 2	137.497	27496	21.30	3.478
Prefiltered Sampled Points 3	41.762	10841	20.79	3.606
Prefiltered Sampled Points 4	34.707	8108	21.36	3.268
Prefiltered Sampled Points 5	33.707	6903	22.01	2.992
Projection Search	164.226	20943	20.20	4.062
Sampled Pts 2	73.198	14920	26.83	1.947
Sampled Pts 3	117.876	20191	24.81	2.336
Sampled Pts 4	105.222	19181	25.52	2.195
Sampled Pts 5	133.260	21211	24.05	2.521
Subblock Full Search	4.270	7.127	23.36	2.442
TSS	3.978	5.167	22.38	2.747
Tile search	29.558	5132	24.57	2.530
Variance	194.094	22934	20.27	4.051

Table B.23: Comparison of Algorithms for the sequence **ttennis**

Algorithm	Entropy		
	Pixel	Motion Vector	Cumulative
2Dlog Search	5.33	2.95	5.28
Block Based Gradient Descent Search	5.56	2.59	5.51
Conjugate Directional Search	5.52	2.69	5.47
Conjugate Search	5.52	2.68	5.47
Cross Pattern Search	5.29	3.42	5.26
Cross Search	5.67	3.57	5.63
Dynamic SW Adjustment	5.36	2.71	5.31
Energy	5.15	8.05	5.30
Four Step Search	5.30	3.04	5.26
Improved TSS	5.12	3.46	5.09
Modified Conjugate Search	5.16	2.80	5.13
Modified Feature Search	4.98	4.07	4.98
Modified Monotony	5.23	8.22	5.38
Modified SEA	4.85	5.51	4.91
Modified SEA v2	6.84	9.53	7.00
Modified TSS	5.24	3.01	5.20
Monotony	5.22	8.20	5.37
New DMD	5.62	3.73	5.58
One Dimensional Full Search	6.21	0.69	6.14
PBIL	5.35	4.24	5.36
Parallel Hierarchical One D Search	6.31	0.00	6.23
Plus Pattern Search	5.33	3.46	5.29
Prefiltered Sampled Points 2	5.64	6.64	5.74
Prefiltered Sampled Points 3	5.63	4.10	5.63
Prefiltered Sampled Points 4	5.49	4.18	5.48
Prefiltered Sampled Points 5	5.37	4.34	5.37
Projection Search	5.87	8.87	6.01
Sampled Pts 2	4.92	5.85	5.00
Sampled Pts 3	5.13	7.64	5.26
Sampled Pts 4	5.06	6.70	5.17
Sampled Pts 5	5.22	8.02	5.37
Subblock Full Search	5.10	3.09	5.06
TSS	5.24	3.12	5.20
Tile search	5.28	4.57	5.29
Variance	5.86	9.23	6.04

Table B.24: Comparison of Algorithms for the sequence **ttennis**

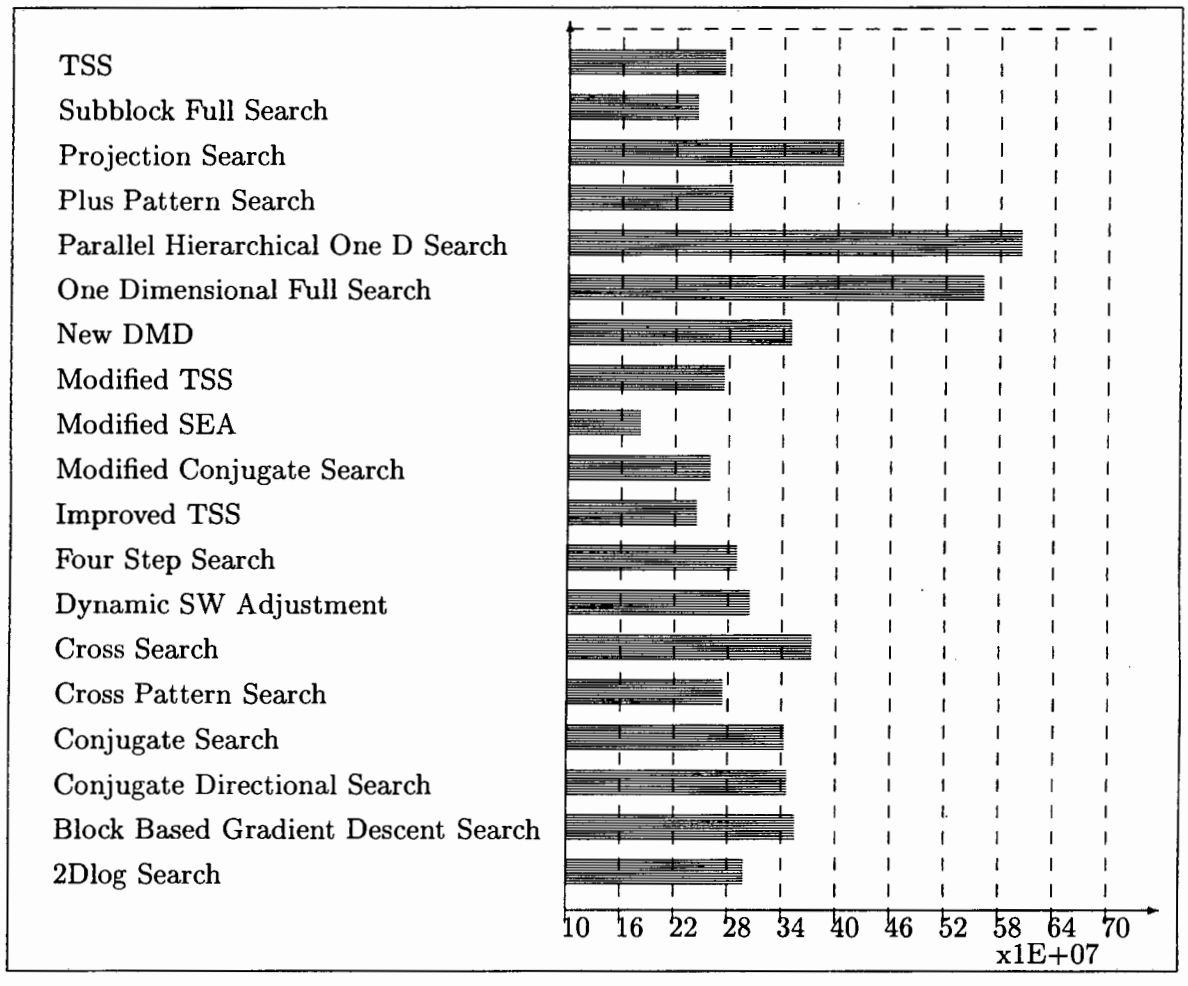


Figure B.64: **DFD** for sequence **ttennis**

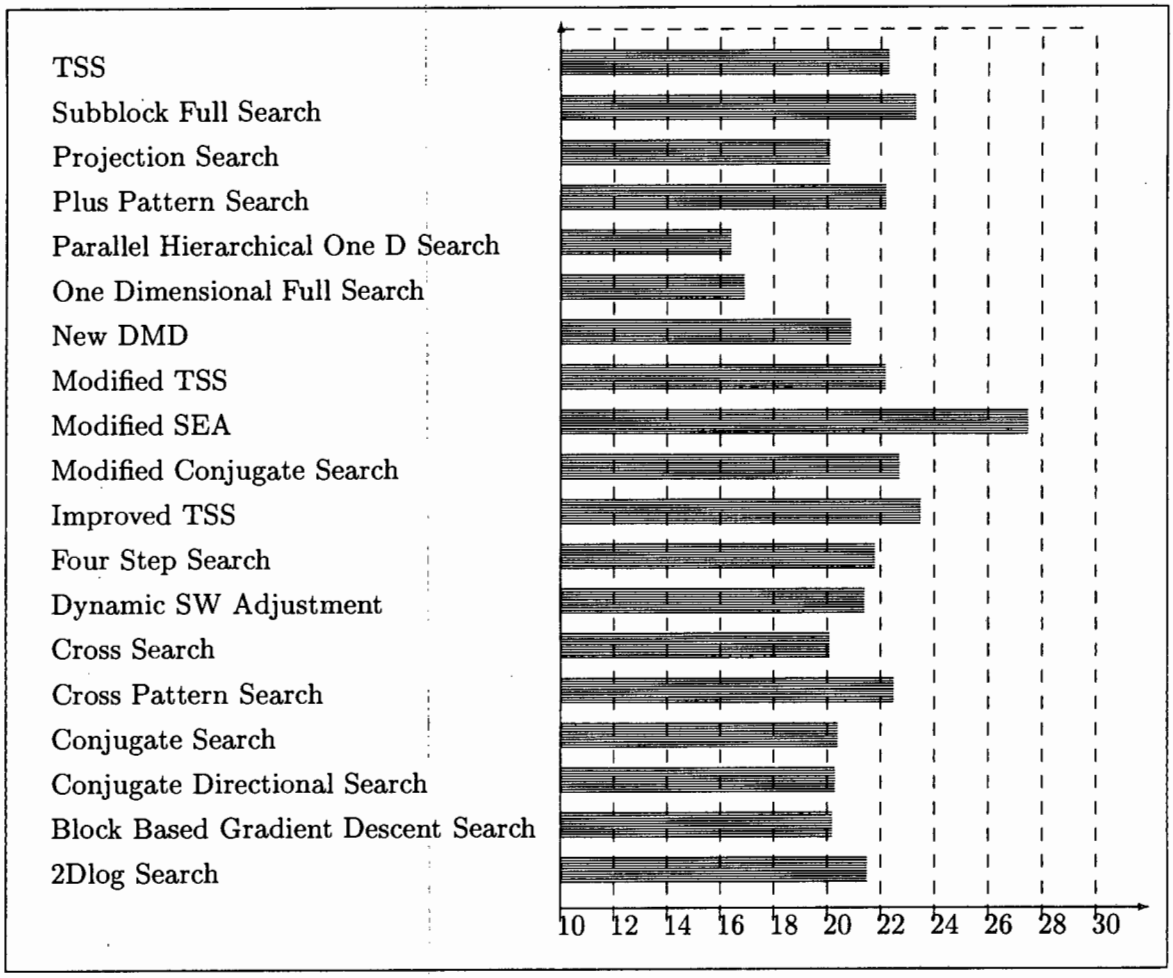


Figure B.65: **PSNR** for sequence **ttennis**

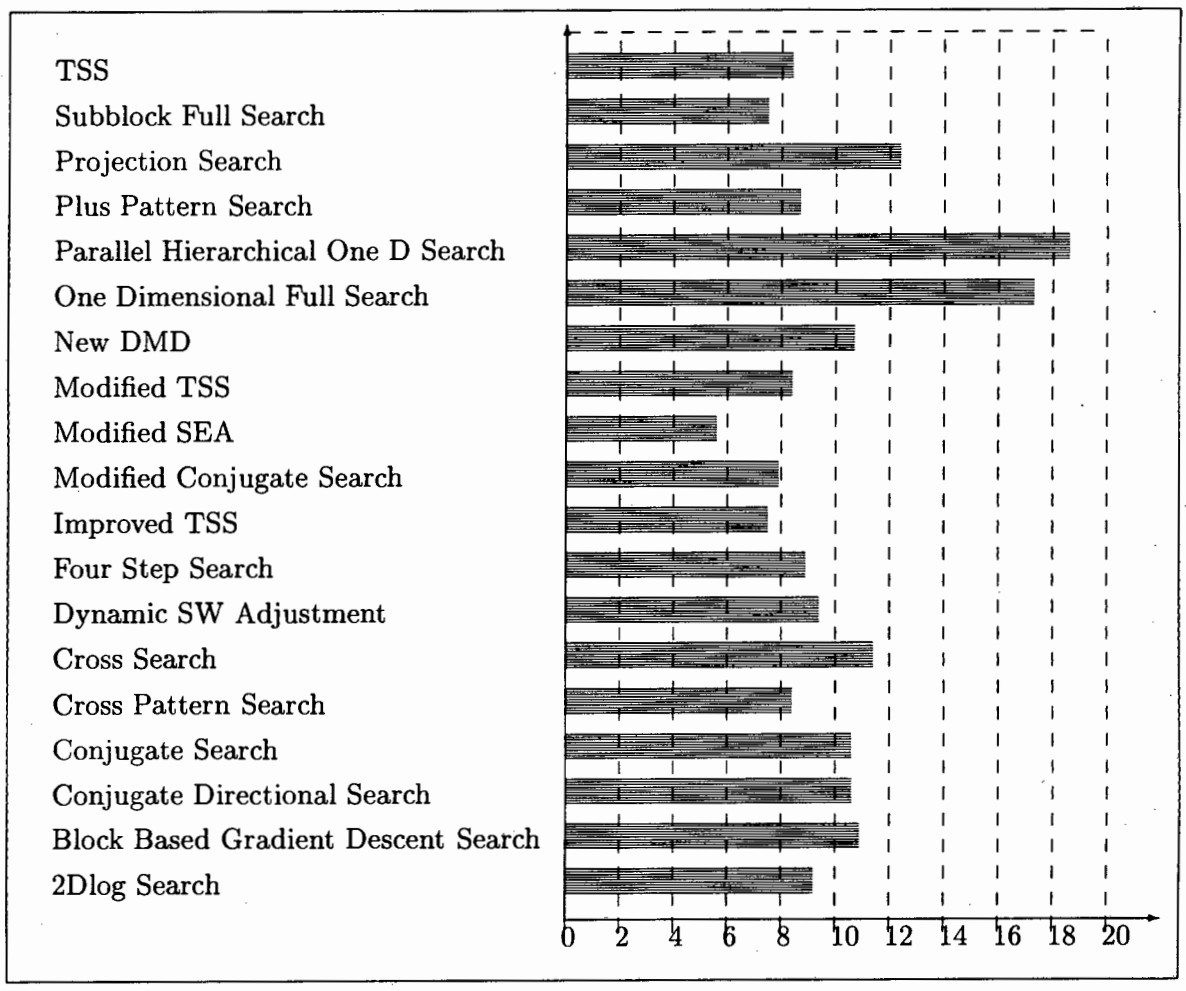


Figure B.66: **Di abs mean** for sequence **ttennis**

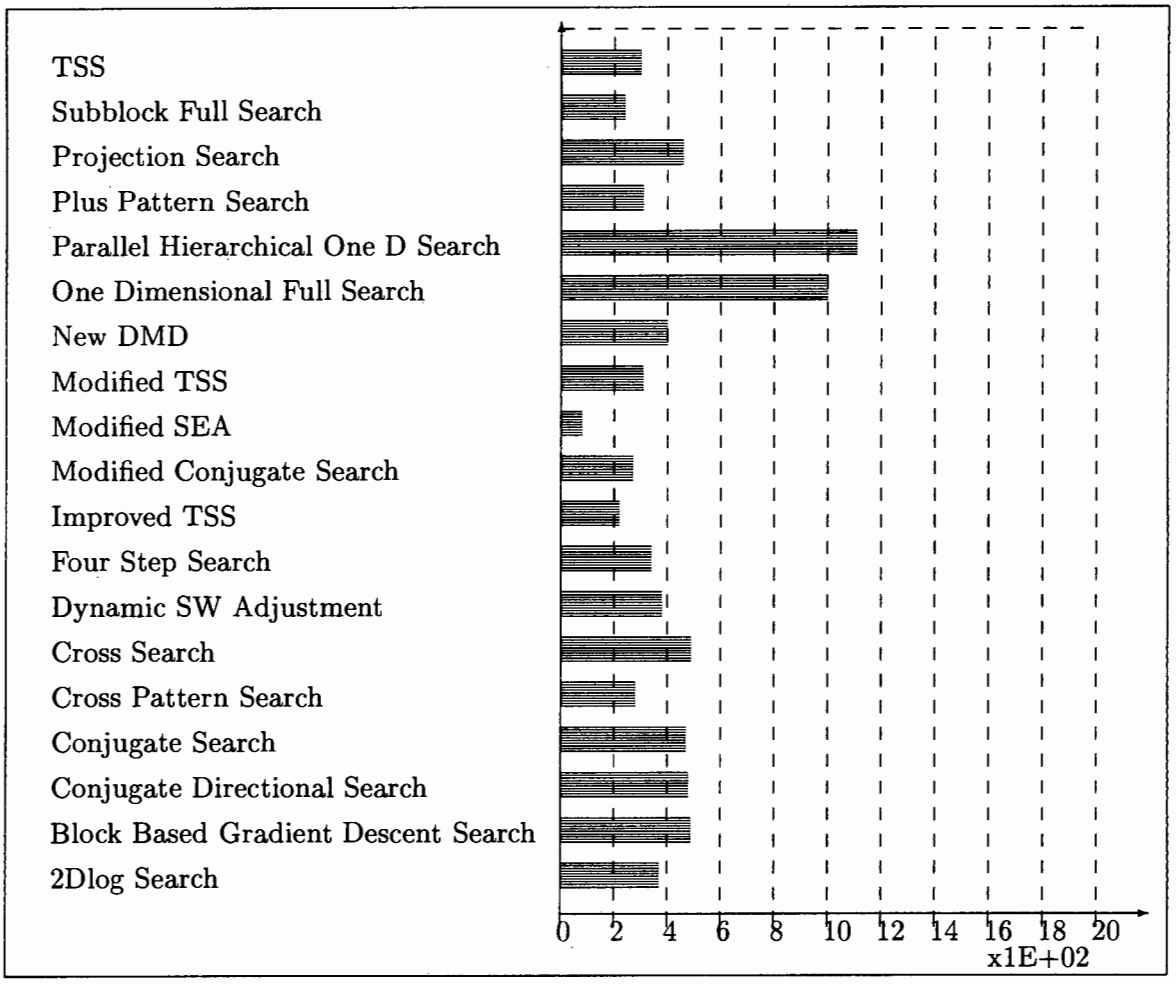


Figure B.67: **Di abs var** for sequence **ttennis**

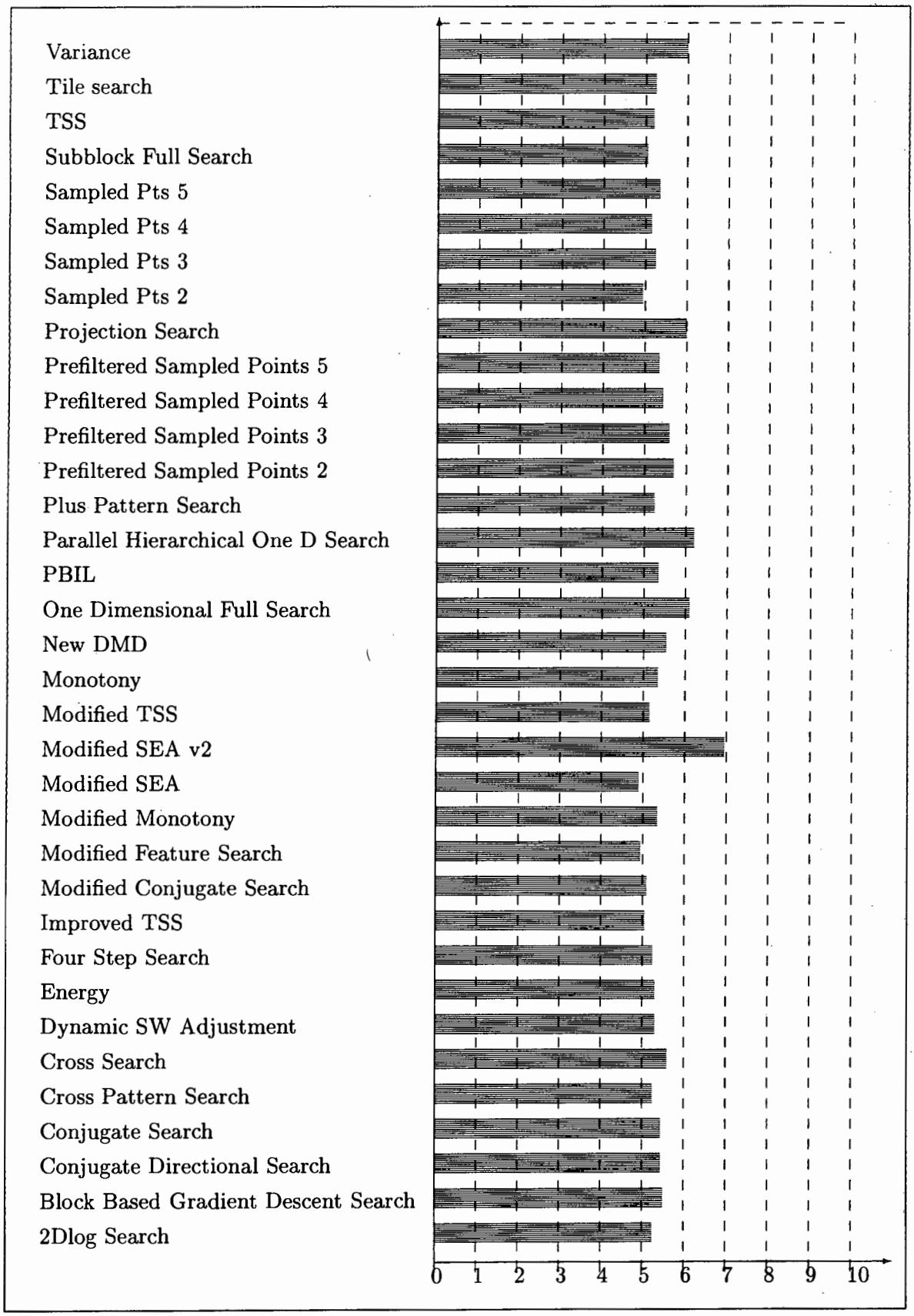


Figure B.68: **entropy cumulative** for sequence **ttennis**

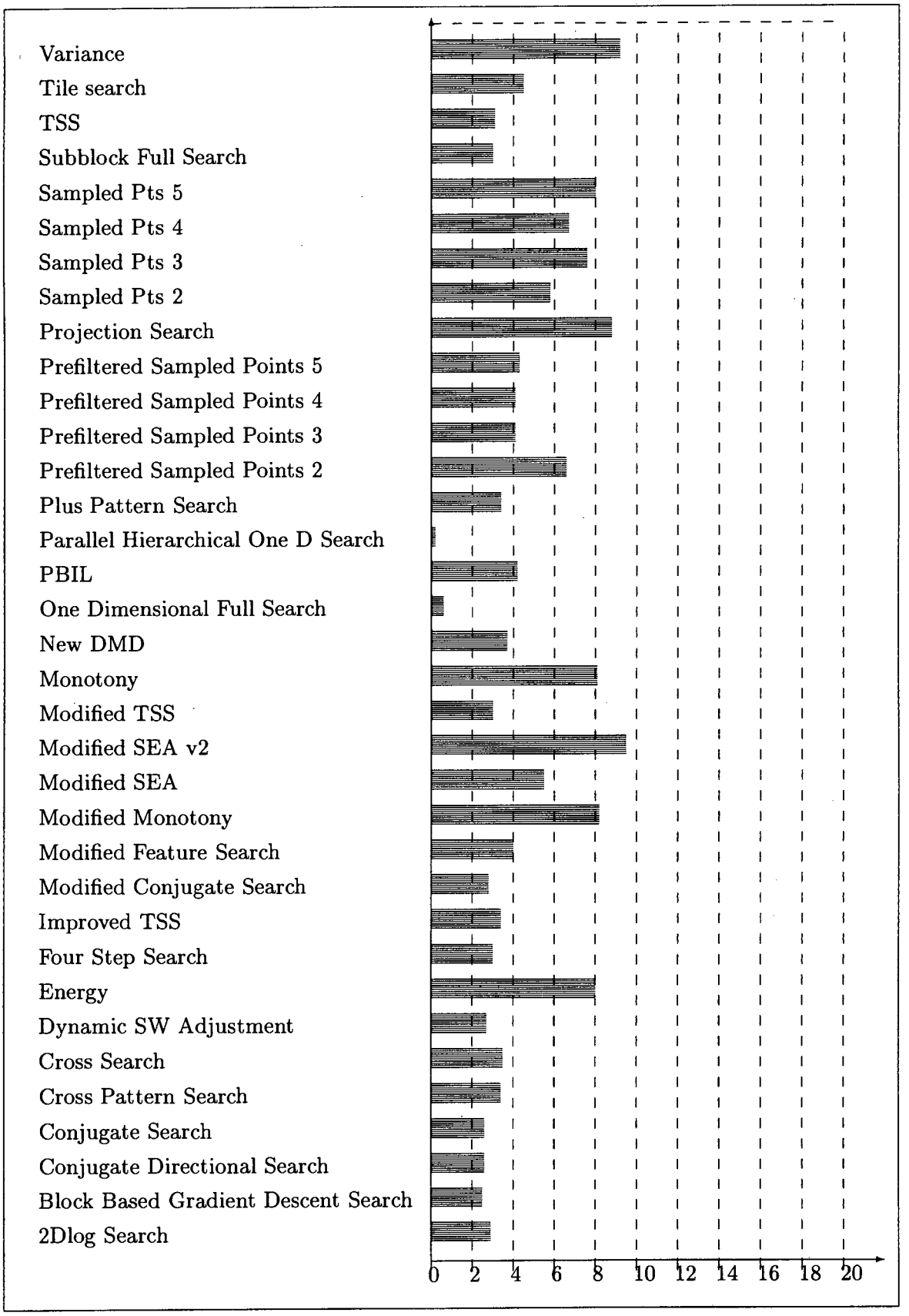


Figure B.69: **entropy mtn vec** for sequence **ttennis**

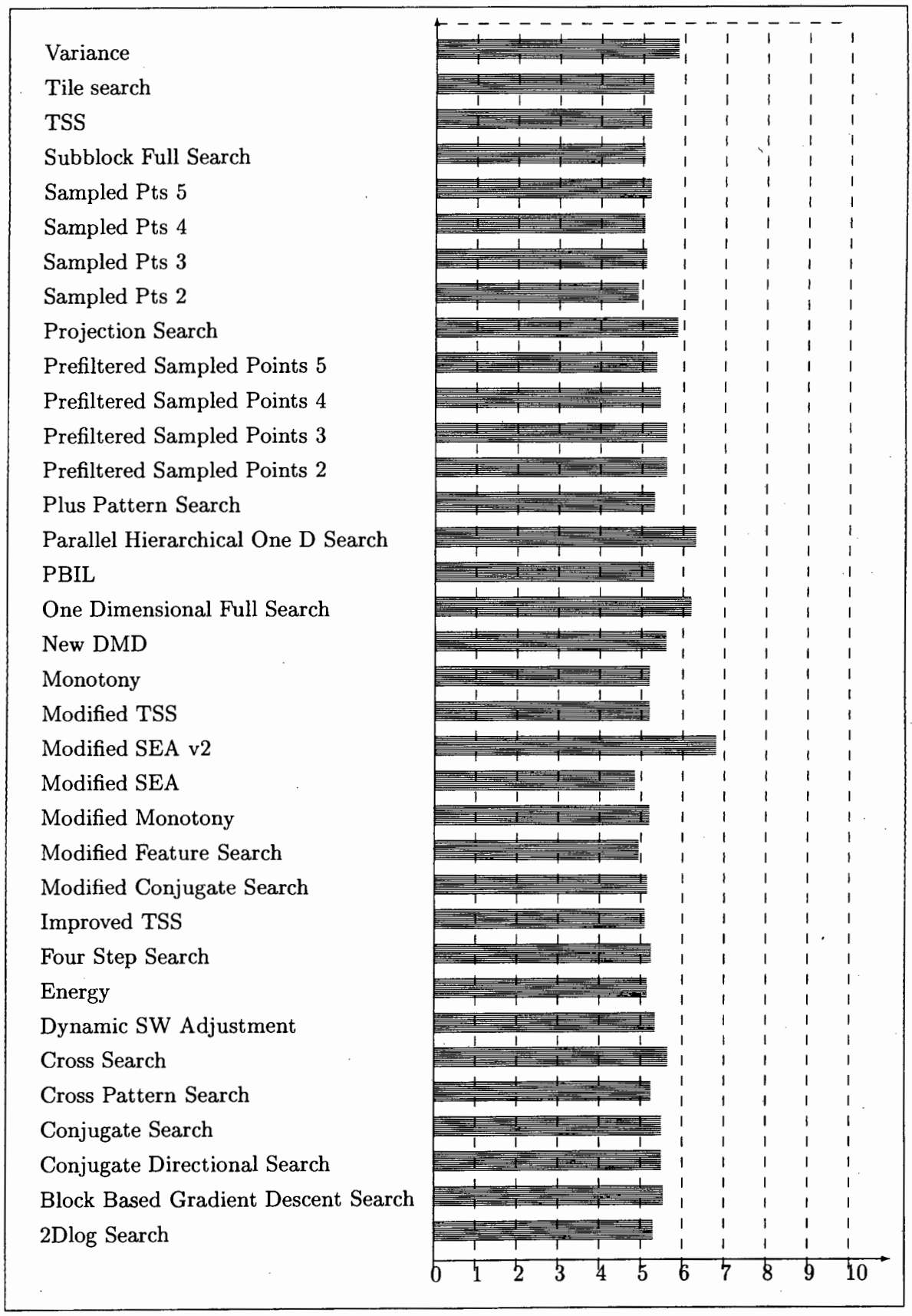


Figure B.70: **entropy pixel** for sequence **ttennis**

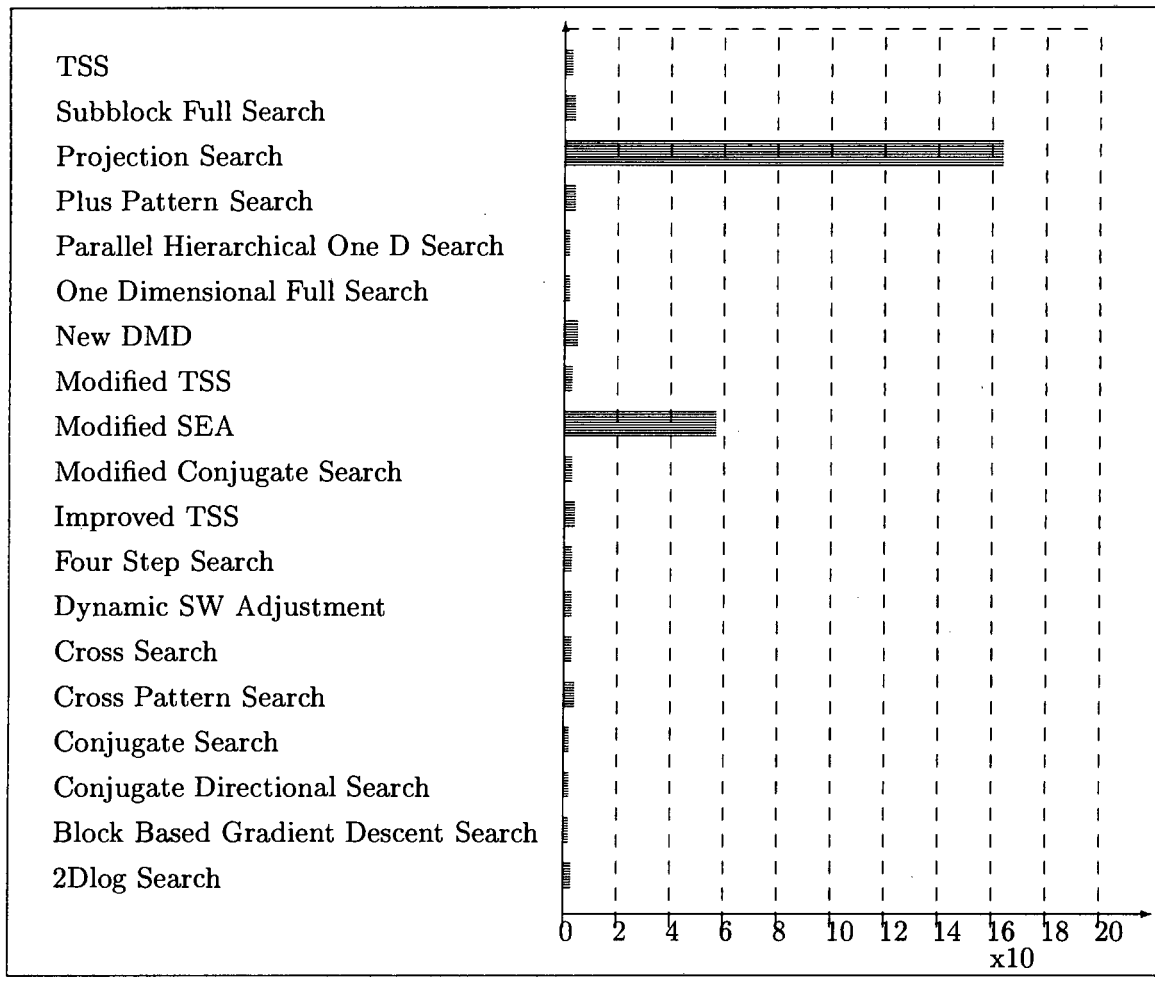


Figure B.71: **Motion Vector Means** for sequence **ttennis**

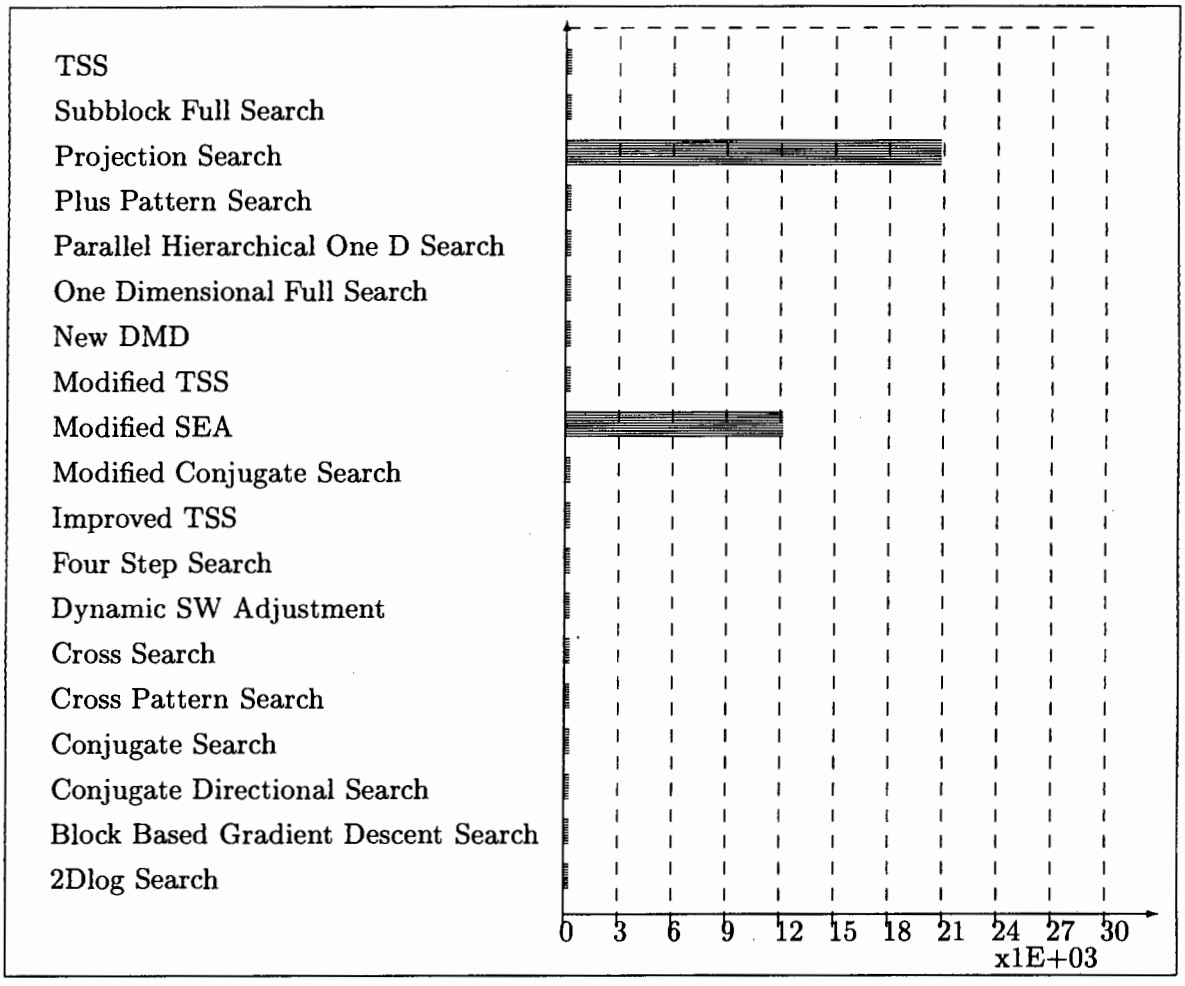


Figure B.72: **Motion Vector Var** for sequence **ttennis**

Appendix C

Essential Code Listings

C.1 Basic Classes

C.1.1 Bag.h

```
#ifndef __MY_BAG_STORAGE__
#define __MY_BAG_STORAGE__

#include <iostream.h>

//-------------------------------------------------------------
/*
This is just a storage class really...


 */

template< class T >
class Bag
{
public:

  Bag():data(NULL),no_elements(0),next_item(0),
    index(0){};
  Bag( const int size):data(NULL),no_elements(size),
    next_item(0),index(0){ alloc_mem();};
  Bag( const Bag<T> &b);
  ~Bag(){ delete [] data;};

  Bag<T>& operator=( const Bag<T> &b) ;

  // accessor functions.

  void reset_iterator() const { index = 0;};
  const T& current() const { return data[index] ;};
  int current( const int pos, T &out) const ;

  int insert_item( const T& item) ;
  int delete_item( const T& item) ;

  int get_size() const { return no_elements;};
  int get_number_of_stored_elements() const
  { return next_item;};

  T get_element( const int i){ if( (i >= 0) &&
   ( i < next_item))
    return data[i]; else {cerr << "OUt of bounds
BAG::get_element!";exit(1);};};


protected:

  T *data ;
  short int no_elements , next_item ;

  mutable short int index;

  void alloc_mem() ;

private:
  friend ostream& operator<<( ostream &o,
      const Bag<T> &b);
};
//-------------------------------------------------------------

template< class T>
Bag<T>::Bag( const Bag<T> &b)
{
  no_elements = b.no_elements ;
  next_item = b.next_item ;
  index = 0 ;

  alloc_mem() ;
}
//-------------------------------------------------------------
template< class T>
Bag<T> &
Bag<T>::operator=( const Bag<T> &b)
{
  no_elements = b.no_elements ;
  next_item = b.next_item ;
  index = 0 ;

  alloc_mem() ;

  for( int i = 0 ; i < next_item ; i++)
    data[i] = b.data[i] ;

  return *this ;
}
//-------------------------------------------------------------
template< class T>
int
Bag<T>::current( const int pos, T &out) const
{
  if( (pos < no_elements) && ( pos >= 0))
    {
      out =  data[pos] ;
      index = pos ;
      return 1;
    }
  return 0;
}

//-------------------------------------------------------------
template< class T>
int
Bag<T>::insert_item( const T& item)
{
  if( (next_item + 1) <= no_elements )
    {
      data[next_item] = item ;
      next_item++;
      return 1;
    }
  return 0;
}
//-------------------------------------------------------------

template< class T>
int
Bag<T>::delete_item( const T& item)
{
  for( int i = 0 ; i < next_item ; i++)
    {
      if( data[i] == item)
{
  for( int j = i + 1; j < next_item ; j++)
    data[ j - 1] = data[j] ;
  // shift everything one down.

  next_item--;

  return 1;
}
    }
  return 0;
}

//-------------------------------------------------------------
template< class T>
void
Bag<T>::alloc_mem()
{
  delete data;

  if( no_elements < 0)
    {
      cerr <<
"BAG:Number of elements to bag must be positive!"
   << endl ;
      exit(1);
    }

  data = new T[no_elements] ;

  if( data == NULL)
    {
      cerr << "BAG:Unable to allocate memory!" << endl ;
      exit(1);
    }

  for( int i = 0 ; i < no_elements ; i++)
    data[i] = T();
}
//-------------------------------------------------------------
template< class T>
ostream& operator<<( ostream &o, const Bag<T> &b)
{
  for( int i = 0 ; i < b.next_item ; i++ )
    o << b.data[i] << endl ;

  return o;
}

//-------------------------------------------------------------

#endif
```

C.1.2 PBIL.h

```
// stuff to implement a PBIL for motion vector searches

#ifndef __MY_PBIL__
#define __MY_PBIL__

#include <iostream.h>
#include <stdlib.h>


const int PBIL_LENGTH = 18 ;
const int HALF_PBIL_LENGTH = 9;
const float L = 0.1; // old = 0.05
const float MUTATION_VALUE = 0.03;
const unsigned int LOCAL_MASK = 511 ;

class PBIL
{
public:

  PBIL(){zero();} ;
  ~PBIL(){};

  int extract_x() const ;
  int extract_y() const ;

  void set_x( const int x) ;
  void set_y( const int y) ;

  void set_results_x( const int x);
  void set_results_y( const int y);

  void evaluation( const int x ,
   const int last_evaluation);

  void update_x() ;
  void update_y() ;

  void mutate() ;
  void reseed( const unsigned int seed);
  void reseed() ;

  void next() ;
  void zero() ;

  int final_output_x() const ;
  int final_output_y() const ;

private:

  float result_x[ HALF_PBIL_LENGTH ];
  float result_y[ HALF_PBIL_LENGTH ];

  char bits_x[ HALF_PBIL_LENGTH ];
  char bits_y[ HALF_PBIL_LENGTH ];

  char best_bits_x[ HALF_PBIL_LENGTH ];
  char best_bits_y[ HALF_PBIL_LENGTH ];

  //char log[512][512];

  int get_random() ;

};

//-----------------------------------------------------------------

void
PBIL::zero()
{
  /*for( int i = 0 ; i < 512 ; i++)
    for( int j = 0 ; j < 512 ; j++)
      {
log[i][j] = 0;
      }*/

  for( int j = 0 ; j < HALF_PBIL_LENGTH; j++)
    {
      result_x[j] = 0.5 ;
      result_y[j] = 0.5 ;
    }
}
//-----------------------------------------------------------------
inline int
PBIL::get_random()
{
  int rnd = rand() ;

  return (rnd>>5);
}
//-----------------------------------------------------------------
inline int
PBIL::extract_x() const
{
  int res = 0 ;
  int bit_val = 1;

  for( int i = 0 ; i < HALF_PBIL_LENGTH; i++)
    {
      res += ( bits_x[i] == 1) ? bit_val : 0;
      bit_val = bit_val<<1 ;
    }
  return res ;
}
//-----------------------------------------------------------------
inline int
PBIL::extract_y() const
{
  int res = 0 ;
  int bit_val = 1;

  for( int i = 0 ; i < HALF_PBIL_LENGTH; i++)
    {
      res += ( bits_y[i] == 1) ? bit_val : 0;
      bit_val = bit_val<<1 ;
    }
  return res ;
}

//-----------------------------------------------------------------

inline void
PBIL::set_x( const int val)
{
  int mask = 1;

  if( val < 0)
    {
      cerr << "Require a positive value"
 <<  endl ;
      exit(1);
    }

  unsigned int num = (unsigned int)(val);

  for( int i = 0 ; i < HALF_PBIL_LENGTH; i++)
    {
      if( (mask & num) > 0)
{
  bits_x[i] = 1;
}
      else bits_x[i] = 0;

      mask = mask << 1;
    }
}
//-----------------------------------------------------------------
inline void
PBIL::set_y( const int val)
{
  int mask = 1;

  if( val < 0)
    {
      cerr << "Require a positive value"
 <<  endl ;
      exit(1);
    }

  unsigned int num = (unsigned int)(val);

  for( int i = 0 ; i < HALF_PBIL_LENGTH; i++)
    {
      if( (mask & num) > 0)
{
  bits_x[i] = 1;
}
      else bits_x[i] = 0;

      mask = mask << 1;
    }
}//-----------------------------------------------------------------
inline void
PBIL::set_results_x( const int x)
{
  int mask = 1;

  for( int i = 0 ; i < HALF_PBIL_LENGTH; i++)
    {
```

```
      result_x[i] = ( (x & mask) > 0 )
? 0.75 : 0.25;

      mask = mask<<1 ;
    }
}
//---------------------------------------------------------------
inline void
PBIL::set_results_y( const int y)
{
  int mask = 1;

  for( int i = 0 ; i <  HALF_PBIL_LENGTH; i++)
    {
      result_y[i] = ( (y & mask) > 0 )
? 0.75 : 0.25;

      mask = mask<<1 ;
    }
}
//---------------------------------------------------------------
inline void
PBIL::evaluation( const int x ,
  const int last_evaluation)
{
  if( x >= last_evaluation )
   return ;

  for( int i = 0 ; i < HALF_PBIL_LENGTH; i++)
    {
      best_bits_x[i] = bits_x[i] ;
      best_bits_y[i] = bits_y[i] ;
    }
}
//---------------------------------------------------------------
inline void
PBIL::update_x()
{
  for( int i = 0 ; i < HALF_PBIL_LENGTH; i++)
    {
      result_x[i] += ( best_bits_x[i] == 1)
? (L*(1.0 - result_x[i]))
: (L*(result_x[i] - 1.0)) ;
    }
}
//---------------------------------------------------------------
inline void
PBIL::update_y()
{
  for( int i = 0 ; i < HALF_PBIL_LENGTH; i++)
    {
      result_y[i] += ( best_bits_y[i] == 1)
? (L*(1.0 - result_y[i]))
: (L*(result_y[i] - 1.0)) ;
    }
}
//---------------------------------------------------------------

inline int
PBIL::final_output_x() const
{
  int res = 0 ;
  int bit_val = 1;

  for( int i = 0 ; i < HALF_PBIL_LENGTH; i++)
    {
      res += ( result_x[i] >= 0.5) ? bit_val : 0;
      bit_val = bit_val<<1 ;
    }
  return res ;
}
//---------------------------------------------------------------

inline int
PBIL::final_output_y() const
{
  int res = 0 ;
  int bit_val = 1;

  for( int i = 0 ; i < HALF_PBIL_LENGTH; i++)
    {
      res += ( result_y[i] >= 0.5) ? bit_val : 0;
      bit_val = bit_val<<1 ;
    }
  return res ;
}
//---------------------------------------------------------------
inline void
PBIL::mutate()
{
  int mask, scaled, rnd , sign;

  for( int i = 0 ; i < HALF_PBIL_LENGTH; i++)
    {
      rnd = get_random() ;
      mask = (1<<10) - 1;
      scaled = (rnd & mask) & 256 ;
      sign = (rnd & 512 ) ;

      if( scaled > 0)
result_x[i] += ( sign == 0) ? -MUTATION_VALUE
  : MUTATION_VALUE ;
    }

  for( int i = 0 ; i < HALF_PBIL_LENGTH; i++)
    {
      rnd = get_random() ;
      mask = (1<<10) - 1;
      scaled = (rnd & mask) & 256 ;
      sign = (rnd & 512 ) ;

      if( scaled > 0)
result_y[i] += ( sign == 0) ? -MUTATION_VALUE
  : MUTATION_VALUE ;
    }
}
//---------------------------------------------------------------
inline void
PBIL::reseed( const unsigned int seed)
{
  srand( seed );
}
//---------------------------------------------------------------
inline void
PBIL::reseed()
{
  int seed = 1 ;

  srand( seed );
}
//---------------------------------------------------------------
inline void
PBIL::next()
{
  int x = 0,y = 0 ;
  float test_bit_x = 0.0,  test_bit_y = 0.0 ;

  //do{
    for( int i = 0 ; i <  HALF_PBIL_LENGTH; i++)
      {
x = get_random() & LOCAL_MASK;
y = get_random() & LOCAL_MASK;

test_bit_x = x/( (float)(LOCAL_MASK)) ;
test_bit_y = y/( (float)(LOCAL_MASK)) ;

if(  test_bit_x <= result_x[i] )
  bits_x[i] = 1;
else bits_x[i] = 0;

if(  test_bit_y <= result_y[i] )
  bits_y[i] = 1;
else bits_y[i] = 0;

//cout << "rnd_x: " << x
//<< " rnd_y: " << y << endl;
      }

    // }while( log[extract_x()][extract_y()] == 1);
    // no point looking at the same location twice

    //log[extract_x()][extract_y()] = 1;
}
//---------------------------------------------------------------

#endif
```

C.1.3 Point.h

```
// stuff to inplement a point data class

#ifndef __POINTS__
#define __POINTS__

#include <iostream.h>
#include <fstream.h>

class point
```

```
{
public:

  point():lx(0),ly(0){};
  point(const short int xa, const short int ya):
    lx(xa), ly(ya){} ;
  point( const point &p):lx(p.lx),ly(p.ly){};
  ~point(){};

  const int x() const { return lx ;};
  const int y() const { return ly; };

  point& operator=( const point &p){ lx = p.lx ;
  ly = p.ly; return *this;};
  unsigned long int operator()( const point &q) const;
  // returns the distance between two points .
  const point operator()( const point &a,
  const point &b ) const;
  const point operator()(const short int a,
 const short int b)
  {lx = a;ly = b; return *this;};
  // compares 2 pts relative with self as reference and returns pt
  // closest.

  int operator==(const point &pt) const
  {return (( (lx == pt.lx ) && (ly == pt.ly) ) ? 1 : 0);};
  int operator!=(const point &pt) const
  { return !( *this == pt );};

  unsigned long int mod() const { return (lx*lx + ly*ly) ;};
  const point operator-( const point &a ) const ;

protected:

  short int lx, ly;

private:
  friend ostream& operator<<(ostream &o,const point &p)
  { o << p.lx << " \t" << p.ly ; return o;};

};
//-------------------------------------------------

inline
unsigned long int
point::operator()(const point &q) const
{
  int a = lx - q.lx ;
  int b = ly - q.ly ;

  return a*a + b*b ;
}
//-------------------------------------------------
inline
const point
point::operator()(const point &a, const point &b) const
{
  return ( a(*this) > b(*this)) ? a : b ;
}
//-------------------------------------------------
inline
const point
point::operator-(const point & a) const
{
  point res(lx,ly) ;
  res.lx -= a.lx ;
  res.ly -= a.ly ;
  return res ;
}
//-------------------------------------------------

#endif //__POINTS__
```

C.2 Basic Algorithms

```
//Basic algs implemanting the different motion searches

#include <iostream.h>
#include <stdlib.h>

//-------------------------------------------------

const int  M_SIZE = 8, N_SIZE = 8 , THRESHOLD_1 = 23,
  THRESHOLD_2 =345, OK = 1, INTRA = -1, DONE_PIXEL = -1 ;

const int WIDTH = 7 , MTN_WIDTH = 15;
const int SEARCH_WIDTH = 400 ;

const int MAX_OUT = 30000 ;

//-------------------------------------------------
inline
long int
MSE( const int i, const int j, const unsigned char *ref,
     const unsigned char *mtn,
     const int cols , const int rows)
{
  int k,l ,error, offset, pos_offset = i + j*cols;
  long int s_error = 0 ;
  int arg ;

  for( k = 0; k <= WIDTH ; k++)
    {
      for( l = 0 ; l <= WIDTH ; l++)
{
  if( ((i+k) < cols) && ( (j+l) < rows ) )
    {
      offset = k + l*cols ;
      arg = offset + pos_offset ;

      error = *( ref + arg ) -
*( mtn + arg ) ;
      s_error += error*error ;
    }
  else
    {
      return MAX_OUT ;
      // Block out of bounds => suppress  mtn vector...
    }
}
    }

  //cout << "x: " << i
  //<< " y: " << j
  //<< " Distortion : "
  //<< s_error <<
  // endl ;
  return s_error ;
}
inline
int MAE( const int i, const int j, const int a ,
 const int b, const unsigned char *ref,
 const unsigned char *mtn, const int cols,
 const int rows)
{
  int k,l ,error, s_error = 0;

  long int offset, mtn_offset = i + j*cols ;
  long int pos_offset = a + b*cols ;

   for( k = 0; k < 8 ; k++)
    {
      for( l = 0 ; l < 8 ; l++)
{
   if( ((i+k) < cols) && ( (j+l) < rows ) &&
       ((i+k) >= 0) && ((j+l) >= 0) )
    {
      offset = k + l*cols ;

      error = *( ref + mtn_offset + offset )
- *(mtn + pos_offset + offset ) ;

      error = ( error < 0) ? -error : error ;

      s_error += error ;
    }
   else
    {
      return MAX_OUT ;
      // Block out of bounds => suppress  mtn vector
    }
}
    }
  return s_error ;
}

inline void min( long int *minimum, long int *tmp,
 int try_x,int try_y, int *x, int *y)
{
  if( *tmp < *minimum )
    {
      *x = try_x ;
      *y = try_y ;
      *minimum = *tmp ;
    }
}
```

```
int
find_min( const int no, int *x_comp, int *y_comp,
  const unsigned char *ref,const unsigned char *mtn,
  int *best_x, int *best_y,const int rows,
  const int cols, const int r, const int c)
{
  int i;
  int minimum = MAX_OUT, tmp;

  //cout << "\nFindMIn" << endl ;

  for( i = 0; i < no; i++)
    {
      //cout << i << endl ;
      tmp = MAE( x_comp[i] + c, y_comp[i] + r, c , r,
 ref, mtn, cols, rows) ;
      //min( &minimum, &tmp, x_comp[i], y_comp[i],
      //best_x , best_y);

      if( tmp < minimum)
{
  *best_x = x_comp[i] ;
  *best_y = y_comp[i] ;
  minimum = tmp ;
}
      else if( (tmp == minimum) && (tmp != MAX_OUT))
{
  *best_x = x_comp[i] ;
  *best_y = y_comp[i] ;
}
    }
  return minimum;
}

void find_min_2( int no, int *x_comp, int *y_comp,
 const unsigned char *ref,
 const unsigned char *mtn, int *best_x,
 int *best_y, const int *x_pos_comp ,
 const int *y_pos_comp, int rows, int cols)
{
  int i, j;
  long int minimum = MAX_OUT, tmp;
  static int done[SEARCH_WIDTH][SEARCH_WIDTH]  ;

  if( ((x_comp[0] == 0)&&(y_comp[0] ==0)) ||
      /* set up @ start of block*/
      ((x_comp[no-1] == 0)&&(y_comp[no-1] ==0)))
    {
      for( i = 0; i < no ;i++)
for(j = 0; j < no ; j++) done[i][j] = DONE_PIXEL ;
    }

  for( i = 0; i < no; i++)
    {
      if( done[ x_comp[i]][ y_comp[i] ] != DONE_PIXEL)
{
  tmp = MAE( x_comp[i], y_comp[i], x_pos_comp[i],
     y_pos_comp[i], ref, mtn, cols, rows) ;

  done[x_comp[i]][ y_comp[i] ] = tmp ;

  min( &minimum, &tmp, x_comp[i],
       y_comp[i], best_x , best_y) ;
}
    }
}
//-------------------------------------------------

void
find_min_MAE( const unsigned char *ref ,
      const unsigned char *mtn ,
      int i, int j, const int rows, const int cols,
      int &x , int &y , const int subsample = 1)
{
  int offset = i + j*cols ;
  unsigned char *ref_offset = offset + (unsigned char *)*ref ,
    *offset_mtn ;

  int minimum  = MAX_OUT ;
  int next_min = 0 , value = 0;
  int current_length = 0 , next_length  = 0;

  cout << "\nMAE!\n" << flush ;

  for( int r = 0 ; r < (cols - 8) ; r++)
    {
      for( int s = 0 ; s < ( rows - 8) ; s++)
{
  offset_mtn = (unsigned char *)(*mtn) + r + s*cols ;

  for( int t = 0 ; t <= (WIDTH*subsample) ;
       t += subsample)
    {
      for( int u = 0 ; u <= (WIDTH*subsample) ;
 u += subsample)
{
  value = *(offset_mtn + t + u*cols )
    - *(ref_offset + t + u*cols) ;

  value = ( value < 0) ? -value : value ;

  next_min += value ;
}
    }

  if( next_min < minimum)// find closest motion vector
    {
      minimum = next_min ;
      x = r ;
      y = s;
    }
  else if( (next_min == minimum) &&
   (next_min != MAX_OUT) &&
   (minimum != MAX_OUT) )
    {
      current_length = (x - i)*(x - i) + ( y - j)*(y - j) ;
      next_length = (r - i)*(r - i) + (s - j)*(s - j) ;

      if( next_length < current_length)
{
  x = r;
  y = s;
}

    }
}
    }
}
//-----
void
gen_difference_block( const int x, const int y,const int a,
      const int b, const unsigned char *ref ,
      const unsigned char *mtn , int *di,
      const int cols, const int rows)
{
  int offset_a = 0, mtn_offset = x + y*cols;
  int offset_b = a + b*cols ;

  if( 1)//(x < ( cols - 7)) && ( y < (rows - 7) ) )
    {
      for( int i = 0 ; i < 8 ; i++)
{
  for( int j = 0 ; j < 8 ; j++)
    {
      offset_a = j + i*cols ;

      *(di + offset_a + offset_b) = *(mtn + offset_a + offset_b) -
*(ref + offset_a + offset_b + mtn_offset) ;
    }
}
      }
  if( ((x + a) < 0) || ( (x + a) >= ( cols )) ||
      ((y + b) < 0) || ( (y + b) >= ( rows )) )
{
  cout << "\n error gen diff block!" << endl;
  cout << "pos_x: " << a << " pos_b: " << b
       << "x: " << x << " y: " << y
       << " offset_b: " << offset_b
       << " motion offset: " << mtn_offset
       << " Distortion: "
       << MAE( x + a, y + b, a, b,(unsigned char *)ref,
       (unsigned char *)mtn, cols, rows)
       << endl ;
  //exit(1) ;
}
  //cout << "end diffing" << endl ;
}
//-------------------------------------------------------

void
fsa( const unsigned char *ref , const unsigned char *mtn ,
     int *di , int *vecs,
     const int rows , const int cols , const int subsample)
{
  int x , y ;
  int counter = 0;

  cout << "fsai\n" << flush ;

  for( int i = 0 ; i < cols ; i += 8)
```

```
    {
      for( int j = 0 ;j < rows ; j += 8)
{
  find_min_MAE( ref , mtn, i, j, rows, cols ,
x , y , subsample);
  //cout << "fsa2\n" << flush ;

  *(vecs + counter) = x - i; counter++ ;
  // Note the motion vectors..
  *(vecs + counter) = y - j; counter++ ;

  gen_difference_block( x - i ,y - j, i, j ,
ref, mtn, di, cols, rows) ;
}
    }
}
//-------------------------------------------------

const int sub_fsa_search_width = 8 ;

void
subblock_fsa(const unsigned char *ref,const unsigned char *mtn,
     int *di , int *vecs, const int rows ,
     const int cols , const int subsample)
{
  int x , y , tmp_x , tmp_y;
  int current_length = 0 , next_length = 0 ;
  int counter = 0;
  long int sum_min , minimum;

  cout << "sub fsa1\n" << flush ;

  for( int i = 0 ; i < cols ; i += 8)
    {
      for( int j = 0 ;j < rows ; j += 8)
{
  minimum = 0;
  sum_min = MAX_OUT ;
  tmp_x = 0;
  tmp_y = 0;
  x = 0 ;
  y = 0 ;

  for( int r = -sub_fsa_search_width ;
       r <= sub_fsa_search_width ;
       r++)
    for( int c = -sub_fsa_search_width ;
 c <= sub_fsa_search_width ;
 c++)
      {
tmp_y = j + r ;
tmp_x = i + c ;

if( (tmp_x >= 0) && ( tmp_x < cols) &&
    (tmp_y >= 0) && ( tmp_y < rows) ){

  minimum = MAE( tmp_x , tmp_y, i, j, ref,
 mtn, cols, rows);

  /*cout << "r:" << r << " c:"
   << c << " min:"
   << minimum << " tmp_x:"
   << tmp_x << " tmp_y"
   << tmp_y << endl ;*/

if( minimum < sum_min)
  {
    sum_min = minimum ;
    x = c;
    y = r ;
  }
else if( (minimum == sum_min)
 && (sum_min < MAX_OUT)&&
 (minimum < MAX_OUT))
 {
    current_length = x*x + y*y ;
    next_length = r*r + c*c ;

    if( current_length < next_length )
      {
x = c;
y = r;
      }
  }
}
      }

  *(vecs + counter) = x ; counter++ ;
  // Note the motion vectors..
  *(vecs + counter) = y ; counter++ ;

  gen_difference_block( x,y, i, j ,ref, mtn,
di, cols, rows) ;
  /*if( (x != 0) || ( y != 0))
    cout << "x:" << x << " y:" << y << " min:" << sum_min
 << " i:" << i << " j:" << j << endl ;*/
}
    }
}
//-------------------------------------------------

void
TTS( const unsigned char *ref ,const unsigned char *mtn ,
     int *di , int *vecs, const int rows , const int cols ,
     const int sub_sample_factor)
{
  static int x_comp[9] ;
  static int y_comp[9] ;

  int i , best_x = 0, best_y = 0;
  int w = 4 ;

  int counter = 0;

  //cerr << "rows: " << rows
  //<< "cols: " << cols << endl ;

  for( int r = 0 ; r < rows ; r += 8)
    {
      for( int c = 0 ; c < cols ; c += 8)
{
  w = 4 ;
  best_x = 0;
  best_y = 0;

  x_comp[0] = 0; y_comp[0] = w; // 0,w
  x_comp[1] = w; y_comp[1] = w; // w,w
  x_comp[2] = w; y_comp[2] = 0; // w,0
  x_comp[3] = w; y_comp[3] = -w; // w,-w
  x_comp[4] = 0; y_comp[4] = -w; // 0,-w
  x_comp[5] = -w;y_comp[5] = -w; //-w,-w
  x_comp[6] = -w;y_comp[6] = 0; //-w,0
  x_comp[7] = -w;y_comp[7] = w; // -w,w
  x_comp[8] = 0; y_comp[8] = 0; // 0,0

  //cout << "TTS " << counter <<  " r "
  //<< r << "c " << c << endl ;

  find_min( 9, x_comp, y_comp, ref, mtn,
    &best_x, &best_y, rows, cols,
    r, c) ;

  for( i = 0; i < 2; i++)
    {
      w = w/2 ;

      x_comp[0] = 0 + best_x ;
      y_comp[0] = w + best_y ;/* 0,w*/
      x_comp[1] = w + best_x ;
      y_comp[1] = w + best_y ;/* w,w */
      x_comp[2] = w + best_x ;
      y_comp[2] = 0 + best_y ;/* w,0 */
      x_comp[3] = w + best_x ;
      y_comp[3] = -w + best_y ;/* w,-w */
      x_comp[4] = 0 + best_x ;
      y_comp[4] = -w + best_y ;/* 0,-w */
      x_comp[5] = -w + best_x ;
      y_comp[5] = -w + best_y ;/*-w,-w */
      x_comp[6] = -w + best_x;
      y_comp[6] = 0 + best_y ;/*-w,0 */
      x_comp[7] = -w + best_x ;
      y_comp[7] = w + best_y;/* -w,w */
      x_comp[8] = 0 + best_x ;
      y_comp[8] = 0 + best_y; // 0,0

      find_min( 9, x_comp, y_comp, ref, mtn,
&best_x, &best_y,
rows, cols, r, c) ;
    }
  *(vecs + counter ) = best_x ; counter++ ;
  *(vecs + counter ) = best_y ; counter++ ;

  gen_difference_block( best_x , best_y ,c,r,ref,
mtn, di, cols, rows) ;
}
    }
}
//-------------------------------------------------

//One dimensional full search...
```

```
void
IDFS(const unsigned char *ref , const unsigned char *mtn,
     int *di , int *vecs, const int rows ,
     const int cols  , const int sub_sample_factor)
{
  int best_x = 0 ,best_y = 0;
  int x = 0 , y = 0 ;

  int counter = 0;
  long int tmp = MAX_OUT , minimum = MAX_OUT;

  for( int r = 0 ; r < rows ; r += 8)
    {
      for( int c = 0; c < cols ; c += 8)
{
  y = 0;// x_row
  x = 0; //RESET
  best_x = 0 ;
  best_y = 0 ;
  minimum = MAX_OUT ;

  for( int i = -7 ; i < 8 ; i++)
    {
      tmp = MAE( i + c, y + r,c, r, ref ,mtn,
 cols ,rows );

      if( tmp < minimum)
{
  x = i; minimum = tmp ;
}
    }
  for( int i = -7 ; i < 8 ; i++)
    {
      tmp = MAE( x + c, i + r, c , r, ref ,mtn,
 cols ,rows );

      if( tmp < minimum)
{
  y = i; minimum = tmp ;
}
    }
  for( int i = -3 ; i < 4 ; i++)
    {
      tmp = MAE( i + x + c, y + r, c, r, ref,
 mtn ,cols ,rows );

      if( tmp < minimum)
{
  best_x = i + x; minimum = tmp ;
}
    }
  for( int i = -3 ; i < 4 ; i++)
    {
      tmp = MAE( best_x + c, i + y + r,c, r, ref,
 mtn ,cols ,rows );

      if( tmp < minimum)
{
  best_y = i + y; minimum = tmp ;
}
    }

  *(vecs + counter ) = best_x ; counter++ ;
  // write mtn vec
  *(vecs + counter ) = best_y ; counter++ ;

  gen_difference_block( best_x , best_y ,c,r,
ref, mtn, di, cols,
rows) ;
}
    }
}
//-------------------------------------------------

void
cross_search(const unsigned char *ref,const unsigned char *mtn,
     int *di , int *vecs, const int rows ,
     const int cols  , const int sub_sample_factor)
{
  int w = 4 , x = 0 , y = 0;
  int x_comp[5] , y_comp[5] , best_x = 0 , best_y = 0 ;
  int greek_cross_x[5] , greek_cross_y[5] , st_a_x[5] ,
    st_a_y[5] ;
  long int tmp = MAX_OUT, minimum = MAX_OUT;
  int flag = 0;
  int *x_pointer = NULL, *y_pointer = NULL;

  int counter = 0;

  greek_cross_x[0] = 0; greek_cross_y[0] = 0;
  greek_cross_x[1] = -1; greek_cross_y[1] = -1;
  greek_cross_x[2] = -1; greek_cross_y[2] = 1;
  greek_cross_x[3] = 1; greek_cross_y[3] = -1;
  greek_cross_x[4] = 1; greek_cross_y[4] = 1;

  st_a_x[0] = 0; st_a_y[0] = 0;
  st_a_x[1] = -1; st_a_y[1] = 0 ;
  st_a_x[2] = 0; st_a_y[2] =  -1;
  st_a_x[3] = 1; st_a_y[3] = 0 ;
  st_a_x[4] = 0; st_a_y[4] = 1 ;

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
minimum = MAX_OUT ;
w = 4 ;  // Reset  action ...

for( int count = 0 ; count < 2 ; count++)
  {
    x_comp[0] = 0 + best_x ; y_comp[0] = 0 + best_y; //0,0
    x_comp[1] = -w + best_x; y_comp[1] = -w + best_y; //-w,-w
    x_comp[2] = -w + best_x; y_comp[2] = w + best_y; //-w,w
    x_comp[3] = w + best_x; y_comp[3] = -w + best_y; //w,-w
    x_comp[4] = w + best_x; y_comp[4] = w + best_y; //w,w

    for( int i = 0 ; i < 5 ; i++)
      {
tmp = MAE( x_comp[i] + c, y_comp[i] + r, c,
   r, ref ,mtn ,cols
   ,rows );
if( tmp < minimum )
  {
    best_x = x_comp[i] ; best_y = y_comp[i] ;
    minimum = tmp ;

    flag = ( (i==0) || (i==1) || (i==4) )
      ? 1 : 0; //End case
  }
      }
    w = w/2 ;
  }

x = 0 ;
y = 0 ; // Reset action...

if(flag == 1)
  {
    x_pointer = st_a_x ; y_pointer = st_a_y ;
  }
  else
    {
      x_pointer = greek_cross_x ;
      y_pointer = greek_cross_y ;
    }

for( int i = 0 ; i < 5 ; i++)
    {
      tmp = MAE( best_x + x_pointer[i] + c, best_y +
 y_pointer[i] + r, c, r, ref ,
 mtn ,cols ,rows );
      if( tmp < minimum )
{
  x = x_pointer[i] ; y = y_pointer[i] ;
  minimum = tmp ;
}
    }

best_x += x ;
best_y += y ;

*(vecs + counter ) = best_x ; counter++ ; // write mtn vec
*(vecs + counter ) = best_y ; counter++ ;

gen_difference_block( best_x , best_y , c,r,ref,
      mtn, di, cols,rows) ;
      }
}
//-------------------------------------------------------

void
modified_TTS(const unsigned char *ref,const unsigned char *mtn ,
     int *di , int *vecs, const int rows ,
     const int cols   , const int sub_sample_factor )
{
  int x_comp[16] , y_comp[16] , best_x = 0, best_y = 0;
  int w = 4 ;
  int counter = 0 ;
```

```
  //cerr << "rows: " << rows << "cols: " << cols << endl ;

  for( int r = 0 ; r < rows ; r += 8)
    {
      for( int c = 0 ; c < cols ; c += 8)
{
  w = 4 ;
  best_x = 0 ;
  best_y = 0;

  x_comp[0] = 0; y_comp[0] = w; // 0,w
  x_comp[1] = w; y_comp[1] = w; // w,w
  x_comp[2] = w; y_comp[2] = 0; // w,0
  x_comp[3] = w; y_comp[3] = -w; // w,-w
  x_comp[4] = 0; y_comp[4] = -w; // 0,-w
  x_comp[5] = -w;y_comp[5] = -w; //-w,-w
  x_comp[6] = -w;y_comp[6] = 0; //-w,0
  x_comp[7] = -w;y_comp[7] = w; // -w,w
  x_comp[8] = 0; y_comp[8] = 0; // 0,0

  x_comp[9] = 0 ; y_comp[9] = 1 ;
  x_comp[10] = 1; y_comp[10] = 1 ;
  x_comp[11] = 1; y_comp[11] = 0 ;
  x_comp[12] = 1; y_comp[12] = -1 ;
  x_comp[13] = 0; y_comp[13] = -1 ;
  x_comp[14] = -1; y_comp[14] = -1 ;
  x_comp[15] = -1; y_comp[15] = 0 ;
  x_comp[16] = -1; y_comp[16] = 1 ;

  //cout << "TTS\n" << endl ;

  find_min( 17, x_comp, y_comp, ref, mtn,  &best_x,
    &best_y, rows, cols , r, c) ;

  //Is the best soln close to the centre...
  if( (best_x==1) || ( best_x==-1) || (best_x==0) )
    {
      x_comp[0] = 0 + best_x; y_comp[0] = 1 + best_y;
      x_comp[1] = 1 + best_x; y_comp[1] = 1 + best_y;
      x_comp[2] = 1 + best_x; y_comp[2] = 0 + best_y;
      x_comp[3] = 1 + best_x; y_comp[3] = -1 + best_y;
      x_comp[4] = 0 + best_x; y_comp[4] = -1 + best_y;
      x_comp[5] = -1 + best_x; y_comp[5] = -1 + best_y;
      x_comp[6] = -1 + best_x; y_comp[6] = 0 + best_y;
      x_comp[7] = -1 + best_x; y_comp[7] = 1 + best_y;
      x_comp[8] = best_x ; y_comp[8] = best_y ;

      find_min( 9, x_comp, y_comp, ref, mtn,
&best_x, &best_y, rows, cols , r, c) ;
    }
  else
    { // normal TTS

      for( int i = 0; i < 2; i++)
{
  w = w/2 ;

  x_comp[0] = 0 + best_x ;
  y_comp[0] = w + best_y ;/* 0,w*/
  x_comp[1] = w + best_x ;
  y_comp[1] = w + best_y ;/* w,w */
  x_comp[2] = w + best_x ;
  y_comp[2] = 0 + best_y ;/* w,0 */
  x_comp[3] = w + best_x ;
  y_comp[3] = -w + best_y ;/* w,-w */
  x_comp[4] = 0 + best_x ;
  y_comp[4] = -w + best_y ;/* 0,-w */
  x_comp[5] = -w + best_x ;
  y_comp[5] = -w + best_y ;/*-w,-w */
  x_comp[6] = -w + best_x;
  y_comp[6] = 0 + best_y ;/*-w,0 */
  x_comp[7] = -w + best_x ;
  y_comp[7] = w + best_y;/* -w,w */
  x_comp[8] = 0 + best_x ;
  y_comp[8] = 0 + best_y; // 0,0

  find_min( 9, x_comp, y_comp, ref, mtn,
    &best_x, &best_y,  rows, cols,
    r, c) ;
}
    }
  *(vecs + counter ) = best_x ; counter++ ;
  *(vecs + counter ) = best_y ; counter++ ;

  gen_difference_block( best_x , best_y ,c,r,ref,
mtn, di, cols, rows) ;
}
    }
}
//-------------------------------------------------------------


void
twoD_log_search(const unsigned char *ref,const unsigned char *mtn,
       int *di , int *vecs, const int rows ,
       const int cols  , const int sub_sample_factor)
{
  int x_comp[9] , y_comp[9] , best_x = 0, best_y = 0,  end = 0;
  int tmp = 0, minimum = MAX_OUT;

  int counter = 0;

  for( int r = 0 ; r < rows ; r += 8)
    {
      for( int c = 0 ; c < cols ; c += 8)
{
  best_x = 0 ;
  best_y = 0;
  end = 0 ;
  minimum = MAX_OUT ;

  while( end == 0)
    {
      //new_min = MAX_OUT ;
      // Force re-examination of minimum value...

      x_comp[0] = 0 + best_x; y_comp[0] = 2 + best_y; // 0,2
      x_comp[1] = 2 + best_x; y_comp[1] = 0 + best_y ; // 2,0
      x_comp[2] = 0 + best_x; y_comp[2] = -2 + best_y; // 0,-2
      x_comp[3] = -2 + best_x;y_comp[3] = 0 + best_y ; //-2,0
      x_comp[4] = 0 + best_x; y_comp[4] = 0 + best_y; // 0,0

      for( int i = 0 ; i < 5 ; i++)
{
  if( (x_comp[i] > WIDTH) || ( x_comp[i] < -WIDTH) ||
      (y_comp[i] > WIDTH) || ( y_comp[i] < -WIDTH) )
    {
      end = 1;
      break ;// End  case..boundary
    }
  else
    {
      tmp = MAE( x_comp[i] + c, y_comp[i] + r, c,
r, ref ,mtn,cols ,rows );

      if( (i == 4 ) && ( tmp <= minimum) )
{
  end = 1 ;
  best_x = x_comp[i] ; best_y = y_comp[i] ;
  minimum = tmp ;
  break;
} // End case..centre  point .......danger

      if( tmp < minimum )
{
  best_x = x_comp[i] ; best_y = y_comp[i] ;
  minimum = tmp ;
}
    }
}
    }

  x_comp[0] = 0 + best_x; y_comp[0] = 1 + best_y; // 0,1
  x_comp[1] = 1 + best_x; y_comp[1] = 1 + best_y; // 1,1
  x_comp[2] = 1 + best_x; y_comp[2] = 0 + best_y; // 1,0
  x_comp[3] = 1 + best_x; y_comp[3] = -1 + best_y; // 1,-1
  x_comp[4] = 0 + best_x; y_comp[4] = -1 + best_y; // 0. -1
  x_comp[5] = -1 + best_x; y_comp[5] = -1 + best_y;// -1,-1
  x_comp[6] = -1 + best_x; y_comp[6] = 0 + best_y; // -1,0
  x_comp[7] = -1 + best_x; y_comp[7] = 1 + best_y; // -1,1
  x_comp[8] = best_x ; y_comp[8] = best_y ; //0,0

  find_min( 9, x_comp, y_comp, ref, mtn,  &best_x, &best_y,
    rows, cols , r, c) ; // Look at pels round pt...

  //cout << "MTNVEC" << best_x
  //<< " " << best_y << endl ;

  *(vecs + counter ) = best_x ; counter++ ;
  *(vecs + counter ) = best_y ; counter++ ;

  gen_difference_block( best_x , best_y ,c,r,ref, mtn,
di, cols,rows) ;
}
    }
}
//-------------------------------------------------------------


void
SEA(const unsigned char *ref , const unsigned char *mtn ,
```

```
    int *di , int *vecs, const int rows ,
    const int cols  , const int sub_sample_factor)
{
  int best_x = 0, best_y = 0;
  int offset , offset_p;
  int R = 0;
  int old_MAE = MAX_OUT , new_MAE = MAX_OUT ;

  long int dist_old , dist_new ;//<><><><

  unsigned int counter = 0;

  const int new_cols = cols - 8 , new_rows = rows - 8 ;

  //static int data[300000] ;
  static int sum_norm[300000] ;

  /*
    //First col...
  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += *( ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...
  for( int j = 0 ; j < rows ; j++)
    {
      for( int i = 1 ; i <= new_cols ; i++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  -
  *(ref + offset - 1)
    + *( ref + offset + 7) ;
}
    }
  //first row ..... col wise.....
  for( l = 0 ; l <= new_cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += data[ k*cols + l];

      sum_norm[l] = sum;
    }

 // Rest of the rows......col wise
  const int seven_cols_up = 7*cols ;

  for( int i = 0 ; i <= new_cols  ; i++)
    {
      for( int j = 1 ; j <= new_rows  ; j++)
{
  offset = i + j*cols ;

  sum_norm[offset] = sum_norm[offset - cols]
    - data[offset - cols] + data[offset + seven_cols_up ] ;
}
    }
  //---
  */

  for( int j = 0 ; j <= new_rows ; j++)
    for( int i = 0 ; i <= new_cols ; i++)
      {
sum_norm[ i + j*cols] = 0 ;

for( int k = 0 ; k < 8 ; k++)
  for( int l = 0 ; l < 8 ; l++)
    {
      sum_norm[i + j*cols] += *(ref + i + j*cols + l +
k*cols);

      if( ((j + k) > 511) || ((i + l) < 0) ||
  ((j + k) < 0) || (( i + l) > 511 ))
{
  cout << "ERROR YYYYYYYYYYYYYYYYYY" << endl ;
}
    }
      }


  for( int r = 0 ; r < rows ; r += 8)
    for( int c = 0 ; c < cols ; c += 8)
      {
offset = c + r*cols ;

// Calc R
R = 0 ; // Reset R.

for( int m = 0 ; m < 8 ; m++)
  for( int n = 0 ; n < 8 ; n++)
    R += *(mtn + offset + m + n*cols );

//---Find mtn vectors----

old_MAE = MAE( c , r , c , r , ref, mtn, cols, rows);
//sum_norm[0] ; // Initial estimate
//old_MAE = ( old_MAE > 0) ? old_MAE : -old_MAE ;

best_x = 0;
best_y = 0;
dist_old = rows*cols;//<><><><

/*if( sum_norm[offset] != R)
  {
    cout << sum_norm[offset] << "\t" <<
    R << "\t" << c << "\t"
 << r << endl ;

  }*/

//cout << r << "\t" << c << endl;

for( int u = 0 ; u <= new_rows; u++ ) // Orig -7 to 8
  for( int v = 0 ; v <= new_cols ; v++) // Orig -7 to 8
    {
      offset_p = v + u*cols ; // Offset fot the data!!!!!!!...

      if( ((R - old_MAE) <= sum_norm[offset_p] ) &&
((R + old_MAE) >= sum_norm[offset_p] ) )
{
  /*new_MAE = R - sum_norm[ offset_p ] ;
      new_MAE = ( new_MAE > 0 ) ? new_MAE : -new_MAE ;*/

  new_MAE = MAE( v , u , c, r, ref, mtn,
 cols, rows);

  if( new_MAE < old_MAE )
    {
      best_x = v - c;
      best_y = u - r;

      old_MAE = new_MAE;

      dist_old = best_x*best_x + best_y*best_y;
      //<><><><
    }
  else if( new_MAE == old_MAE) // Ensure min dist...
  /*((R - old_MAE) == sum_norm[offset_p] ) &&
  ((R + old_MAE) == sum_norm[offset_p] ) )*/
    {
      dist_new = (v - c)*(v - c ) + (u - r)*(u - r) ;

      if( dist_new < dist_old)
{
  best_x = v - c;
  best_y = u - r;

  dist_old = dist_new;
}
    }
}
      //}
    }
*(vecs + counter ) = best_x ; counter++ ;
*(vecs + counter ) = best_y ; counter++ ;

gen_difference_block( best_x , best_y ,c,r,ref, mtn,
      di, cols,rows) ;

      }
}
//-----------------------------------------------------

void
conjugate_search(const unsigned char *ref,const unsigned char *mtn,
 int *di , int *vecs, const int rows ,
 const int cols, const int sub_sample_factor)
{
  int best_x = 0 , best_y = 0 ;
  int end_x = 0, end_y = 0, offset_x = 0, offset_y = 0;

  long int x , y , z ;
```

```
  int counter = 0;

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
offset_x = 1;
offset_y = 0;
end_x = 0;
end_y = 0;

x = MAE( c - 1, r, c, r, ref ,mtn , cols ,rows ) ;
y = MAE( c    , r, c, r, ref ,mtn , cols ,rows ) ;
z = MAE( c + 1, r, c, r, ref ,mtn , cols ,rows ) ;

while( end_x == 0 ) // x-component...
  {
    if( x < y )
      {
z = y ;
y = x ;

best_x-- ;

x = MAE( best_x - 1 + c, r,  c, r, ref ,mtn,
 cols ,rows ) ;
      }
    else if( z < y)
      {
x = y;
y = z;

best_x++ ;

z = MAE( best_x + 1 + c, r,  c, r, ref ,mtn,
 cols ,rows ) ;
      }
    else
      {
end_x = 1;

offset_y = 1 ;
offset_x = 0 ;
      }

    if( (best_x < -7) || ( best_x > 7) )
      // Check boundary cdns....
      {
best_x = (best_x < -7) ? -7 : 7 ;

end_x = 1;
offset_x = 0;
offset_y = 1 ;
      }

    //cout << best_x << " " << best_y
    //<< " " << r << " " << c
    //  << endl ;
  }

offset_x = 0 ;
offset_y = 1 ;

x = MAE(best_x + c, r - 1,c, r, ref ,mtn, cols,rows) ;
y = MAE(best_x + c, r    ,c, r, ref ,mtn, cols,rows) ;
z = MAE(best_x + c, r + 1,c, r, ref ,mtn, cols,rows) ;

while( end_y == 0 ) // y-component...
  {
    if( x < y )
      {
z = y ;
y = x ;

best_y -= 1 ;

x = MAE( best_x + c, best_y - 1 + r, c, r,
 ref ,mtn ,cols ,rows );
      }
    else if( z < y)
      {
x = y;
y = z;

best_y += 1;

z = MAE( best_x + c, best_y + 1 + r, c, r,
 ref ,mtn ,cols ,rows );
      }
    else
      {
end_y = 1;

offset_y = 0 ;
offset_x = 0 ;
      }

     // Check boundary cdns....

    if( (best_y < -7) || ( best_y > 7) )
      {
best_y = (best_y < -7) ? -7 : 7 ;
end_y = 1;
offset_y = 0;
      }
    //cout << best_x << " " << best_y
    //<< " " << r << " " << c
    //  << endl ;
  }
*(vecs + counter ) = best_x ; counter++ ;
*(vecs + counter ) = best_y ; counter++ ;

gen_difference_block( best_x , best_y ,c,r,ref, mtn,
      di, cols,rows) ;
      }
}

//------------------------------------------------------
void
modified_conjugate_search(const unsigned char *ref ,
  const unsigned char *mtn ,
  int *di , int *vecs, const int rows ,
  const int cols,const int sub_sample_factor)
{
  int minimum = MAX_OUT , tmp , best_x = 0 , best_y = 0 ;
  int counter = 0;
  int length = 0 , new_length = 0 ;

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
minimum = MAX_OUT ;

for( int i = -7 ; i < 8 ; i++)
  {
    tmp = MAE( i + c, r,  c, r, ref ,mtn ,cols ,rows ) ;
    if( tmp < minimum )
      {
best_x = i ;
minimum = tmp;
      }
    else if( (tmp == minimum) && ( tmp < MAX_OUT))
      {
length = best_x*best_x ;
new_length = i*i ;

if( new_length < length )
  {
    best_x = i ;
  }
      }
    /*cout << "1x:" << best_x << " y:"
      //<< best_y << " min:" << min
 << " c:" << c << " r:" << r << endl ;;;*/
  }

minimum = MAX_OUT ; // Reset ...
for( int j = -7 ; j < 8 ; j++)
  {
    tmp = MAE( c + best_x , r + j,  c, r, ref ,mtn ,cols
       ,rows ) ;
    if( tmp < minimum )
      {
best_y = j ;
minimum = tmp;
      }
    else if( (tmp == minimum) && ( tmp < MAX_OUT))
      {
length = best_y*best_y ;
new_length = j*j ; // yes this is right!

if( new_length < length )
  {
    best_y = j ;
  }
      }
    /*cout << "2x:" << best_x << " y:"
```

```
      //<< best_y << " min:" << min
       << " c:" << c << " r:" << r << endl ;;;*/
  }
*(vecs + counter ) = best_x ; counter++ ;
*(vecs + counter ) = best_y ; counter++ ;

gen_difference_block( best_x , best_y ,c,r,ref, mtn,
      di, cols,rows) ;

/*if( (best_x != 0) || ( best_y != 0))
  cout << "written x:" << best_x << " y:"
  //<< best_y << " min:" << min
       << " c:" << c << " r:" << r << endl ;;;*/
      }
}

//-------------------------------------------------

void
PHODS(const unsigned char *ref , const unsigned char *mtn ,
      int *di , int *vecs, const int rows ,
      const int cols  , const int sub_sample_factor)
{
  int best_x = 0 , best_y = 0;
  int S = 4 ; // See article
  const int log_2_p = 2 ; // See article
  unsigned int x1, x2, x3 , y1, y2, y3 ;
  int counter = 0;

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;

for( int i = 0 ; i < ( 1 + log_2_p) ; i++)
  {
    //--
    x1 = MAE(c + best_x - S,r,c,r,ref,mtn,cols,rows);
    x2 = MAE(c + best_x    ,r,c,r,ref,mtn,cols,rows);
    x3 = MAE(c + best_x + S,r,c,r,ref,mtn,cols,rows);

    best_x = ((x1<x2) && (x1<x3)) ?(best_x - S):best_x;
    best_x = ((x3<x2) && (x3<x1)) ?(best_x + S):best_x;
    //--

    y1 = MAE(c,r + best_y - S,c,r,ref,mtn,cols,rows);
    y2 = MAE(c,r + best_y    ,c,r,ref,mtn,cols,rows);
    y3 = MAE(c,r + best_y + S,c,r,ref,mtn,cols,rows);

    best_y = ((y1<y2) && (y1<y3)) ?(best_y - S):best_y;
    best_y = ((y3<y2) && (y3<y1)) ?(best_y + S):best_y;

    //--

    S = S/2 ;
  }

*(vecs + counter ) = best_x ; counter++ ;
*(vecs + counter ) = best_y ; counter++ ;
gen_difference_block( best_x , best_y ,c,r,ref, mtn,
      di, cols,rows) ;
      }
}
//-------------------------------------------------

int plus_MAE( const int x, const int y, const int c, const int r,
      const unsigned char *ref , const unsigned char *mtn,
      const int cols, const int rows)
{
  const int H = 3 ; // n - 1 see paper

  int result = 0 , tmp , mtn_offset , offset ;
  int h_offset;

  if( (x < 0) || ( x > (cols - 8)) ||
      (y < 0) || ( y > (rows - 8)))
    return MAX_OUT;

  h_offset = H*cols ;
  mtn_offset = x + y*cols ;
  offset = c + r*cols ;

  for( int a = 0 ; a < 8 ; a++)
    {
      tmp =  *( ref + mtn_offset + h_offset + a) -
*(mtn + offset + h_offset + a) ;

      result += ( tmp < 0) ? -tmp : tmp ;
    }

  for( int b = 0 ; b < 8 ; b++)
    {
      tmp =  *( ref + mtn_offset + H + b*cols) -
*(mtn + offset  + H + b*cols) ;

      result += ( tmp < 0) ? -tmp : tmp ;
    }

  return result ;
}

void
PPS(const unsigned char *ref , const unsigned char *mtn ,
      int *di , int *vecs, const int rows ,
      const int cols  , const int sub_sample_factor)
{
  int best_x , best_y;
  long int old_MAE , new_MAE;

  int counter = 0 ;

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
old_MAE = MAX_OUT;

//cout << c << " " << r << endl ;

for( int i = -sub_fsa_search_width + c;
     i < ( sub_fsa_search_width + c);
     i++)
  for( int j = -sub_fsa_search_width + r ;
       j < ( sub_fsa_search_width + r);
       j++)
    {
      new_MAE = plus_MAE(i,j,c,r,ref,mtn,cols,rows);

      if( new_MAE < old_MAE)
{
  best_x = i - c;
  best_y = j - r;

  old_MAE = new_MAE;
}else if( old_MAE == new_MAE)
  {
    best_x = i - c;
    best_y = j - r;
  }
    }

*(vecs + counter ) = best_x ; counter++ ;
*(vecs + counter ) = best_y ; counter++ ;
gen_difference_block( best_x , best_y ,c,r,ref, mtn,
      di, cols,rows) ;
      }
}
//-------------------------------------------------------

int cross_MAE(const int x, const int y,const int c,const int r,
      const unsigned char *ref ,const unsigned char *mtn,
      const int cols, const int rows)
{
  const int H = 7 ; // see paper

  int result = 0 , tmp , mtn_offset , offset ;

  if( (x < 0) || ( x > (cols - 8)) ||
      (y < 0) || ( y > (rows - 8)))
    return MAX_OUT;

  mtn_offset = x + y*cols ;
  offset = c + r*cols ;

  for( int a = 0 ; a < H ; a++)
    {
      tmp =  *( ref + mtn_offset + a + a*cols) -
*(mtn + offset + a + a*cols) ;

      result += ( tmp < 0) ? -tmp : tmp ;

      tmp =  *( ref + mtn_offset + H - a + a*cols) -
*(mtn + offset + H - a + a*cols) ;

      result += ( tmp < 0) ? -tmp : tmp ;
    }
  return result ;
}
```

```
void
CPS(const unsigned char *ref , const unsigned char *mtn ,
      int *di , int *vecs, const int rows ,
      const int cols  , const int sub_sample_factor)
{
  int best_x , best_y;
  long int old_MAE , new_MAE;
  int counter = 0 ;

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
old_MAE = MAX_OUT;

//cout << c << " " << r << endl ;

for( int i = -sub_fsa_search_width + c;
     i < (sub_fsa_search_width + c);
     i++)
  for( int j = -sub_fsa_search_width + r;
       j < (sub_fsa_search_width + r);
       j++)
    {
      new_MAE = cross_MAE( i,j,c,r,ref,mtn,cols,rows );

      if( new_MAE < old_MAE)
{
  best_x = i - c;
  best_y = j - r;

  old_MAE = new_MAE;
}else if( old_MAE == new_MAE)
  {
    best_x = i - c;
    best_y = j - r;
  }
    }

*(vecs + counter ) = best_x ; counter++ ;
*(vecs + counter ) = best_y ; counter++ ;
gen_difference_block( best_x , best_y ,c,r,ref, mtn,
      di, cols,rows) ;
      }
}
//-----------------------------------------------------

void
Four_SS(const unsigned char *ref , const unsigned char *mtn ,
int *di , int *vecs, const int rows ,
const int cols  , const int sub_sample_factor)
{
  int best_x , best_y;
  int x_comp[9] ;
  int y_comp[9] ;
  int w ;
  int counter = 0 ;

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
w = 1;

do{
x_comp[0] = -2 + best_x ; y_comp[0] = -2 + best_y ;
x_comp[1] =  0 + best_x ; y_comp[1] = -2 + best_y ;
x_comp[2] =  2 + best_x ; y_comp[2] = -2 + best_y ;

x_comp[3] = -2 + best_x ; y_comp[3] = 0 + best_y ;
x_comp[4] =  0 + best_x ; y_comp[4] = 0 + best_y ;
x_comp[5] =  2 + best_x ; y_comp[5] = 0 + best_y ;

x_comp[6] = -2 + best_x ; y_comp[6] = 2 + best_y ;
x_comp[7] =  0 + best_x ; y_comp[7] = 2 + best_y ;
x_comp[8] =  2 + best_x ; y_comp[8] = 2 + best_y ;

find_min( 9,x_comp, y_comp, ref, mtn,&best_x,&best_y,
  rows, cols , r, c) ;

w++;
}while(w < 4);

x_comp[0] = -1 + best_x ; y_comp[0] = -1 + best_y ;
x_comp[1] =  0 + best_x ; y_comp[1] = -1 + best_y ;
x_comp[2] =  1 + best_x ; y_comp[2] = -1 + best_y ;
x_comp[3] = -1 + best_x ; y_comp[3] = 0 + best_y ;
x_comp[4] =  0 + best_x ; y_comp[4] = 0 + best_y ;
x_comp[5] =  1 + best_x ; y_comp[5] = 0 + best_y ;

x_comp[6] = -1 + best_x ; y_comp[6] = 1 + best_y ;
x_comp[7] =  0 + best_x ; y_comp[7] = 1 + best_y ;
x_comp[8] =  1 + best_x ; y_comp[8] = 1 + best_y ;

find_min(9, x_comp,y_comp,ref,mtn,&best_x, &best_y,
  rows, cols , r, c) ;

*(vecs + counter ) = best_x ; counter++ ;
*(vecs + counter ) = best_y ; counter++ ;
gen_difference_block( best_x , best_y ,c,r,ref, mtn,
      di, cols,rows) ;
      }
}
//-----------------------------------------------------
void
new_DMD(const unsigned char *ref , const unsigned char *mtn ,
int *di , int *vecs, const int rows ,
const int cols  , const int sub_sample_factor)
{
  int best_x , best_y, w;
  long int old_MAE , new_MAE;
  int counter = 0 ;
  int x_comp[9] , y_comp[9] ;


  const threshold = 32 ;

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
old_MAE = MAX_OUT;
w = 4;

do{
new_MAE = MAE( c + best_x ,r + best_y, c, r,
       ref ,mtn ,cols ,rows) ;
old_MAE = new_MAE ;

if( new_MAE < threshold) break;

x_comp[0] = -w + best_x ; y_comp[0] =      best_y ;
x_comp[1] =      best_x ; y_comp[1] =  w + best_y ;
x_comp[2] =  w + best_x ; y_comp[2] =      best_y ;
x_comp[3] =      best_x ; y_comp[3] = -w + best_y ;

new_MAE = find_min( 4, x_comp, y_comp, ref, mtn,
    &best_x, &best_y,
    rows, cols , r, c) ;

if( new_MAE < threshold) break;

if( new_MAE <= old_MAE ){

//---
x_comp[0] = -w + best_x ; y_comp[0] =      best_y ;
x_comp[1] =      best_x ; y_comp[1] =  w + best_y ;
x_comp[2] =  w + best_x ; y_comp[2] =      best_y ;
x_comp[3] =      best_x ; y_comp[3] = -w + best_y ;

new_MAE = find_min( 4, x_comp, y_comp, ref, mtn,
    &best_x, &best_y,
    rows, cols , r, c) ;

if( new_MAE < threshold) break;
}

w = w/2;
}while( w > 0);


*(vecs + counter ) = best_x ; counter++ ;
*(vecs + counter ) = best_y ; counter++ ;
gen_difference_block( best_x , best_y ,c,r,ref, mtn,
      di, cols,rows) ;
      }
}
//-----------------------------------------------------
void
CDS(const unsigned char *ref , const unsigned char *mtn ,
    int *di , int *vecs, const int rows ,
    const int cols  , const int sub_sample_factor)
{
  int best_x = 0 , best_y = 0 ;
  int end_x = 0, end_y = 0, end_slant,offset_x = 0, offset_y = 0;
```

```
  long int x , y , z ;
  int counter = 0;

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
offset_x = 1;
offset_y = 0;
end_x = 0;
end_y = 0;
end_slant = 0;

x = MAE( c - 1, r, c, r, ref ,mtn , cols ,rows ) ;
y = MAE( c    , r, c, r, ref ,mtn , cols ,rows ) ;
z = MAE( c + 1, r, c, r, ref ,mtn , cols ,rows ) ;

while( end_x == 0 ) // x-component...
  {
    if( x < y )
      {
z = y ;
y = x ;

best_x-- ;

x = MAE( best_x - 1 + c, r,  c, r, ref,
 mtn , cols ,rows ) ;
      }
    else if( z < y)
      {
x = y;
y = z;

best_x++ ;

z = MAE( best_x + 1 + c, r,  c, r, ref,
 mtn , cols ,rows ) ;
      }
    else
      {
end_x = 1;

offset_y = 1 ;
offset_x = 0 ;
      }

    if( (best_x < -7) || ( best_x > 7) )
      // Check boundary cdns....
      {
best_x = (best_x < -7) ? -7 : 7 ;

end_x = 1;
offset_x = 0;
offset_y = 1 ;
      }

    //cout << best_x << " " << best_y <<
    //" " << r << " " << c
    //  << endl ;
  }

offset_x = 0 ;
offset_y = 1 ;

x = MAE( best_x + c, r - 1,  c, r, ref,
 mtn , cols ,rows) ;
y = MAE( best_x + c, r    ,  c, r, ref,
 mtn , cols ,rows ) ;
z = MAE( best_x + c, r + 1,  c, r, ref,
 mtn , cols ,rows ) ;

while( end_y == 0 ) // y-component...
  {
    if( x < y )
      {
z = y ;
y = x ;

best_y -= 1 ;

x = MAE( best_x + c, best_y - 1 + r, c, r,
 ref ,mtn ,cols ,rows );
      }
    else if( z < y)
      {
x = y;
y = z;

best_y += 1;

z = MAE( best_x + c, best_y + 1 + r, c, r,
 ref ,mtn ,cols ,rows );
      }
    else
      {
end_y = 1;

offset_y = 0 ;
offset_x = 0 ;
      }

     // Check boundary cdns....

    if( (best_y < -7) || ( best_y > 7) )
      {
best_y = (best_y < -7) ? -7 : 7 ;
end_y = 1;
offset_y = 0;
      }
    //cout << best_x << " " << best_y
    //<< " " << r << " " << c
    //  << endl ;
  }

if( (best_x != 0) && (best_y != 0)){

  x = MAE(best_x + c - 1, r - 1 + best_y,
  c, r,ref ,mtn ,cols ,rows);
  y = MAE(best_x + c    , r     + best_y,
  c, r,ref ,mtn ,cols ,rows);
  z = MAE(best_x + c + 1, r + 1 + best_y,
  c, r,ref ,mtn ,cols ,rows);

while( end_slant == 0 ) // slant-component...
  {
    if( x < y )
      {
z = y ;
y = x ;

best_y -= 1 ;
best_x -= 1;

x = MAE( best_x + c - 1,
 best_y - 1 + r, c, r,
 ref ,mtn ,cols ,rows );
      }
    else if( z < y)
      {
x = y;
y = z;

best_y += 1;
best_x += 1;

z = MAE( best_x + c + 1,
 best_y + 1 + r, c, r,
 ref ,mtn ,cols ,rows );
      }
    else
      {
end_slant = 1;

offset_y = 0 ;
offset_x = 0 ;
      }

     // Check boundary cdns....

    if( (best_y < -7) || ( best_y > 7) ||
(best_x < -7) || ( best_x > 7))
      {
best_y = (best_y < -7) ? -7 : 7 ;
best_x = (best_x < -7) ? -7 : 7 ;

end_y = 1;
offset_y = 0;
      }
  }
}

*(vecs + counter ) = best_x ; counter++ ;
*(vecs + counter ) = best_y ; counter++ ;

gen_difference_block( best_x , best_y ,c,r,ref, mtn,
      di, cols,rows) ;
      }
```

```
}

//-------------------------------------------------
void
BBGDS(const unsigned char *ref , const unsigned char *mtn ,
      int *di , int *vecs, const int rows ,
      const int cols  , const int sub_sample_factor)
{
  int best_x = 0 , best_y = 0 , tx, ty;
  int new_MAE , old_MAE , stop;

  int counter = 0;

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
tx = 0;
ty = 0;
old_MAE = MAX_OUT ;
stop = 0;

// cout << r << " " << c << endl ;

//while( stop == 0){

for( int qq = 0 ; qq < 7; qq++){

  tx = 0;
  ty = 0;

  if( stop == 0)
    {

      for( int yy = -1 ; yy < 2 ; yy++)
{
  /*if( ((yy + best_y) < -7 ) || ( (yy + best_y)> 7))
      {
      stop = 1;
      continue;
      }*/

  for( int xx = -1 ; xx < 2 ; xx++)
    {
      /*if( ((xx + best_x) < -7 ) || ( (xx + best_x) > 7))
{
  stop = 1;
  continue;
}*/

      new_MAE = MAE(best_x + c + xx,best_y + r + yy,
    c, r, ref ,mtn , cols ,rows) ;

      if( new_MAE < old_MAE )
{
  tx = xx;
  ty = yy;
  old_MAE = new_MAE ;

  if( (xx == 0) && ( yy == 0))
    { stop = 2; break;}
}
    }
  if( stop == 2) break; // cntr found best match.
}
    }else break;

  best_x = best_x + tx;
  best_y = best_y + ty;
}

*(vecs + counter ) = best_x ; counter++ ;
*(vecs + counter ) = best_y ; counter++ ;

gen_difference_block( best_x , best_y ,c,r,ref, mtn,
      di, cols,rows) ;
      }
}
//-------------------------------------------------
void
DSWA_IS(const unsigned char *ref , const unsigned char *mtn ,
int *di , int *vecs, const int rows ,
const int cols  , const int sub_sample_factor)
{
  int best_x = 0 , best_y = 0 ;
  int new_MAE , old_MAE , second_last = MAX_OUT;

  const float T_l = 0.3;
  const float T_h = 0.6; // see article

  float G ;

  int x_comp[5] ;
  int y_comp[5] ;
  int w ;

  int counter = 0;

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
old_MAE = MAX_OUT ;
w = 4;

//cout << c << " " << r << endl;

while( w > 1){ cout << w << endl ;

x_comp[0] =      best_x ; y_comp[0] =      best_y ;
x_comp[1] =  w + best_x ; y_comp[1] =  w + best_y ;
x_comp[2] =  w + best_x ; y_comp[2] = -w + best_y ;
x_comp[3] = -w + best_x ; y_comp[3] =  w + best_y ;
x_comp[4] = -w + best_x ; y_comp[4] = -w + best_y ;

for( int i = 0 ; i < 5 ; i++)
  {
    new_MAE = MAE(best_x + c , best_y + r ,
  c, r, ref ,mtn , cols ,rows) ;

    if( new_MAE < old_MAE )
      {
second_last = old_MAE;
old_MAE = new_MAE ;
      }
  }

G = (second_last - old_MAE)/second_last ;

if( G > T_h) w =  w/4 ;
else
  if( ( G <= T_h) && ( G >= T_l)) w = w/2;
else
  w = ( w*3)/4 ;

x_comp[0] =      best_x ; y_comp[0] =      best_y ;
x_comp[1] =  w + best_x ; y_comp[1] =      best_y ;
x_comp[2] = -w + best_x ; y_comp[2] =      best_y ;
x_comp[3] =      best_x ; y_comp[3] =  w + best_y ;
x_comp[4] =      best_x ; y_comp[4] = -w + best_y ;

for( int i = 0 ; i < 5 ; i++)
  {
    new_MAE = MAE(c + x_comp[i], r + y_comp[i] ,
  c, r, ref ,mtn , cols ,rows) ;

    if( new_MAE < old_MAE )
      {
best_x = x_comp[i];
best_y = y_comp[i];
second_last = old_MAE;
old_MAE = new_MAE ;
      }
  }

G = (second_last - old_MAE)/second_last ;

if( G > T_h) w =  w/4 ;
else
  if( ( G <= T_h) && ( G >= T_l)) w = w/2;
else
  w = (w*3)/4 ;
}

for( int i = -1 ; i < 2 ; i++)
  for( int j = -1 ; j < 2; j++)
    {
      new_MAE = MAE(c + best_x + i,
    r + best_y + j,
    c, r, ref ,mtn , cols ,rows) ;

    if( new_MAE < old_MAE )
      {
best_x = best_x + i;
best_y = best_y + j;
old_MAE = new_MAE ;
      }
    }
```

```
*(vecs + counter ) = best_x ; counter++ ;
*(vecs + counter ) = best_y ; counter++ ;

gen_difference_block( best_x , best_y ,c,r,ref, mtn,
      di, cols,rows) ;
      }
}
//-------------------------------------------------------

void iterate_TSS_step( const int w, const int index,
       int best_x[], int best_y[],
       int second_best_x[], int second_best_y[],
       int old_MAE[], int second_old_MAE[],
       const unsigned char *ref,
       const unsigned char *mtn,
       const int rows, const int cols,
       const int r, const int c)
{
  int x_comp[9] , y_comp[9] ;
  int new_MAE;

  x_comp[0] = 0 + best_x[index] ;
  y_comp[0] = w + best_y[index] ;/* 0,w*/

  x_comp[1] = w + best_x[index] ;
  y_comp[1] = w + best_y[index] ;/* w,w */

  x_comp[2] = w + best_x[index] ;
  y_comp[2] = 0 + best_y[index] ;/* w,0 */

  x_comp[3] =  w + best_x[index] ;
  y_comp[3] = -w + best_y[index] ;/* w,-w */

  x_comp[4] =  0 + best_x[index] ;
  y_comp[4] = -w + best_y[index] ;/* 0,-w */

  x_comp[5] = -w + best_x[index]   ;
  y_comp[5] = -w + best_y[index] ;/*-w,-w */

  x_comp[6] = -w + best_x[index] ;
  y_comp[6] = 0 + best_y[index]  ;/*-w,0 */

  x_comp[7] = -w + best_x[index] ;
  y_comp[7] =  w + best_y[index] ;/* -w,w */

  x_comp[8] = 0 + best_x[index] ;
  y_comp[8] = 0 + best_y[index] ; // 0,0

  best_x[index + 1] = 0;
  best_y[index + 1] = 0;
  second_best_x[index + 1] = 0;
  second_best_y[index + 1] = 0;
  old_MAE[index + 1] = old_MAE[index ];

  for( int i = 0 ; i < 9 ; i++)
    {
      new_MAE = MAE( x_comp[i], y_comp[i], c,r,
     ref,mtn,cols,rows);

      if( new_MAE <= old_MAE[index + 1])
{
  second_old_MAE[index + 1] = old_MAE[index + 1];
  second_best_x[index + 1] = best_x[index + 1];
  second_best_y[index + 1] = best_x[index + 1];

  old_MAE[index + 1] = new_MAE;
  best_x[index + 1] = x_comp[i] ;
  best_y[index + 1] = y_comp[i] ;
}
    }
}

void
Improved_TSS(const unsigned char *ref,
     const unsigned char *mtn ,
     int *di , int *vecs, const int rows ,
     const int cols  , const int sub_sample_factor)
{
  int best_x = 0 , best_y = 0 ;
  int old_MAE,second_old_MAE,second_best_x,second_best_y ;;
  int new_MAE;

  int x_comp[9], y_comp[9] , w;
  int counter = 0;

  int log_x[5] , log_y[5] , log_MAE[5] ;

  const int x_offset[5] = { 0, -7, -7, 7, 7 };
  const int y_offset[5] = { 0,  7, -7, 7,-7 };

  int note_best_x[8], note_best_y[8], note_second_best_x[8],
    note_second_best_y[8] , note_old_MAE[8],
    note_second_old_MAE[8];

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
second_best_x = 0 ;
second_best_y = 0 ;
old_MAE = MAX_OUT ;
second_old_MAE = MAX_OUT ;
w = 4;

//cout << c << " " << r << endl;

for( int q = 0 ; q < 5 ; q++){
  w = 4;

  x_comp[0] =  0 + x_offset[q];
  y_comp[0] =  w + y_offset[q]; // 0,w
  x_comp[1] =  w + x_offset[q];
  y_comp[1] =  w + y_offset[q]; // w,w
  x_comp[2] =  w + x_offset[q];
  y_comp[2] =  0 + y_offset[q]; // w,0
  x_comp[3] =  w + x_offset[q];
  y_comp[3] = -w + y_offset[q]; // w,-w
  x_comp[4] =  0 + x_offset[q];
  y_comp[4] = -w + y_offset[q]; // 0,-w
  x_comp[5] = -w + x_offset[q];
  y_comp[5] = -w + y_offset[q]; //-w,-w
  x_comp[6] = -w + x_offset[q];
  y_comp[6] =  0 + y_offset[q]; //-w,0
  x_comp[7] = -w + x_offset[q];
  y_comp[7] =  w + y_offset[q]; // -w,w
  x_comp[8] =  0 + x_offset[q];
  y_comp[8] =  0 + y_offset[q]; // 0,0

  //cout << "TTS " << counter <<
  //" r " << r << "c " << c << endl ;

  /*old_MAE = find_min( 9, x_comp, y_comp, ref, mtn,
    &best_x, &best_y,
    rows, cols, r, c) ;*/

  note_best_x[0] = 0;
  note_best_y[0] = 0;
  note_second_best_x[0] = 0;
  note_second_best_y[0] = 0;
  note_old_MAE[0] = MAX_OUT;
  note_second_old_MAE[0] = MAX_OUT ;

  for( int i = 0 ; i < 9 ; i++)
    {
      new_MAE = MAE( x_comp[i] + c, y_comp[i] + r,
     c,r ,ref,mtn,cols,rows);

      if( new_MAE < note_old_MAE[0])
{
  note_second_old_MAE[0] = note_old_MAE[0];
  note_second_best_x[0] = note_best_x[0];
  note_second_best_y[0] = note_best_x[0];

  note_old_MAE[0] = new_MAE;
  note_best_x[0] = x_comp[i] ;
  note_best_y[0] = y_comp[i] ;
}
    }


  /*for( int i = 0; i < 2; i++)
    {
      w = w/2 ;*/
  w = 2;
      /*iterate_TSS_step( w, i, note_best_x, best_y,
note_second_best_x,
note_second_best_y,
note_old_MAE, note_second_old_MAE,
ref, mtn,  rows, cols,  r, c);*/

  x_comp[0] = 0 + note_best_x[0] ;
  y_comp[0] = w + note_best_y[0] ;// 0,w

  x_comp[1] = w + note_best_x[0] ;
  y_comp[1] = w + note_best_y[0] ;// w,w

  x_comp[2] = w + note_best_x[0] ;
  y_comp[2] = 0 + note_best_y[0] ;// w,0
```

```
  x_comp[3] =  w + note_best_x[0] ;
  y_comp[3] = -w + note_best_y[0] ;// w,-w

  x_comp[4] =  0 + note_best_x[0] ;
  y_comp[4] = -w + note_best_y[0] ;// 0,-w

  x_comp[5] = -w + note_best_x[0] ;
  y_comp[5] = -w + note_best_y[0] ;//-w,-w

  x_comp[6] = -w + note_best_x[0] ;
  y_comp[6] = 0 + note_best_y[0]  ;//-w,0

  x_comp[7] = -w + note_best_x[0] ;
  y_comp[7] =  w + note_best_y[0] ;// -w,w

  x_comp[8] = 0 + note_best_x[0] ;
  y_comp[8] = 0 + note_best_y[0] ; // 0,0

      /*old_MAE = find_min( 9, x_comp, y_comp, ref, mtn,
  &best_x, &best_y,
  rows, cols, r, c) ;*/
  note_old_MAE[1] = MAX_OUT;
  note_best_x[1] = 0;
  note_best_y[1] = 0;

  for( int i = 0 ; i < 9 ; i++)
    {
      new_MAE = MAE( x_comp[i] + c, y_comp[i] + r,
     c,r ,ref,mtn,cols,rows);

      if( new_MAE < note_old_MAE[1])
{
  note_second_old_MAE[1] = note_old_MAE[1];
  note_second_best_x[1] = note_best_x[1];
  note_second_best_y[1] = note_best_y[1];

  note_old_MAE[1] = new_MAE;
  note_best_x[1] = x_comp[i] ;
  note_best_y[1] = y_comp[i] ;
}
    }


  x_comp[0] = 0 + note_second_best_x[0] ;
  y_comp[0] = w + note_second_best_y[0] ;// 0,w

  x_comp[1] = w + note_second_best_x[0] ;
  y_comp[1] = w + note_second_best_y[0] ;// w,w

  x_comp[2] = w + note_second_best_x[0] ;
  y_comp[2] = 0 + note_second_best_y[0] ;// w,0

  x_comp[3] =  w + note_second_best_x[0] ;
  y_comp[3] = -w + note_second_best_y[0] ;// w,-w

  x_comp[4] =  0 + note_second_best_x[0] ;
  y_comp[4] = -w + note_second_best_y[0] ;// 0,-w

  x_comp[5] = -w + note_second_best_x[0] ;
  y_comp[5] = -w + note_second_best_y[0] ;//-w,-w

  x_comp[6] = -w + note_second_best_x[0] ;
  y_comp[6] = 0 + note_second_best_y[0]  ;//-w,0

  x_comp[7] = -w + note_second_best_x[0] ;
  y_comp[7] =  w + note_second_best_y[0] ;// -w,w

  x_comp[8] = 0 + note_second_best_x[0] ;
  y_comp[8] = 0 + note_second_best_y[0] ; // 0,0

      /*old_MAE = find_min( 9, x_comp, y_comp, ref, mtn,
  &best_x, &best_y,
  rows, cols, r, c) ;*/
  note_old_MAE[2] = MAX_OUT;
  note_best_x[2] = 0;
  note_best_y[2] = 0;

  for( int i = 0 ; i < 9 ; i++)
    {
      new_MAE = MAE( x_comp[i] + c, y_comp[i] + r,
     c,r ,ref,mtn,cols,rows);

      if( new_MAE < note_old_MAE[2])
{
  note_second_old_MAE[2] = note_old_MAE[2];
  note_second_best_x[2] = note_best_x[2];
  note_second_best_y[2] = note_best_y[2];

  note_old_MAE[2] = new_MAE;
  note_best_x[2] = x_comp[i] ;
  note_best_y[2] = y_comp[i] ;
}
    }

      ///=======================================
  w = 1;


  x_comp[0] = 0 + note_best_x[1] ;
  y_comp[0] = w + note_best_y[1] ;// 0,w

  x_comp[1] = w + note_best_x[1] ;
  y_comp[1] = w + note_best_y[1] ;// w,w

  x_comp[2] = w + note_best_x[1] ;
  y_comp[2] = 0 + note_best_y[1] ;// w,0

  x_comp[3] =  w + note_best_x[1] ;
  y_comp[3] = -w + note_best_y[1] ;// w,-w

  x_comp[4] =  0 + note_best_x[1] ;
  y_comp[4] = -w + note_best_y[1] ;// 0,-w

  x_comp[5] = -w + note_best_x[1] ;
  y_comp[5] = -w + note_best_y[1] ;//-w,-w

  x_comp[6] = -w + note_best_x[1] ;
  y_comp[6] = 0 + note_best_y[1]  ;//-w,0

  x_comp[7] = -w + note_best_x[1] ;
  y_comp[7] =  w + note_best_y[1] ;// -w,w

  x_comp[8] = 0 + note_best_x[1] ;
  y_comp[8] = 0 + note_best_y[1] ; // 0,0

      /*old_MAE = find_min( 9, x_comp, y_comp, ref, mtn,
  &best_x, &best_y,
  rows, cols, r, c) ;*/
  note_old_MAE[3] = MAX_OUT;
  note_best_x[3] = 0;
  note_best_y[3] = 0;

  for( int i = 0 ; i < 9 ; i++)
    {
      new_MAE = MAE( x_comp[i] + c, y_comp[i] + r,
     c,r ,ref,mtn,cols,rows);

      if( new_MAE < note_old_MAE[3])
{
  note_second_old_MAE[3] = note_old_MAE[3];
  note_second_best_x[3] = note_best_x[3];
  note_second_best_y[3] = note_best_y[3];

  note_old_MAE[3] = new_MAE;
  note_best_x[3] = x_comp[i] ;
  note_best_y[3] = y_comp[i] ;
}
    }


  x_comp[0] = 0 + note_second_best_x[1] ;
  y_comp[0] = w + note_second_best_y[1] ;// 0,w

  x_comp[1] = w + note_second_best_x[1] ;
  y_comp[1] = w + note_second_best_y[1] ;// w,w

  x_comp[2] = w + note_second_best_x[1] ;
  y_comp[2] = 0 + note_second_best_y[1] ;// w,0

  x_comp[3] =  w + note_second_best_x[1] ;
  y_comp[3] = -w + note_second_best_y[1] ;// w,-w

  x_comp[4] =  0 + note_second_best_x[1] ;
  y_comp[4] = -w + note_second_best_y[1] ;// 0,-w

  x_comp[5] = -w + note_second_best_x[1] ;
  y_comp[5] = -w + note_second_best_y[1] ;//-w,-w

  x_comp[6] = -w + note_second_best_x[1] ;
  y_comp[6] = 0 + note_second_best_y[1]  ;//-w,0

  x_comp[7] = -w + note_second_best_x[1] ;
  y_comp[7] =  w + note_second_best_y[1] ;// -w,w

  x_comp[8] = 0 + note_second_best_x[1] ;
  y_comp[8] = 0 + note_second_best_y[1] ; // 0,0

      /*old_MAE = find_min( 9, x_comp, y_comp, ref, mtn,
  &best_x, &best_y,
```

```
  rows, cols, r, c) ;*/
  note_old_MAE[4] = MAX_OUT ;
  note_best_x[4] = 0;
  note_best_y[4] = 0;

  for( int i = 0 ; i < 9 ; i++)
    {
      new_MAE = MAE( x_comp[i] + c, y_comp[i] + r,
     c,r ,ref,mtn,cols,rows);

      if( new_MAE < note_old_MAE[4])
{
  note_second_old_MAE[4] = note_old_MAE[4];
  note_second_best_x[4] = note_best_x[4];
  note_second_best_y[4] = note_best_y[4];

  note_old_MAE[4] = new_MAE;
  note_best_x[4] = x_comp[i] ;
  note_best_y[4] = y_comp[i] ;
}
    }
      //---
  x_comp[0] = 0 + note_best_x[2] ;
  y_comp[0] = w + note_best_y[2] ;// 0,w

  x_comp[1] = w + note_best_x[2] ;
  y_comp[1] = w + note_best_y[2] ;// w,w

  x_comp[2] = w + note_best_x[2] ;
  y_comp[2] = 0 + note_best_y[2] ;// w,0

  x_comp[3] =  w + note_best_x[2] ;
  y_comp[3] = -w + note_best_y[2] ;// w,-w

  x_comp[4] =  0 + note_best_x[2] ;
  y_comp[4] = -w + note_best_y[2] ;// 0,-w

  x_comp[5] = -w + note_best_x[2] ;
  y_comp[5] = -w + note_best_y[2] ;//-w,-w

  x_comp[6] = -w + note_best_x[2] ;
  y_comp[6] = 0 + note_best_y[2]  ;//-w,0

  x_comp[7] = -w + note_best_x[2] ;
  y_comp[7] =  w + note_best_y[2] ;// -w,w

  x_comp[8] = 0 + note_best_x[2] ;
  y_comp[8] = 0 + note_best_y[2] ; // 0,0

  note_old_MAE[5] = MAX_OUT;
  note_best_x[5] = 0;
  note_best_y[5] = 0;
  for( int i = 0 ; i < 9 ; i++)
    {
      new_MAE = MAE( x_comp[i] + c, y_comp[i] + r,
     c,r ,ref,mtn,cols,rows);

      if( new_MAE < note_old_MAE[5])
{
  note_second_old_MAE[5] = note_old_MAE[5];
  note_second_best_x[5] = note_best_x[5];
  note_second_best_y[5] = note_best_y[5];

  note_old_MAE[5] = new_MAE;
  note_best_x[5] = x_comp[i] ;
  note_best_y[5] = y_comp[i] ;
}
    }


  x_comp[0] = 0 + note_second_best_x[2] ;
  y_comp[0] = w + note_second_best_y[2] ;// 0,w

  x_comp[1] = w + note_second_best_x[2] ;
  y_comp[1] = w + note_second_best_y[2] ;// w,w

  x_comp[2] = w + note_second_best_x[2] ;
  y_comp[2] = 0 + note_second_best_y[2] ;// w,0

  x_comp[3] =  w + note_second_best_x[2] ;
  y_comp[3] = -w + note_second_best_y[2] ;// w,-w

  x_comp[4] =  0 + note_second_best_x[2] ;
  y_comp[4] = -w + note_second_best_y[2] ;// 0,-w

  x_comp[5] = -w + note_second_best_x[2] ;
  y_comp[5] = -w + note_second_best_y[2] ;//-w,-w

  x_comp[6] = -w + note_second_best_x[2] ;
  y_comp[6] = 0 + note_second_best_y[2]  ;//-w,0

  x_comp[7] = -w + note_second_best_x[2] ;
  y_comp[7] =  w + note_second_best_y[2] ;// -w,w

  x_comp[8] = 0 + note_second_best_x[2] ;
  y_comp[8] = 0 + note_second_best_y[2] ; // 0,0


  note_old_MAE[6] = MAX_OUT;
  note_best_x[6] = 0;
  note_best_y[6] = 0;

  for( int i = 0 ; i < 9 ; i++)
    {
      new_MAE = MAE( x_comp[i] + c, y_comp[i] + r,
     c,r ,ref,mtn,cols,rows);

      if( new_MAE < note_old_MAE[6])
{
  note_second_old_MAE[6] = note_old_MAE[6];
  note_second_best_x[6] = note_best_x[6];
  note_second_best_y[6] = note_best_y[6];

  note_old_MAE[6] = new_MAE;
  note_best_x[6] = x_comp[i] ;
  note_best_y[6] = y_comp[i] ;
}
    }


  old_MAE = note_old_MAE[3];
  best_x = note_best_x[3] ;
  best_y = note_best_y[3] ;

  for( int i = 4 ; i < 7 ; i++)
    {
      if( note_old_MAE[i] < old_MAE )
{
  best_x = note_best_x[i] ;
  best_y = note_best_y[i] ;
  old_MAE = note_old_MAE[i];
}
    }

  for( int i = 3 ; i < 7 ; i++)
    {
      if( note_second_old_MAE[i] < old_MAE )
{
  best_x = note_second_best_x[i] ;
  best_y = note_second_best_y[i] ;
  old_MAE = note_second_old_MAE[i];
}
    }

  log_x[q] = best_x ;
  log_y[q] = best_y;
  log_MAE[q] = old_MAE ;
}

for( int a = 0 ; a < 5; a++)
  {
    if( log_MAE[a] < old_MAE )
      {
best_x = log_x[a];
best_y = log_y[a];
old_MAE = log_MAE[a];
      }
  }

*(vecs + counter ) = best_x ; counter++ ;
*(vecs + counter ) = best_y ; counter++ ;

gen_difference_block( best_x , best_y ,c,r,ref, mtn,
      di, cols,rows) ;
      }
}
//-----------------------------------------------------
```

C.3 Proposed New Algorithms

```
// Optimised algorithms for fast motion searches .This is
// original/self modified stuff.....

#include <point.h>
#include <bag.h>
#include <vector.h>
#include <vector.cc>
```

```
#include <pbil.h>

#include <algorithms.cc>
// #include <dct.h>

/* Housekeeping  stuff at EOF****/

const long int LIST_SIZE = 16321 ;
const int LIST_SCALE_FACTOR = 58 ; // was 16
//const int ZERO_SHIFT = LIST_SIZE*LIST_SCALE_FACTOR/2 ;
const long int FRAME_SIZE_MAX = 272484 ;

static long int index[LIST_SIZE*LIST_SCALE_FACTOR] ;
static long int entry_index[LIST_SIZE*LIST_SCALE_FACTOR] ;

static point stuff[FRAME_SIZE_MAX];
static long int data[FRAME_SIZE_MAX] ;
static long int scratch[FRAME_SIZE_MAX];
static long int sum_norm[FRAME_SIZE_MAX] ;

const int VECTOR_LENGTH = 4;
template class vector<int, VECTOR_LENGTH> ;
static vector<int,VECTOR_LENGTH> features[FRAME_SIZE_MAX] ;

template class Bag<point>;

template class
ostream& operator<<( ostream& , const Bag<point>&);

static Bag<point> *point_bags[LIST_SIZE*LIST_SCALE_FACTOR] ;

void
modified_SEA(const unsigned char *ref ,
     const unsigned char *mtn ,
     int *di , int *vecs, const int rows ,
     const int cols , const int dummy)
{
  int offset ;
  long int sum = 0, R = 0;

  int old_MAE = MAX_OUT , new_MAE ;

  long int position, other_position , tmp , tmp_2;

  long int test_a , test_b ;

  int counter = 0;

  const int new_cols = cols - 8 , new_rows = rows - 8 ;

  //--Clear index array...
  for( int init = 0 ; init < LIST_SIZE ; init++)
    {
      index[init] = 0;
      entry_index[init] = 0 ;
    }
  //--

  //First col...
  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += *(ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= new_cols ; i++)
    {
      for( int j = 0 ; j < rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  - *(ref + offset - 1)
    + *(ref + offset + 7) ;
}
    }
 //first row ..... col wise.....
  for( int l = 0 ; l < cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += data[k*cols + l];

      sum_norm[l] = sum;
    }
 // Rest of the rows......col wise
  const int seven_cols_up = 7*cols ;

  for( int i = 0 ; i <= new_cols  ; i++)
    {
      for( int j = 1 ; j <= new_rows  ; j++)
{
  offset = i + j*cols ;

  sum_norm[offset] = sum_norm[offset - cols] - data[offset - cols]
    + data[offset + seven_cols_up ] ;
}
    }
  //---
  point pt , best_so_far;

  for( int i = 0 ; i <= new_cols ; i++)
    // make histogram of values ...
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;
index[ sum_norm[offset] ] += 1;
      }

  tmp = index[0] ;
  index[0] = 0 ;

  for( int i = 1 ; i < LIST_SIZE ; i++)
    {
      //index[i] += index[i - 1 ];
      tmp_2 = index[i] ;
      index[i] = ( index[i] > 0 ) ? tmp : 0  ;
      tmp += tmp_2 ;
    }

  for( int i = 0 ; i <= new_cols ; i++)
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;

stuff[ (index[ sum_norm[offset] ]) +
     (entry_index[sum_norm[offset] ]) ]
  =  point(i,j);

entry_index[ sum_norm[offset] ]++ ;
      }
  //--

  int stop = 0;
  int not_found = 0 ;

  //cout << "HERERERERERRERERERERERE " << R << endl ;

  for( int r = 0 ; r < rows ; r += 8)
    for( int c = 0 ; c < cols ; c += 8)
      {
offset = c + r*cols ;

// Calc R
R = 0 ; // Reset R.

for( int m = 0 ; m < 8 ; m++)
  for( int n = 0 ; n < 8 ; n++)
    R += *( mtn + offset + m + n*cols );

//---Find mtn vectors----

pt( c,r );
not_found = 1;
stop = 0;

old_MAE = MAX_OUT ;
position = R;
other_position = R - 1;

//cout << c << " " << r << endl ;

while( ( (R - old_MAE) <= position) &&
       ( (R + old_MAE) >= position) && ( position < LIST_SIZE)/*&&

       ( (R - old_MAE) <= other_position) &&
       ( (R + old_MAE) >= other_position)*/ ){

  if( (entry_index[position] != 0) /*&& ( position < LIST_SIZE)*/)
    {
      for( int q = 0 ; q < entry_index[position]; q++)
{
  new_MAE = MAE( stuff[ index[position] + q].x(),
 stuff[ index[position] + q].y(),
```

```
 c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = ( best_so_far - pt).mod() ;
  test_b = ( stuff[ index[position] + q] - pt).mod();

  best_so_far = ( test_a > test_b )
    ? stuff[ index[position] + q] : best_so_far ;

/*cout << test_a << " next" << test_b << " pt" << pt
  << " best " << best_so_far << endl ;*/
}
}
    }
  position++;
}

while( ( (R - old_MAE) <= other_position) &&
       ( (R + old_MAE) >= other_position) &&
       ( other_position >= 0) && (other_position != position)){
  if( (entry_index[other_position] != 0) /*&&
      ( other_position >= 0)*/)
    {
      for( int q = 0 ; q < entry_index[other_position] ; q++)
{
  new_MAE = MAE( stuff[ index[other_position] + q].x(),
 stuff[ index[other_position] + q].y(),
 c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[other_position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = ( best_so_far - pt).mod() ;
  test_b = ( stuff[ index[other_position] + q]
    - pt).mod()  ;

  best_so_far = ( test_b < test_a )
    ? stuff[ index[other_position] + q] :
    best_so_far ;
      }
      }
    }
  other_position--;

  //cout << old_MAE << endl ;

}
//----
*(vecs + counter ) = (best_so_far.x() - c) ; counter++ ;
*(vecs + counter ) = (best_so_far.y() - r) ; counter++ ;

gen_difference_block( best_so_far.x() - c,
      best_so_far.y() - r,
      c,r,ref, mtn,
      di, cols,rows) ;

      }
}
//-----------------------------------------------

static int old_MADs[300000] ;
static char old_block[70] ;

void
modified_SEA_v2(const unsigned char *ref ,
const unsigned char *mtn ,
int *di , int *vecs, const int rows ,
const int cols , const int dummy)
{
  static int logger = 0;

  int offset , test_abs_val ;
  long int sum = 0, R = 0, D , test_pos ;

  long int position, other_position , tmp , tmp_2;

  long int test_a , test_b ;

  unsigned int counter = 0;

  const int new_cols = cols - 8 , new_rows = rows - 8 ;
  int old_MAE = MAX_OUT , new_MAE ;

  /*static unsigned long int index[LIST_SIZE] ;
  static unsigned long int entry_index[LIST_SIZE] ;
  static point stuff[FRAME_SIZE_MAX];
  static unsigned int data[300000] ;
  static unsigned int sum_norm[300000] ;*/

  //--Clear index array...
  for( int init = 0 ; init < LIST_SIZE ; init++)
    {
      index[init] = 0;
      entry_index[init] = 0 ;
    }
  //--

  //First col...
  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += *( ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= new_cols ; i++)
    {
      for( int j = 0 ; j < rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  - *(ref + offset - 1)
    + *( ref + offset + 7) ;
}
    }
 //first row ..... col wise.....
  for(int l = 0 ; l < cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += data[k*cols + l];

      sum_norm[l] = sum;
    }
 // Rest of the rows......col wise
  const int seven_cols_up = 7*cols ;

  for( int i = 0 ; i <= new_cols  ; i++)
    {
      for( int j = 1 ; j <= new_rows  ; j++)
{
  offset = i + j*cols ;

  sum_norm[offset] = sum_norm[offset - cols]  - data[offset - cols]
    + data[offset + seven_cols_up ] ;
}
    }
  //---
  point pt , best_so_far;

  for( int i = 0 ; i <= new_cols ; i++)
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;
index[ sum_norm[offset] ] += 1;
      }

  tmp = index[0] ;
  index[0] = 0 ;

  for( int i = 1 ; i < LIST_SIZE ; i++)
    {
      //index[i] += index[i - 1 ];
      tmp_2 = index[i] ;
      index[i] = tmp ;
      tmp += tmp_2 ;
    }

  for( int i = 0 ; i <= new_cols ; i++)
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;

stuff[ index[ sum_norm[offset] ] +
```

```
    entry_index[sum_norm[offset] ] ] =
  pt(i,j);

entry_index[ sum_norm[offset] ] =
  entry_index[ sum_norm[offset] ] + 1;
      }

  //--


  int stop = 0;
  int not_found = 0 ;
//-----------------------------------------

  if( logger == 0)
    {
      for( int a = 0 ; a < 8 ; a++)
{
  for( int b = 0 ; b < 8 ; b++)
    {
      old_block[ b + a*8] = *(mtn + b + a*cols);
    }
}

      for( int i = 0 ; i <= new_rows ; i++ )
for( int j = 0 ; j <= new_cols ; j++)
  {
    offset = j + i*cols;
    old_MADs[ offset] = MAE( j ,i ,0 ,0, ref, mtn ,cols,
     rows);
/*if( old_MAE < old_MADs[ offset])
  {
    best_so_far( j , i) ;

    old_MAE = old_MADs[ offset] ;
  } else if( old_MAE <= old_MADs[ offset])
    {
      best_so_far( j , i);
    }*/
  }
      logger = 1;
    }
  // handle the first one. 0,0
  /*(vecs + counter ) = (best_so_far.x() ) ; counter++ ;
  *(vecs + counter ) = (best_so_far.y() ) ; counter++ ;

  gen_difference_block(best_so_far.x()  , best_so_far.y() ,
       0,0,ref, mtn, di, cols,rows) ;*/
  //-----------------------------------------

  for( int r = 0 ; r < rows ; r += 8)
    for( int c = 0 ; c < cols ; c += 8)
      {
offset = c + r*cols ;

//if( (r == 0) && ( c == 0) ) continue ;// skip 0,0

// cout << r << " " << c << endl ;

// Calc R
R = 0 ; // Reset R.
D = 0 ; // Reset D.

for( int m = 0 ; m < 8 ; m++)
  for( int n = 0 ; n < 8 ; n++)
    {
      R += *( mtn + offset + m + n*cols );
      tmp =  ( *( mtn + offset + m + n*cols )  -
       *( old_block + m + n*8 ) );
      tmp = ( tmp < 0) ? -tmp : tmp ;
      D += tmp ;
    }


//---Find mtn vectors----

pt( c,r );
not_found = 1;
stop = 0;
old_MAE = MAX_OUT ;

position = R;
other_position = R - 1;

while( ( (R - old_MAE) <= position) &&
       ( (R + old_MAE) >= position) /*&&

       ( (R - old_MAE) <= other_position) &&
       ( (R + old_MAE) >= other_position)*/ ){
  if( (entry_index[position] != 0) && ( position < LIST_SIZE))
    {
      for( int q = 0 ; q < entry_index[position]; q++)
{
  test_pos = stuff[ index[position] + q].x() +
    stuff[ index[position] + q].y()*cols ;

  test_abs_val = old_MADs[test_pos] - D;
  test_abs_val = ( test_abs_val < 0) ? -test_abs_val
    :test_abs_val;

  if( test_abs_val <= old_MAE ){
    new_MAE = MAE( stuff[ index[position] + q].x(),
   stuff[ index[position] + q].y(),
   c , r, ref, mtn , cols, rows );

    if( new_MAE < old_MAE)
      {
best_so_far = stuff[ index[position] + q];
old_MAE = new_MAE;
      }else if( new_MAE == old_MAE)
{
  test_a = pt(best_so_far) ;
  test_b = pt(stuff[ index[position] + q]) ;
  best_so_far = ( test_a > test_b )
    ? stuff[ index[position] + q] : best_so_far
    ;
}
  }
}
    }
  position++;
}
while( /*( (R - old_MAE) <= position) &&
       ( (R + old_MAE) >= position) &&*/

       ( (R - old_MAE) <= other_position) &&
       ( (R + old_MAE) >= other_position)){
  if( (entry_index[other_position] != 0) &&
      ( other_position >= 0) && (other_position != position) )
    {
      for( int q = 0 ; q < entry_index[other_position] ; q++)
{
  test_pos = stuff[ index[other_position] + q].x() +
    stuff[ index[other_position] + q].y()*cols ;

test_abs_val = old_MADs[test_pos] - D;
test_abs_val = ( test_abs_val < 0) ? -test_abs_val
  :test_abs_val;

if( test_abs_val <= old_MAE ){
new_MAE = MAE( stuff[ index[other_position] + q].x(),
       stuff[ index[other_position] + q].y(),
       c , r, ref, mtn , cols, rows );

if( new_MAE < old_MAE)
  {
    best_so_far = stuff[ index[other_position] + q];
    old_MAE = new_MAE;
  }else if( new_MAE == old_MAE)
    {
      test_a = ( best_so_far - pt).mod() ;
      test_b = ( stuff[ index[other_position] + q] - pt).mod();

      best_so_far = ( test_a > test_b)
? stuff[ index[other_position] + q] :
best_so_far ;
    }
}
      }
  }
  other_position--;
}
//----
*(vecs + counter ) = (best_so_far.x() - c) ; counter++ ;
*(vecs + counter ) = (best_so_far.y() - r); counter++ ;

gen_difference_block(best_so_far.x() - c ,
     best_so_far.y() - r,
     c,r,ref, mtn, di, cols,rows) ;
      }
}
//-----------------------------------------
void
sampled_pts(const unsigned char *ref ,
    const unsigned char *mtn ,
    int *di , int *vecs, const int rows ,
    const int cols , const int sub_sample)
{
  //dct forward_dct ;
```

```
  int offset ;
  long int sum = 0, R = 0;

  long int position, other_position , tmp ,
    tmp_2, old_MAE, new_MAE;
  long int test_a , test_b ;

  unsigned int counter = 0;

  const int new_cols = cols - 8 , new_rows = rows - 8 ;
  /*
  static unsigned long int index[LIST_SIZE] ;
  static unsigned long int entry_index[LIST_SIZE] ;
  static point stuff[FRAME_SIZE_MAX];
  static unsigned int data[300000] ;
  static unsigned int sum_norm[300000] ;*/

  //--Clear index array...
  for( int init = 0 ; init < LIST_SIZE ; init++)
    {
      index[init] = 0;
      entry_index[init] = 0 ;
    }
  //--

  //First col...
  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += *(ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= new_cols ; i++)
    {
      for( int j = 0 ; j < rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  - *(ref + offset - 1)
    + *(ref + offset + 7) ;
}
    }
 //first row ..... col wise.....
  for( int l = 0 ; l < cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += data[k*cols + l];

      sum_norm[l] = sum;
    }
 // Rest of the rows.....col wise
  const int seven_cols_up = 7*cols ;

  for( int i = 0 ; i <= new_cols  ; i++)
    {
      for( int j = 1 ; j <= new_rows  ; j++)
{
  offset = i + j*cols ;

  sum_norm[offset] = sum_norm[offset - cols]  - data[offset - cols]
    + data[offset + seven_cols_up ] ;
}
    }
  //---
  point pt , best_so_far;

  for( int i = 0 ; i <= new_cols ; i += sub_sample)
    for( int j = 0 ; j <= new_rows ; j += sub_sample)
      {
offset = i + j*cols ;
index[ sum_norm[offset] ] += 1;
      }

  tmp = index[0] ;
  index[0] = 0 ;

  for( int i = 1 ; i < LIST_SIZE ; i++)
    {
      //index[i] += index[i - 1 ];
      tmp_2 = index[i] ;
      index[i] = tmp ;
      tmp += tmp_2 ;
    }

  for( int i = 0 ; i <= new_cols ; i += sub_sample)
    for( int j = 0 ; j <= new_rows ; j += sub_sample)
      {
offset = i + j*cols ;

stuff[ index[ sum_norm[offset] ] +
     entry_index[sum_norm[offset] ] ] =
  pt(i,j);

entry_index[ sum_norm[offset] ] =
  entry_index[ sum_norm[offset] ] + 1;
      }

  //--

  int stop = 0;
  int not_found = 0 ;

  for( int r = 0 ; r < rows ; r += 8)
    for( int c = 0 ; c < cols ; c += 8)
      {
offset = c + r*cols ;

// Calc R
R = 0 ; // Reset R.

for( int m = 0 ; m < 8 ; m++)
  for( int n = 0 ; n < 8 ; n++)
    R += *(mtn + offset + m + n*cols );

//---Find mtn vectors----

pt( c,r );
best_so_far(c,r);
not_found = 1;
stop = 0;
old_MAE = MAX_OUT;

// cout << r << "\t" << c << endl ;

old_MAE = MAE( c, r, c, r, ref, mtn, cols, rows);
if( old_MAE  == 0 )
  {
    best_so_far = pt ;
  }
else
  {
    position = R;
    other_position = R - 1;

    while( ( (R - old_MAE) <= position) &&
   ( (R + old_MAE) >= position) && ( position < LIST_SIZE)

   /*( (R - old_MAE) <= other_position) &&
   ( (R + old_MAE) >= other_position)*/){

      if( (entry_index[position] != 0) /*&&
  ( position < LIST_SIZE)*/){

      for( int q = 0 ; q < entry_index[position]; q++)
{
  new_MAE = MAE( stuff[ index[position] + q].x(),
 stuff[ index[position] + q].y(),
 c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = (best_so_far - pt).mod() ;
  test_b = (stuff[ index[position] + q] - pt).mod() ;
  best_so_far = ( test_a > test_b )
    ? stuff[ index[position] + q] :
    best_so_far ;
}
}
      }
      position++;
    }
    while( /*( (R - old_MAE) <= position) &&
   ( (R + old_MAE) >= position) &&*/

   ( (R - old_MAE) <= other_position) &&
   ( (R + old_MAE) >= other_position) &&
   ( other_position >= 0)){
```

```
      if( (entry_index[other_position] != 0) &&
  /*( other_position >= 0) &&*/ (position != other_position) ){

for( int q = 0 ; q < entry_index[other_position] ; q++)
  {
    new_MAE = MAE( stuff[ index[other_position] + q].x(),
   stuff[ index[other_position] + q].y(),
   c,r, ref , mtn , cols , rows);

    if( new_MAE < old_MAE)
      {
best_so_far = stuff[ index[other_position] + q];
old_MAE = new_MAE ;
      }else
if( new_MAE == old_MAE)
  {
    test_a = (best_so_far - pt).mod() ;
    test_b = (stuff[ index[other_position] +
   q] - pt).mod() ;

    best_so_far = (test_a > test_b )
      ? stuff[ index[other_position] + q] :
      best_so_far ;
  }
  }
      }
      //position++;
      other_position--;
    }
  }
//----

point best_point( best_so_far ) ;
long int max = MAX_OUT, new_pos , length ;
long int old_length = ( best_so_far.x() - c)*( best_so_far.x()- c)
  + (best_so_far.y() - r)*( best_so_far.y() - r);

const int range = sub_sample - 1;


for( int ly = best_so_far.y() - range;
     ly <= (best_so_far.y() + range) ; ly++)
  for( int lx = best_so_far.x() - range ;
       lx <= (best_so_far.x() + range) ; lx++)
    {
      if( (ly >= 0) && ( ly <= new_rows) &&
  (lx >= 0) && ( lx <= new_cols) )
{
  new_pos = MAE( lx , ly ,
 c,//best_so_far.x() ,
 r,//best_so_far.y() ,
 ref, mtn,
 cols, rows );

  if( new_pos < max )
    {
      best_point = point( lx , ly ) ;
      max = new_pos ;
    }
  else if( (new_pos == max) && ( new_pos != MAX_OUT))
    {
      length = (lx - c)*(lx - c) + (ly - r)*(ly - r);

      if( length < old_length)
{
  old_length = length;
  best_point( lx , ly );
}
    }
}
    }

//----
*(vecs + counter ) = (best_point.x() - c) ; counter++ ;
*(vecs + counter ) = (best_point.y() - r); counter++ ;

gen_difference_block(best_point.x() - c ,best_point.y() - r ,
     c,r,ref, mtn, di, cols,rows) ;
//forward_dct.perform_dct( 8 , 8, *di , *di , r , c  ) ;
      }
}
//-------------------------------------------

void
feature_search(const unsigned char *ref ,
       const unsigned char *mtn ,
       int *di , int *vecs, const int rows ,
       const int cols , const int sub_sample)
{
  int offset ;
  long int sum = 0, R = 0;

  int old_MAE = MAX_OUT , new_MAE ;

  long int position, other_position ;

  long int test_a , test_b ;

  int counter = 0;

  const int new_cols = cols - 8 , new_rows = rows - 8 ;

  //--Clear index array...
  for( int init = 0 ; init < LIST_SIZE ; init++)
    {
      index[init] = 0;
      entry_index[init] = 0 ;
      point_bags[init] = NULL ;
    }
  //--

  //First col...
  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += *(ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= new_cols ; i++)
    {
      for( int j = 0 ; j < rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  - *(ref + offset - 1)
    + *(ref + offset + 7) ;
}
    }
 //first row ..... col wise.....
  for( int l = 0 ; l < cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += data[k*cols + l];

      sum_norm[l] = sum;
    }
 // Rest of the rows.....col wise
  const int seven_cols_up = 7*cols ;

  for( int i = 0 ; i <= new_cols  ; i++)
    {
      for( int j = 1 ; j <= new_rows  ; j++)
{
  offset = i + j*cols ;

  sum_norm[offset] = sum_norm[offset - cols]
    - data[offset - cols]
    + data[offset + seven_cols_up ] ;
}
    }
  //---
  point pt , best_so_far, current_point ;

  for( int i = 0 ; i <= new_cols ; i++)
    // make histogram of values ...
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;
index[ sum_norm[offset] ]++;
      }

  //tmp = index[0] ;
  //index[0] = 0 ;

  for(int i = 0 ; i < LIST_SIZE ; i++)
    {/*
      //index[i] += index[i - 1 ];
      tmp_2 = index[i] ;
      index[i] = ( index[i] > 0 ) ? tmp : 0  ;
      tmp += tmp_2 ;*/

      point_bags[i] = new Bag<point>(index[i]) ;
```

```
      if( point_bags[i] == NULL)
{
  cerr << "Unable to allocate bag !" << endl;
  exit(1);
}
    }

  for( int i  = 0 ; i <= new_cols ; i++)
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;

point_bags[ sum_norm[offset] ]->insert_item( point(i,j) ) ;
      }
  //--

  int stop = 0;
  int not_found = 0 ;

  //cout << "HERERERERERERERERERERERE " << R << endl ;

  for( int r = 0 ; r < rows ; r += 8)
    for( int c = 0 ; c < cols ; c += 8)
      {
offset = c + r*cols ;

// Calc R
R = 0 ; // Reset R.

for( int m = 0 ; m < 8 ; m++)
  for( int n = 0 ; n < 8 ; n++)
    R += *( mtn + offset + m + n*cols );

//---Find mtn vectors----

pt( c,r );
best_so_far(c,r);
not_found = 1;
stop = 0;

old_MAE = MAX_OUT ;
position = R;
other_position = R - 1;

//cout << c << " " << r << endl ;

while( ( (R - old_MAE) <= position) &&
       ( (R + old_MAE) >= position) && ( position < LIST_SIZE)

       /*( (R - old_MAE) <= other_position) &&
       ( (R + old_MAE) >= other_position)*/ ){

  if( (index[position] != 0)
      /*&& ( position < LIST_SIZE) */)
    {
      //cout << " ++pos " << position << endl ;

      /*if( point_bags[ position ]
->current(0, best_so_far) ==
  0)
{
  cerr << "Out of bounds:bag !" << endl ;
  exit(1);
}*/

      for( int q = 0 ;
   q < point_bags[position]
     ->get_number_of_stored_elements()
     ; q++)
{
  if( point_bags[position]
      ->current(q, current_point)==
      0)
    {
      cerr << "Out of bounds:bag !" << endl ;
      exit(1);
    }

  new_MAE = MAE( current_point.x(),
 current_point.y(),
 c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = current_point;
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = ( best_so_far - pt).mod() ;
  test_b = ( current_point - pt).mod();

  best_so_far = ( test_a > test_b )
    ? current_point : best_so_far ;

/*cout << test_a << " next" << test_b << " pt" << pt
  << " best " << best_so_far << endl ;*/
}
}
    }
  position++;
}
while( /*( (R - old_MAE) <= position) &&
       ( (R + old_MAE) >= position) &&*/

       ( (R - old_MAE) <= other_position) &&
       ( (R + old_MAE) >= other_position) &&
       (other_position >= 0) ){

  if( (index[other_position] != 0) &&
      /*(other_position >= 0) && */
      (other_position != position))
    {
      /*if( point_bags[ other_position ]
->current(0, best_so_far) ==
  0)
{
  cerr << "Out of bounds:bag !" << endl ;
  exit(1);
}*/
      for( int q = 1 ;
   q < point_bags[other_position]->
     get_number_of_stored_elements();
   q++)
{
  if( point_bags[other_position]->current(q, current_point)==
      0)
    {
      cerr << "Out of bounds:bag !" << endl ;
      exit(1);
    }
  new_MAE = MAE( current_point.x(),
 current_point.y(),
 c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = current_point;
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = ( best_so_far - pt).mod() ;
  test_b = ( current_point - pt).mod()  ;

  best_so_far = ( test_a > test_b )
    ? current_point :  best_so_far ;
      }
      }
    }
  other_position--;

  //cout << old_MAE << endl ;

}
//----
*(vecs + counter ) = (best_so_far.x() - c) ; counter++ ;
*(vecs + counter ) = (best_so_far.y() - r) ; counter++ ;

gen_difference_block( best_so_far.x() - c,
      best_so_far.y() - r,
      c,r,ref, mtn,
      di, cols,rows) ;

      }
  delete [] point_bags ;// garbage collection ;;;
}
//-------------------------------------------
inline
int sum_up( const int x, const int y,
    const int a, const int b,
    const int cols , const unsigned char *ref)
{
  int tmp = 0;

  /*cout << "x: " << x << " y: " << y
       << " a: " << a << " b:" << b << endl ;*/

  if( ( x >= a) || ( y >= b))
```

```
    {
      cerr << "(x,y) must be before (a,b)" << endl ;
      exit(1);
    }
  //cout << "su " << ((void *)ref) << endl ;

  for( int j = y ; j <= b ; j++)
    for( int i = x ; i <= a ; i++)
      {
/*cout << ((void *)ref) << " "
     << ((void*)(ref + i + j*cols))
     << " j" << j << " i " << i << endl ;*/
tmp += *(ref + i + j*cols) ;
      }
  //cout << "su end" << endl ;
  return tmp;
}


inline
int feature_MAE( const int v , const int u ,
 const int c , const int r ,
 const vector<int, VECTOR_LENGTH> *features,
 const int cols, const int rows)
{
  int result = 0 , tmp , tmp_1 , tmp_2 ;;
  int offset_1 = c + r*cols ;
  int offset_2 = v + u*cols ;

  if( (c < 0) || (r < 0) || ( c >= cols)
      || ( r >= rows) ||
      (v < 0) || (u < 0) || ( v > (cols - 8))
      || ( u > ( rows - 8)))
    {
      cerr << "Input out of bounds! Feature_MAD" << endl ;
      cerr << " c: " << c << " r: " << r << " v: " << v
   << " u: " << u << endl ;
      return MAX_OUT;
    }
  //cout << " Arsytcdyxc " << endl ;

  for( int n = 0 ; n < VECTOR_LENGTH ; n++)
    {
      //cout << "\t" << n ;
      tmp_1 = features[offset_1][n] ;
      //cout << " a" ;
      tmp_2 = features[offset_2][n] ;
      //cout << " b" << endl ;
      tmp = tmp_1 - tmp_2 ;

      tmp = ( tmp < 0) ? -tmp : tmp ;
      result += tmp ;
    }
  //cout << " Brsytcdyxc " << endl ;

  return result;
}

void
feature_search_2(const unsigned char *ref ,
 const unsigned char *mtn ,
 int *di , int *vecs, const int rows ,
 const int cols  ,
 const int sub_sample_factor)
{
  int offset ;
  int R = 0;
  int old_MAE = MAX_OUT , new_MAE = MAX_OUT ;
  int new_feature_MAE , old_feature_MAE = MAX_OUT ;
  int position , other_position , sum , tmp_2 , test_a , test_b ;

  //long int dist_old , dist_new ;//<><><><

  unsigned int counter = 0;

  const int new_cols = cols - 8 , new_rows = rows - 8 ;

  int q1 , q2, q3 , q4 ;

  //static int data[300000] ;
  //static int sum_norm[300000] ;

  //static vector<int,VECTOR_LENGTH>
  //features[300000] ;

  int tmp ;

  //--Clear index array...
  for( int init = 0 ; init < LIST_SIZE ; init++)
    {
      index[init] = 0;
      entry_index[init] = 0 ;
    }

  // making feature----------

  const int N = 4;
  const int N_1_cols_up = ( N - 1)*cols;
  //--------
  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < N ; k++)
sum += *(ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= (new_cols + 4); i++)
    {
      for( int j = 0 ; j < rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  - *(ref + offset - 1)
    + *(ref + offset + N - 1) ;
}
    }
  //first row ..... col wise.....
  for( int l = 0 ; l < cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < N ; k++)
sum += data[k*cols + l];

      scratch[l] = sum;
    }
  // Rest of the rows......col wise

  for( int i = 0 ; i <= (new_cols + 4) ; i++)
    {
      for( int j = 1 ; j <= (new_rows + 4) ; j++)
{
  offset = i + j*cols ;

  scratch[offset] = scratch[offset - cols]  -
    data[offset - cols] + data[offset + N_1_cols_up ] ;
}
    }

  //---
  for( int j = 0 ; j <= new_rows ; j++)
    for( int i = 0 ; i <= new_cols ; i++)
      {
offset = i + j*cols ;

/*q1 = sum_up( i,j, i+3, j+3,cols, ref) ;
q2 = sum_up( i+4,j, i+7, j+3,cols, ref) ;
q3 = sum_up( i,j+4, i+3, j+7,cols, ref) ;
q4 = sum_up( i+4,j+4, i+7, j+7,cols, ref) ;*/

q1 = scratch[ offset ] ;
q2 = scratch[ offset + 4 ];
q3 = scratch[ offset + 4*cols ];
q4 = scratch[ offset + 4 + 4*cols ];

(features[offset])[0] = q1 + q2 + q3 + q4 ;
(features[offset])[1] = q1 + q2 - q3 - q4 ;
(features[offset])[2] = q1 + q3 - q2 - q4;
(features[offset])[3] = q1 - q2 - q3 + q4 ;

sum_norm[offset] = (features[offset])[0];;

/*test = sum_up(i,j,i+7,j+7, cols, ref);

if( test != sum_norm[offset])
  {
    cerr << "out_of_range!" << endl ;
    cerr << i << "\t" << j << endl ;
  }*/
      }

  //---
  point pt , best_so_far;

  for( int i = 0 ; i <= new_cols ; i++)
```

```
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;
index[ sum_norm[offset] ] += 1;
      }

  tmp = index[0] ;
  index[0] = 0 ;

  for( int i = 1 ; i < LIST_SIZE ; i++)
    {
      //index[i] += index[i - 1 ];
      tmp_2 = index[i] ;
      index[i] = tmp ;
      tmp += tmp_2 ;
    }

  for( int i = 0 ; i <= new_cols ; i++)
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;

stuff[ index[ sum_norm[offset] ] +
     entry_index[sum_norm[offset] ] ] =
  pt(i,j);

entry_index[ sum_norm[offset] ] =
  entry_index[ sum_norm[offset] ] + 1;
      }

 cout << "end section make feature!" << endl ;

  const int SUB_WIDTH = 10;
  const int THRESHOLD = 300;

  for( int r = 0 ; r < rows ; r += 8)
    for( int c = 0 ; c < cols ; c += 8)
      {

//cout << c << "\t" << r << endl ;

offset = c + r*cols ;

// Calc R
R = 0 ; // Reset R.

for( int m = 0 ; m < 8 ; m++)
  for( int n = 0 ; n < 8 ; n++)
    R += *(mtn + offset + m + n*cols );

//---Find mtn vectors----

old_MAE = MAX_OUT ;
new_MAE = 0;
old_feature_MAE = MAX_OUT ;
new_feature_MAE = 0;

best_so_far(c,r);
pt( c, r );
//dist_old = rows*rows + cols*cols;//<><><><

position = R;
other_position = R - 1;

// do an initial search around the origin of the search block
for( int yy = r - SUB_WIDTH ; yy <= (r + SUB_WIDTH) ; yy += 2)
  {
    if((yy < 0) || ( yy > new_rows)) break;

    for( int xx = c - SUB_WIDTH ; xx <= (c + SUB_WIDTH) ; xx += 2)
      {
if( (xx < 0) || ( xx > new_cols)) break;

new_MAE = MAE( xx, yy, c,r, ref , mtn , cols ,
      rows);
if( new_MAE < old_MAE)
  {
    best_so_far( xx , yy );
    old_MAE = new_MAE;

  }else if( (new_MAE == old_MAE) &&
    (new_MAE != MAX_OUT) &&
    ( (point( xx , yy ) - pt).mod() <
      (best_so_far - pt).mod() ) )
    {
      best_so_far( xx , yy );
      //old_MAE = new_MAE;
    }

/*
new_feature_MAE = feature_MAE( xx, yy, c,r, features,
       cols, rows);

if( new_feature_MAE <= old_feature_MAE)
  {
    old_feature_MAE = new_feature_MAE;

    new_MAE = MAE( xx, yy, c,r, ref , mtn , cols ,
   rows);
    if( new_MAE < old_MAE)
      {
best_so_far( xx , yy );
old_MAE = new_MAE;

      }else if( (new_MAE == old_MAE) &&
//( new_feature_MAE != MAX_OUT) &&
( (point( xx , yy ) - pt).mod() <
  (best_so_far - pt).mod() ) )
{
  //old_feature_MAE = new_feature_MAE;
  best_so_far( xx , yy );
  old_MAE = new_MAE;
}
  }*/
      }
  }

/*old_MAE = MAE( best_so_far.x(),
       best_so_far.y(), c,r, ref , mtn , cols , rows);*/

//-----------------------------------

if( old_MAE > THRESHOLD){

  //cout << old_MAE << " " << R << " "
  //     << position << " " << other_position << endl;

  while( ( (R - old_MAE) <= position) &&
 ( (R + old_MAE) >= position) && ( position < LIST_SIZE)
              /*( (R - old_MAE) <= other_position) &&
 ( (R + old_MAE) >= other_position)*/ ){

    if( (entry_index[position] != 0) /*&&
      ( position < LIST_SIZE)*/ ){

    for( int q = 0 ; q < entry_index[position]; q++)
      {
new_feature_MAE = feature_MAE( stuff[ index[position] + q].x(),
       stuff[ index[position] + q].y(),
       c,r,
       features,cols , rows);

if( new_feature_MAE <= old_feature_MAE ){

  old_feature_MAE = new_feature_MAE;

/*test = sum_up( stuff[ index[position] + q].x(),
       stuff[ index[position] + q].y(),
       stuff[ index[position] + q].x() +7,
       stuff[ index[position] + q].y() +7,
       cols, ref);

if( test != position)
  {
    cerr << "misclassifed block!" << endl ;
  }*/


  new_MAE = MAE( stuff[ index[position] + q].x(),
 stuff[ index[position] + q].y(),
 c,r, ref , mtn , cols , rows);

  if( ((index[position] + q) < 0) ||
      ((index[position] + q) > (512*512)))
      {
cerr << "bounds!" << endl ;
      }

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = (best_so_far - pt).mod() ;
  test_b = (stuff[ index[position] + q] - pt).mod() ;
  best_so_far = ( test_a > test_b )
    ? stuff[ index[position] + q] :
    best_so_far ;
```

```
}
  }
      }
  }
  position++;
}
while( /*( (R - old_MAE) <= position) &&
       ( (R + old_MAE) >= position) &&*/
      ( (R - old_MAE) <= other_position) &&
      ( (R + old_MAE) >= other_position) && (other_position >= 0)){

  if( (entry_index[other_position] != 0) &&
      /*(other_position >= 0) && */(other_position != position) ){

    for( int q = 0 ; q < entry_index[other_position] ; q++)
      {
new_feature_MAE = feature_MAE( stuff[index[other_position]
     + q].x(),
       stuff[index[other_position]
     + q].y(),
       c,r,
       features,cols , rows);
if( new_feature_MAE <= old_feature_MAE ){

  old_feature_MAE = new_feature_MAE;

/*test = sum_up( stuff[ index[other_position] + q].x(),
       stuff[ index[other_position] + q].y(),
       stuff[ index[other_position] + q].x() +7,
       stuff[ index[other_position] + q].y() +7,
       cols, ref);

if( test != other_position)
  {
    cerr << "@ misclassifed block!" << endl ;
  }*/

 if( ((index[other_position] + q) < 0) ||
     ((index[other_position] + q) > (512*512)))
      {
cerr << "bounds!" << endl ;
      }
  new_MAE = MAE( stuff[ index[other_position] + q].x(),
 stuff[ index[other_position] + q].y(),
   c,r, ref , mtn , cols , rows);

    if( new_MAE < old_MAE)
      {
best_so_far = stuff[ index[other_position] + q];
old_MAE = new_MAE ;
      }else
if( new_MAE == old_MAE)
  {
    test_a = (best_so_far - pt).mod() ;
    test_b = (stuff[ index[other_position] +
   q] - pt).mod() ;

    best_so_far = (test_a > test_b )
      ? stuff[ index[other_position] + q] :
      best_so_far ;
  }
    }
      }
  }
  // position++;
  other_position--;
}
}
*(vecs + counter ) = best_so_far.x() - c ; counter++ ;
*(vecs + counter ) = best_so_far.y() - r ; counter++ ;

gen_difference_block( best_so_far.x() - c ,
      best_so_far.y() - r ,c,r,ref, mtn,
      di, cols,rows) ;

      }
}
//-------------------------------------------

class PBIL;

void
Pop_BIL(const unsigned char *ref , const unsigned char *mtn ,
     int *di , int *vecs, const int rows ,
     const int cols  , const int sub_sample_factor)
{
  unsigned int counter = 0;
  int best_x , best_y , offset, new_MAE, old_MAE;

  PBIL loc ;

  for( int r = 0 ; r < rows ; r += 8)
    for( int c = 0 ; c < cols ; c += 8)
      {
offset = c + r*cols ;
best_x = 0;
best_y = 0;
new_MAE = MAX_OUT;
old_MAE = MAX_OUT ;

loc.zero();

loc.set_x(c);
loc.set_y(r);

loc.set_results_x( c );
loc.set_results_y( r );

//cout << r << "\t" << c << endl ;

for( int g = 0 ; g < 100 ; g++)
  {
    for( int t = 0 ; t < 30 ; t++ )
      {
new_MAE = MAE(loc.extract_x(),
      loc.extract_y(),
      c,r, ref , mtn , cols , rows);

loc.evaluation(new_MAE, old_MAE);

//cout << "!" << endl;
loc.next();
//cout << "@" << endl;

if( new_MAE < old_MAE ) old_MAE = new_MAE;
      }
    loc.update_x();
    loc.update_y();
    //cout << "$" << endl ;
    loc.mutate() ;
  }

best_x = loc.final_output_x() ;
best_y = loc.final_output_y() ;

if( best_x < 0) best_x = 0;
if( best_x > ( cols - 8)) best_x = cols - 8 ;
if( best_y < 0) best_y = 0;
if( best_y > ( rows - 8)) best_y = rows - 8 ;

*(vecs + counter ) = best_x - c ; counter++ ;
*(vecs + counter ) = best_y - r ; counter++ ;

gen_difference_block( best_x - c ,
      best_y - r ,c,r,ref, mtn,
      di, cols,rows) ;
     }
}
//-------------------------------------------
/*
const int MAX_C = 512;
const int MAC_R = 512;

template class Block<int> ;
int forward_DCT( Block<it> &b);
int forward_DCT( const Block<int> &ref, Block<int> &out);

void
DCT_domain(const unsigned char *ref , const unsigned char *mtn ,
   int *di , int *vecs, const int rows ,
   const int cols  , const int sub_sample_factor)
{
  unsigned int counter = 0 ;
  int best_x , best_y , offset;

  static Blocks<int> b[MAX_C][MAX_R];

  for( int i = 0 ; i < rows ; i++)
    for( int j = 0; j < cols ; j++)
      b[i][j].copy_data( *(ref + j + i*cols), cols , rows);

  forward_DCT( b[0][0] );



  for( int r = 0 ; r < rows ; r += 8)
    for( int c = 0 ; c < cols ; c += 8)
      {
offset = c + r*cols ;
best_x = 0;
```

```
best_y = 0;


*(vecs + counter ) = best_x - c ; counter++ ;
*(vecs + counter ) = best_y - r ; counter++ ;

gen_difference_block( best_x - c ,
      best_y - r ,c,r,ref, mtn,
      di, cols,rows) ;
    }
}*/
//--------------------------------------------


const int F_MTD = 20 ;
const int NOISE_ADJ = 7;
void
Feature_Monotony(const unsigned char *ref ,
 const unsigned char *mtn ,
 int *di , int *vecs, const int rows ,
 const int cols ,
 const int sub_sample_factor)
{
  int best_x , best_y;
  long int old_MAE , new_MAE;
  int counter = 0 , offset, monotony, offset_p,
    R, point_monotony;
  int position , other_position, n, tmp, tmp_2,
    test_a, test_b, sum;
  int MAE_2 , old_MAE_2 ;

  // make feature using monotony operator
  const int new_rows = rows - 8;
  const int new_cols = cols - 8;
  const int centre = 4 + 4*cols;

  //--Clear index array...
  for( int init = 0 ; init < LIST_SIZE ; init++)
    {
      index[init] = 0;
      entry_index[init] = 0 ;
      point_bags[init] = NULL ;
    }
  //--

  //First col...
  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += *(ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= new_cols ; i++)
    {
      for( int j = 0 ; j < rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  - *(ref + offset - 1)
    + *(ref + offset + 7) ;
}
    }
 //first row ..... col wise.....
  for( int l = 0 ; l < cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += data[k*cols + l];

      sum_norm[l] = sum;
    }
 // Rest of the rows......col wise
  const int seven_cols_up = 7*cols ;

  for( int i = 0 ; i <= new_cols  ; i++)
    {
      for( int j = 1 ; j <= new_rows  ; j++)
{
  offset = i + j*cols ;

  sum_norm[offset] = sum_norm[offset - cols]
    - data[offset - cols]
    + data[offset + seven_cols_up ] ;
}
    }
  //---
  point pt , best_so_far, current_point ;

  for( int i = 0 ; i <= new_cols ; i++)
    // make histogram of values ...
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;
index[ sum_norm[offset] ]++;
      }

  tmp = index[0] ;
  index[0] = 0 ;

  for( int i = 1 ; i < LIST_SIZE ; i++)
    {
      //index[i] += index[i - 1 ];
      tmp_2 = index[i] ;
      index[i] = ( index[i] > 0 ) ? tmp : 0  ;
      tmp += tmp_2 ;
    }

  for( int i = 0 ; i <= new_cols ; i++)
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;

stuff[ (index[ sum_norm[offset] ]) +
     (entry_index[sum_norm[offset] ]) ] =  point(i,j);

entry_index[ sum_norm[offset] ]++ ;
      }
  //--
  // reusing sum_norm here
  for( int i = 0 ; i <= new_rows ; i++)
    {
      offset = i*cols ;

      for( int j = 0 ; j <= new_cols; j++)
{
  monotony = 0;
  n = 1;
  for( int k = 0; k < 8 ; k++)
    {
      offset_p = k*cols + offset + j ;

      for( int l = 0 ; l < 8 ; l++)
{
  monotony += ( *(ref + l + offset_p) <=
*(ref + centre + offset))
    ? n : 0 ;
  //n++;
}
    }
  monotony -= 1; // account for centre pixel;

  sum_norm[offset + j] = monotony;
}
    }
  //-----------------------------------------------

  for( int r = 0 ; r <= new_rows ; r += 8 )
    for( int c = 0 ; c <= new_cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
old_MAE = MAX_OUT;
old_MAE_2 = MAX_OUT;
offset = c + r*cols ;
R = 0;
monotony = 0;

//cout << c << "\t" << r << endl ;

for( int k = 0; k < 8 ; k++)
  {
    offset_p = k*cols + offset ;

    for( int l = 0 ; l < 8 ; l++)
      {
monotony += ( (mtn[ l + offset_p]) <=
      (mtn[ centre + offset]))
  ? 1 : 0 ;
R += mtn[offset_p + l] ;
      }
  }

monotony -= 1; // account for centre pixel;
```

```
position = R;
other_position = R - 1;
//------
while( ( (R - old_MAE) <= position) &&
       ( (R + old_MAE) >= position) && (position < LIST_SIZE )){

  if( (entry_index[position] != 0) /*&& ( position < LIST_SIZE)*/)
    {
      old_MAE_2 = MAX_OUT;// Reset for this set of sum norms

      for( int q = 0 ; q < entry_index[position]; q++)
{
  point_monotony = sum_norm[stuff[ index[position] + q].x() +
    stuff[ index[position] + q].y()*cols] ;

  MAE_2 = point_monotony - monotony ;
  MAE_2 = ( MAE_2 < 0) ? -MAE_2 : MAE_2 ;

  if( MAE_2 <= old_MAE_2){

    old_MAE_2 = MAE_2;

    /*if( ( point_monotony <= (monotony + NOISE_ADJ)) &&
      ( point_monotony >= (monotony - NOISE_ADJ)) ){*/
    new_MAE = MAE( stuff[ index[position] + q].x(),
 stuff[ index[position] + q].y(),
 c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = ( best_so_far - pt).mod() ;
  test_b = ( stuff[ index[position] + q] - pt).mod();

  best_so_far = ( test_a > test_b )
    ? stuff[ index[position] + q] : best_so_far ;

/*cout << test_a << " next" << test_b << " pt" << pt
  << " best " << best_so_far << endl ;*/
}
  }
  //if( (position - R) > F_MTD)break;// only search
  //closest points
}
    }
  position++;
}

while( ( (R - old_MAE) <= other_position) &&
       ( (R + old_MAE) >= other_position) &&
       (other_position >= 0 ) && (other_position != position)){
  if( entry_index[other_position] != 0)
    {
      old_MAE_2 = MAX_OUT;// Reset for this set of sum norms

      for( int q = 0 ; q < entry_index[other_position] ; q++)
{
  point_monotony =
    sum_norm[stuff[ index[other_position] + q].x() +
    stuff[ index[other_position] +
q].y()*cols] ;

   MAE_2 = point_monotony - monotony ;
  MAE_2 = ( MAE_2 < 0) ? -MAE_2 : MAE_2 ;

  if( MAE_2 <= old_MAE_2){

    old_MAE_2 = MAE_2;
    /*if( ( point_monotony <= (monotony + NOISE_ADJ)) &&
      ( point_monotony >= (monotony - NOISE_ADJ)) ){*/
 new_MAE = MAE( stuff[ index[other_position] + q].x(),
stuff[ index[other_position] + q].y(),
c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[other_position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = ( best_so_far - pt).mod() ;
  test_b = ( stuff[ index[other_position] + q]
    - pt).mod()  ;

  best_so_far = ( test_b < test_a )
    ? stuff[ index[other_position] + q] :
    best_so_far ;
      }
}
  //if((R - other_position) > F_MTD) break;// only
  //search closest points
}
    }
  other_position--;

  //cout << old_MAE << endl ;

}
//----

//cout << c << " " << r << endl ;
*(vecs + counter ) = best_so_far.x() - c; counter++ ;
*(vecs + counter ) = best_so_far.y() - r; counter++ ;
gen_difference_block( best_so_far.x() - c, best_so_far.y() - r,
      c,r,ref, mtn, di, cols,rows) ;
      }
}
//-----------------------------------------*

const int MTD = 20 ;

void
Feature_Modified_Monotony(const unsigned char *ref ,
  const unsigned char *mtn ,
  int *di , int *vecs,
  const int rows ,const int cols,
  const int sub_sample_factor)
{
  int best_x , best_y;
  long int old_MAE , new_MAE;
  int counter = 0 , offset, monotony,
    offset_p, R, point_monotony;
  int position , other_position, test_a,
    test_b, tmp, tmp_2, sum,n;
  int MAE_2 , old_MAE_2 ;

  //--Clear index array...
  for( int init = 0 ; init < LIST_SIZE ; init++)
    {
      index[init] = 0;
      entry_index[init] = 0 ;
    }
  //--
  // make feature using monotony operator
  const int new_rows = rows - 8;
  const int new_cols = cols - 8;
  const int centre = 4 + 4*cols;

  //--

  //First col...
  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += *(ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= new_cols ; i++)
    {
      for( int j = 0 ; j < rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  - *(ref + offset - 1)
    + *(ref + offset + 7) ;
}
    }
 //first row ..... col wise.....
  for( int l = 0 ; l < cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += data[k*cols + l];

      sum_norm[l] = sum;
    }
 // Rest of the rows......col wise
  const int seven_cols_up = 7*cols ;
```

```
  for( int i = 0 ; i <= new_cols  ; i++)
    {
      for( int j = 1 ; j <= new_rows  ; j++)
{
  offset = i + j*cols ;

  sum_norm[offset] = sum_norm[offset - cols]
    - data[offset - cols]
    + data[offset + seven_cols_up ] ;
}
    }
  //---
  point pt , best_so_far, current_point ;

  for( int i = 0 ; i <= new_cols ; i++)
    // make histogram of values ...
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;
index[ sum_norm[offset] ]++;
      }

  tmp = index[0] ;
  index[0] = 0 ;

  for( int i = 1 ; i < LIST_SIZE ; i++)
    {
      //index[i] += index[i - 1 ];
      tmp_2 = index[i] ;
      index[i] = ( index[i] > 0 ) ? tmp : 0  ;
      tmp += tmp_2 ;
    }

  for( int i = 0 ; i <= new_cols ; i++)
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;

stuff[ (index[ sum_norm[offset] ]) +
     (entry_index[sum_norm[offset] ]) ] =  point(i,j);

entry_index[ sum_norm[offset] ]++ ;
      }
  //--
  // reusing sum_norm here

  for( int i = 0 ; i <= new_rows ; i++)
    {
      offset = i*cols ;

      for( int j = 0 ; j <= new_cols; j++)
{
  monotony = 0;
  n = 1;
  for( int k = 0; k < 8 ; k++)
    {
      offset_p = k*cols + offset + j ;

      for( int l = 0 ; l < 8 ; l++)
{
  monotony += ( (ref[ l + offset_p]) <=
(ref[ centre + offset]))
    ? n : 0 ;
  n++;
}
    }
  monotony -= 36; // account for centre pixel;

  sum_norm[offset + j] = monotony;
}
    }
  //-----------------------------------------------

  for( int r = 0 ; r < rows ; r += 8 )
    for( int c = 0 ; c < cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
old_MAE = MAX_OUT;
old_MAE_2 = MAX_OUT ;
offset = c + r*cols;
R = 0;
monotony = 0;

//cout << c << "\t" << r << endl ;
n = 1;
for( int k = 0; k < 8 ; k++)
  {
    for( int l = 0 ; l < 8 ; l++)
      {
monotony += ( (mtn[ l + k*cols + offset]) <=
      (mtn[ centre + offset]) )
  ? n : 0 ;
n++;
R += mtn[ l + k*cols + offset];
      }
    }
monotony -= 1; // account for centre pixel;

//if(R < 0) cout << "R<0" << endl;

position = R;
other_position = R - 1;

//cout << "hererere" << endl ;
//------
while( ((R - old_MAE) <= position) &&
       ((R + old_MAE) >= position) && (position < LIST_SIZE )){

  if( (entry_index[position] != 0))
    {
      old_MAE_2 = MAX_OUT;// Reset for this set of sum norms

      for( int q = 0 ; q < entry_index[position]; q++)
{
  point_monotony = sum_norm[stuff[ index[position] + q].x() +
    stuff[ index[position] + q].y()*cols] ;

 MAE_2 = point_monotony - monotony ;
  MAE_2 = ( MAE_2 < 0) ? -MAE_2 : MAE_2 ;

  if( MAE_2 <= old_MAE_2){

    old_MAE_2 = MAE_2;
    /*if( ( point_monotony <= (monotony + 35*NOISE_ADJ)) &&
      ( point_monotony >= (monotony - 35*NOISE_ADJ)) ){*/
  new_MAE = MAE( stuff[ index[position] + q].x(),
 stuff[ index[position] + q].y(),
 c,r, ref , mtn , cols , rows);

  //cout << q << endl;

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = ( best_so_far - pt).mod() ;
  test_b = ( stuff[ index[position] + q] - pt).mod();

  best_so_far = ( test_a > test_b )
    ? stuff[ index[position] + q] : best_so_far ;

/*cout << test_a << " next" << test_b << " pt" << pt
  << " best " << best_so_far << endl ;*/
}
}
  //if( (position - R) > MTD)break;//only search
  //closest matcehs to monotony perator
}
    }
  position++;
}
//cout << "22222hererere" << endl ;
while( ( (R - old_MAE) <= other_position) &&
       ( (R + old_MAE) >= other_position) &&
       (other_position >= 0) && (other_position != position) ){
  if( entry_index[other_position] != 0)
    {
      old_MAE_2 = MAX_OUT;// Reset for this set of sum norms

      for( int q = 0 ; q < entry_index[other_position] ; q++)
{
  point_monotony =
    sum_norm[stuff[ index[other_position] + q].x() +
    stuff[ index[other_position] +
 q].y()*cols] ;

  MAE_2 = point_monotony - monotony ;
  MAE_2 = ( MAE_2 < 0) ? -MAE_2 : MAE_2 ;

  if( MAE_2 <= old_MAE_2){

    old_MAE_2 = MAE_2;

  /*if( ( point_monotony <= (monotony + NOISE_ADJ)) &&
      ( point_monotony >= (monotony - NOISE_ADJ)) ){*/
```

```
  new_MAE = MAE( stuff[ index[other_position] + q].x(),
 stuff[ index[other_position] + q].y(),
 c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[other_position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = ( best_so_far - pt).mod() ;
  test_b = ( stuff[ index[other_position] + q] - pt).mod()  ;

  best_so_far = ( test_b < test_a )
    ? stuff[ index[other_position] + q] :
    best_so_far ;
      }
  }
      //if((R - other_position) > MTD)break;
  // only search closest matcehs to monotony perator
}
    }
  other_position--;
}
  //cout << old_MAE << endl ;


//----

//cout << c << " " << r << endl ;
*(vecs + counter ) = best_so_far.x() - c ; counter++ ;
*(vecs + counter ) = best_so_far.y() - r ; counter++ ;
gen_difference_block( best_so_far.x() - c, best_so_far.y() - r,
      c,r,ref, mtn, di, cols,rows) ;
      }
}
//-----------------------------------------*
void
sampled_pts_prefilt(const unsigned char *ref ,
    const unsigned char *mtn ,
    int *di , int *vecs,const int rows ,
    const int cols,const int sub_sample)
{
  int offset ;
  long int sum = 0, R = 0;

  long int position, other_position ,
    tmp , tmp_2, old_MAE, new_MAE;
  long int test_a , test_b ;

  unsigned int counter = 0;

  const int new_cols = cols - 8 , new_rows = rows - 8 ;
  const int f_cols = cols - sub_sample ,
    f_rows = rows - sub_sample;

  //--Clear index array...
  for( int init = 0 ; init < LIST_SIZE ; init++)
    {
      index[init] = 0;
      entry_index[init] = 0 ;
    }
  //-- For scratch pad ---

  //First col...
  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < sub_sample ; k++)
sum += *(ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= f_cols ; i++)
    {
      for( int j = 0 ; j < rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  - *(ref + offset - 1)
    + *(ref + offset + sub_sample - 1) ;
}
    }
 //first row ..... col wise.....
  for( int l = 0 ; l < cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < sub_sample ; k++)
sum += data[k*cols + l];

      scratch[l] = ( sum + sub_sample*sub_sample/2)
/(sub_sample*sub_sample);
    }
 // Rest of the rows......col wise
  const int sub_sam_cols_up = (sub_sample - 1)*cols ;

  for( int i = 0 ; i <= f_cols  ; i++)
    {
      for( int j = 1 ; j <= f_rows ; j++)
{
  offset = i + j*cols ;

  scratch[offset] =
    (scratch[offset - cols]*sub_sample*sub_sample
     - data[offset - cols]
     + data[offset + sub_sam_cols_up ]
     + sub_sample*sub_sample/2)
/(sub_sample*sub_sample);
}
    }

 //  for( int i = 0 ; i < f_cols ; i++)//// Normalize back to usual
//       //sumnorm range.
//     {
//       for( int j = 0 ; j < f_rows ; j++)
//  {
//    offset = i + j*cols ;
//    scratch[offset] = (scratch[offset] + sub_sample*sub_sample/2
//       )/(sub_sample*sub_sample);
//  }
//     }
  //---

  // Generate sum norm for scratch----------------
  //First col...
  for( int l = 0 ; l < f_rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += scratch[k + offset];

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= new_cols ; i++)
    {
      for( int j = 0 ; j < f_rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  - scratch[ offset - 1]
    + scratch[ offset + 7] ;
}
    }
 //first row ..... col wise.....
  for( int l = 0 ; l < f_cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += data[k*cols + l];

      sum_norm[l] = sum;
    }
 // Rest of the rows......col wise
  const int seven_cols_up = 7*cols ;

  for( int i = 0 ; i <= new_cols  ; i++)
    {
      for( int j = 1 ; j <= new_rows  ; j++)
{
  offset = i + j*cols ;

  sum_norm[offset] = sum_norm[offset - cols]  - data[offset - cols]
    + data[offset + seven_cols_up ] ;
}
    }
  //---
  point pt , best_so_far;
  for( int i = 0 ; i <= new_cols ; i += sub_sample)
    for( int j = 0 ; j <= new_rows ; j += sub_sample)
```

```
      {
offset = i + j*cols ;
index[ sum_norm[offset] ] += 1;
      }

  tmp = index[0] ;
  index[0] = 0 ;

  for( int i = 1 ; i < LIST_SIZE ; i++)
    {
      //index[i] += index[i - 1 ];
      tmp_2 = index[i] ;
      index[i] = tmp ;
      tmp += tmp_2 ;
    }

  for( int i = 0 ; i <= new_cols ; i += sub_sample)
    for( int j = 0 ; j <= new_rows ; j += sub_sample)
      {
offset = i + j*cols ;

stuff[ index[ sum_norm[offset] ] +
     entry_index[sum_norm[offset] ] ] =
  pt(i,j);

entry_index[ sum_norm[offset] ] =
  entry_index[ sum_norm[offset] ] + 1;
      }

  //--

  int stop = 0;
  int not_found = 0 ;

  for( int r = 0 ; r < rows ; r += 8)
    for( int c = 0 ; c < cols ; c += 8)
      {
offset = c + r*cols ;

// Calc R
R = 0 ; // Reset R.

for( int m = 0 ; m < 8 ; m++)
  for( int n = 0 ; n < 8 ; n++)
    R += *(mtn + offset + m + n*cols );

//---Find mtn vectors----

pt( c,r );
best_so_far(c,r);
not_found = 1;
stop = 0;
old_MAE = MAX_OUT;

// cout << r << "\t" << c << endl ;

old_MAE = MAE( c, r, c, r, ref, mtn, cols, rows);
if( old_MAE  == 0 )
  {
    best_so_far = pt ;
  }
else
  {
    position = R;
    other_position = R - 1;

    while( ( (R - old_MAE) <= position) &&
   ( (R + old_MAE) >= position) &&( position < LIST_SIZE)

   /*( (R - old_MAE) <= other_position) &&
   ( (R + old_MAE) >= other_position)*/){

      if( (entry_index[position] != 0)/* &&
  ( position < LIST_SIZE)*/){

      for( int q = 0 ; q < entry_index[position]; q++)
{
  new_MAE = MAE( stuff[ index[position] + q].x(),
 stuff[ index[position] + q].y(),
 c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = (best_so_far - pt).mod() ;
  test_b = (stuff[ index[position] + q] - pt).mod() ;

  best_so_far = ( test_a > test_b )
    ? stuff[ index[position] + q] :
    best_so_far ;
}
}
      }
      position++;
    }

    while( /*( (R - old_MAE) <= position) &&
   ( (R + old_MAE) >= position) &&*/

  ( (R - old_MAE) <= other_position) &&
  ( (R + old_MAE) >= other_position) &&
  ( other_position >= 0)){
      if( (entry_index[other_position] != 0) &&
  /*( other_position >= 0) &&*/ (position != other_position) ){

for( int q = 0 ; q < entry_index[other_position] ; q++)
  {
    new_MAE = MAE( stuff[ index[other_position] + q].x(),
   stuff[ index[other_position] + q].y(),
   c,r, ref , mtn , cols , rows);

    if( new_MAE < old_MAE)
      {
best_so_far = stuff[ index[other_position] + q];
old_MAE = new_MAE ;
      }else
if( new_MAE == old_MAE)
  {
    test_a = (best_so_far - pt).mod() ;
    test_b = (stuff[ index[other_position] +
   q] - pt).mod() ;

    best_so_far = (test_a > test_b )
      ? stuff[ index[other_position] + q] :
      best_so_far ;
  }
  }
      }
      //position++;
      other_position--;
    }
  }
//----

point best_point(best_so_far) ;
int max = MAX_OUT, new_pos , length ;
int old_length = ( best_so_far.x() - c)*( best_so_far.x()- c)
  + (best_so_far.y() - r)*( best_so_far.y() - r);

const int range = sub_sample - 1;

for( int ly = best_so_far.y() - range;
     ly <= (best_so_far.y() + range) ; ly++)
  for( int lx = best_so_far.x() - range ;
       lx <= (best_so_far.x() + range) ; lx++)
    {
      if( (ly >= 0) && ( ly <= new_rows) &&
  (lx >= 0) && ( lx <= new_cols) )
{
  new_pos = MAE( lx , ly ,
 c,//best_so_far.x() ,
 r,//best_so_far.y() ,
 ref, mtn,
 cols, rows );

  if( new_pos < max )
    {
      best_point = point( lx , ly ) ;
      max = new_pos ;
    }
  else if( (new_pos == max) && ( new_pos != MAX_OUT))
    {
      length = (lx - c)*(lx - c) + (ly - r)*(ly - r);

      if( length < old_length)
{
  old_length = length;
  best_point( lx , ly );
}
    }
}
    }

//----
*(vecs + counter ) = (best_point.x() - c) ; counter++ ;
*(vecs + counter ) = (best_point.y() - r); counter++ ;
```

```
gen_difference_block(best_point.x() - c ,best_point.y() - r ,
     c,r,ref, mtn, di, cols,rows) ;
//forward_dct.perform_dct( 8 , 8, *di , *di , r , c  ) ;
      }
}
//-------------------------------------------
void
projection_search_2(const unsigned char *ref ,
    const unsigned char *mtn ,
  int *di , int *vecs, const int rows ,
  const int cols,const int sub_sample_factor)
{
  int offset , tmp;
  int R1,R2,R3,R4;
  int F;
  int old_MAE = MAX_OUT , new_MAE = MAX_OUT ;
  int new_feature_MAE , old_feature_MAE = MAX_OUT ;
  int length , length_old , step;
  int log = 0;

  int trial_val[9] ;

  //long int dist_old , dist_new ;//<><><><

  unsigned int counter = 0;

  const int new_cols = cols - 8 , new_rows = rows - 8 ;

  int q1 , q2, q3 , q4 ,sum;
  //int b1, b2, b3, b4 ;
  point last_upper , last_lower;

  //--Clear index array...
  for( int init = 0 ;
       init < (LIST_SIZE*LIST_SCALE_FACTOR) ; init++)
    {
      index[init] = 0;
      point_bags[init] = NULL ;
      entry_index[init] = 0 ;
    }
  //--

  // making feature-----------

  const int N = 4;
  const int N_1_cols_up = ( N - 1)*cols;

  cout << "where " <<endl ;//First col...

  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < N ; k++)
sum += *(ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= (new_cols + 4); i++)
    {
      for( int j = 0 ; j < rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  - *(ref + offset - 1)
    + *(ref + offset + N - 1) ;
}
    }
  //first row ..... col wise.....
  for( int l = 0 ; l < cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < N ; k++)
sum += data[k*cols + l];

      scratch[l] = sum;
    }
  // Rest of the rows......col wise

  for( int i = 0 ; i <= (new_cols + 4) ; i++)
    {
      for( int j = 1 ; j <= (new_rows + 4) ; j++)
{
  offset = i + j*cols ;

  scratch[offset] = scratch[offset - cols]  -
    data[offset - cols] + data[offset + N_1_cols_up ] ;
}
    }
  //---

  cout << "1hxvciwcvwccc!" << endl;////////////////////////////////

  for( int j = 0 ; j <= new_rows ; j++)
    for( int i = 0 ; i <= new_cols ; i++)
      {
offset = i + j*cols ;

q1 = scratch[ offset ] ;
q2 = scratch[ offset + 4 ];
q3 = scratch[ offset + 4*cols ];
q4 = scratch[ offset + 4 + 4*cols ];

(features[offset])[0] = q1 + q2 + q3 + q4 ;
(features[offset])[1] = q1 + q2 - q3 - q4 ;
(features[offset])[2] = q1 + q3 - q2 - q4;
(features[offset])[3] = q1 - q2 - q3 + q4 ;

sum_norm[offset] =  (features[offset])[0];

tmp = ((  ((features[offset])[0]<<9) +
  ((features[offset])[1]<<8) +
  ((features[offset])[2]<<8) +
  (features[offset])[3] )>>4) ;
// calc hash function.
data[offset] = tmp ;
index[ tmp]++ ;
      }

  //---
 cout << "2hxvciwcvwccc!" << endl;////////////////////

  point pt , best_so_far, candidate;

  for( int i = 0 ;
       i < (LIST_SIZE*LIST_SCALE_FACTOR) ; i++)
    {
      point_bags[i] = new Bag<point>(index[i]) ;

      if( point_bags[i] == NULL)
{
  cerr << "Unable to allocate bag !" << endl;
  exit(1);
}
    }
 cout << "3hxvciwcvwccc!" << endl;////////////////////////////////

 for( short int j = 0 ; j <= new_rows ; j++)
   for( short int i = 0 ; i <= new_cols ; i++)
     {
offset = i + j*cols ;

//cout << offset << " do: " << data[offset] << endl;/////////

// cout <<point_bags[ data[offset] ] << endl;/////////////////

point_bags[ data[offset] ]->insert_item( point(i,j) ) ;
      }

  cout << "end section make feature!" << endl ;

  for( int r = 0 ; r < rows ; r += 8)
    for( int c = 0 ; c < cols ; c += 8)
      {

//cout << c << "\t" << r << endl ;

offset = c + r*cols ;

// Calc R
q1 = sum_up( c  ,r  , c+3, r+3,cols, mtn) ;
q2 = sum_up( c+4,r  , c+7, r+3,cols, mtn) ;
q3 = sum_up( c  ,r+4, c+3, r+7,cols, mtn) ;
q4 = sum_up( c+4,r+4, c+7, r+7,cols, mtn) ;

R1 = q1 + q2 + q3 + q4 ;
R2 = q1 + q2 - q3 - q4 ;
R3 = q1 + q3 - q2 - q4 ;
R4 = q1 - q2 - q3 + q4 ;

F = ( 512*R1 + 256*R2 + 256*R3 + R4 )/16 ;

step = 0;////////////////////

//---Find mtn vectors----
```

```
old_MAE = MAX_OUT ;
new_MAE = MAX_OUT + 2;
old_feature_MAE = MAX_OUT ;
new_feature_MAE = MAX_OUT + 2;
length_old = MAX_OUT;

best_so_far(c,r);
pt( c, r );
log++;

//-------------------------------------

while( old_feature_MAE >= step){

  step = ( step > 255*64) ? 255*64 : step ;// set limits

  trial_val[0] = F - (step>>4) ;
  trial_val[1] = F + (step>>4) ;

  trial_val[2] = F - (step<<4) ;
  trial_val[3] = F + (step<<4) ;

  trial_val[4] = F - (step<<5) ;
  trial_val[5] = F + (step<<5) ;

  //trial_val[6] = F - (step<<5) ;
  // trial_val[7] = F + (step<<5) ;

  for( int k = 0 ; k < 6 ; k++)
    {
      tmp = trial_val[k] ;

      if( entry_index[tmp] >= log )
{
  continue;
}

      entry_index[tmp] = log ;// mark pon as read
      if( tmp >=  LIST_SIZE*LIST_SCALE_FACTOR)
break ;

      if( tmp < 0)
{
  //cerr << "Hash function negative!" << endl;
  continue;
  //exit(1);
}

      if( point_bags[tmp]->get_size() > 0)
{
  for( int t = 0 ; t < point_bags[tmp]->get_size()
 ; t++)
    {
      candidate = point_bags[tmp]->get_element(t);

      new_feature_MAE = feature_MAE( candidate.x(),
     candidate.y(),
     c,r,
     features,
     cols , rows);
      if( new_feature_MAE <= old_feature_MAE)
{
  old_feature_MAE = new_feature_MAE;

  new_MAE = MAE( candidate.x(),
 candidate.y(),
 c,r, ref, mtn,
 cols, rows );

  if( new_MAE < old_MAE)
    {
      old_MAE = new_MAE;
      best_so_far = candidate;
    }
  else if( new_MAE == old_MAE)
    {
      length = (candidate.x() - c)*
(candidate.x() - c) +
(candidate.y() - r)*
(candidate.y() - r);

      length_old = (best_so_far.x() - c)*
(best_so_far.x() - c) +
(best_so_far.y() - r)*
(best_so_far.y() - r);

      if( length < length_old)
{
  best_so_far = candidate;
}
    }
}
    }
}
    }
}
    }

  step++; // old step*3 ; step + 16
}

//-------------------------------------

//cout << "final" << old_MAE <<
//" ofm" << old_feature_MAE << endl ;/////////

*(vecs + counter ) = best_so_far.x() - c ; counter++ ;
*(vecs + counter ) = best_so_far.y() - r ; counter++ ;

gen_difference_block( best_so_far.x() - c ,
      best_so_far.y() - r ,
      c,r,ref, mtn,
      di, cols,rows) ;

      }
  for( int i = 0 ;
       i < (LIST_SIZE*LIST_SCALE_FACTOR) ; i++)
    {
      delete point_bags[i] ;
      point_bags[i] = NULL ;
    }
}
//-------------------------------------------

//-------------------------------------------
void
Feature_variance(const unsigned char *ref ,
 const unsigned char *mtn ,
 int *di , int *vecs, const int rows ,
 const int cols,const int sub_sample_factor)
{
  int best_x , best_y;
  long int old_MAE , new_MAE;
  int counter = 0 , offset, variance, offset_p,
    R, point_variance;
  int position , other_position, tmp, tmp_2,
    test_a, test_b, sum;
  int MAE_2 , old_MAE_2 , sq ;

  // make feature using monotony operator
  const int new_rows = rows - 8;
  const int new_cols = cols - 8;

  //--Clear index array...
  for( int init = 0 ; init < LIST_SIZE ; init++)
    {
      index[init] = 0;
      entry_index[init] = 0 ;
      point_bags[init] = NULL ;
    }
  //--

  //First col...
  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += *(ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= new_cols ; i++)
    {
      for( int j = 0 ; j < rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  - *(ref + offset - 1)
    + *(ref + offset + 7) ;
}
    }
 //first row ..... col wise.....
  for( int l = 0 ; l < cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += data[k*cols + l];
```

```
      sum_norm[1] = sum;
    }
 // Rest of the rows.....col wise
  const int seven_cols_up = 7*cols ;

  for( int i = 0 ; i <= new_cols  ; i++)
    {
      for( int j = 1 ; j <= new_rows  ; j++)
{
  offset = i + j*cols ;

  sum_norm[offset] = sum_norm[offset - cols]
    - data[offset - cols]
    + data[offset + seven_cols_up ] ;
}
    }
  //---
  point pt , best_so_far, current_point ;

  for( int i = 0 ; i <= new_cols ; i++)
    // make histogram of values ...
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;
index[ sum_norm[offset] ]++;
      }

  tmp = index[0] ;
  index[0] = 0 ;

  for( int i = 1 ; i < LIST_SIZE ; i++)
    {
      //index[i] += index[i - 1 ];
      tmp_2 = index[i] ;
      index[i] = ( index[i] > 0 ) ? tmp : 0  ;
      tmp += tmp_2 ;
    }

  for( int i = 0 ; i <= new_cols ; i++)
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;

stuff[ (index[ sum_norm[offset] ]) +
     (entry_index[sum_norm[offset] ]) ]
  =  point(i,j);

entry_index[ sum_norm[offset] ]++ ;
      }
  //--
  // reusing sum_norm here
  for( int i = 0 ; i <= new_rows ; i++)
    {
      offset = i*cols ;

      for( int j = 0 ; j <= new_cols; j++)
{
  variance = 0;
  sq = 0;
  for( int k = 0; k < 8 ; k++)
    {
      offset_p = k*cols + offset + j ;

      for( int l = 0 ; l < 8 ; l++)
sq += ref[l + offset_p]*ref[l + offset_p] ;
    }

  sum_norm[offset + j] =
    sq - sum_norm[offset + j]*sum_norm[offset + j];
}
    }
  //-------------------------------------------------

  for( int r = 0 ; r <= new_rows ; r += 8 )
    for( int c = 0 ; c <= new_cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
old_MAE = MAX_OUT;
old_MAE_2 = MAX_OUT;
offset = c + r*cols ;
R = 0;
variance = 0;
sq = 0;

//cout << c << "\t" << r << endl ;

for( int k = 0; k < 8 ; k++)
  {
    offset_p = k*cols + offset ;

    for( int l = 0 ; l < 8 ; l++)
      {
sq += mtn[ l + offset_p]*mtn[ l + offset_p];

R += mtn[offset_p + l] ;
      }
  }

variance = sq - R*R ;
position = R;
other_position = R - 1;
//------
while( ( (R - old_MAE) <= position) &&
       ( (R + old_MAE) >= position) && (position < LIST_SIZE )){

  if( (entry_index[position] != 0) /*&& ( position < LIST_SIZE)*/)
    {
      old_MAE_2 = MAX_OUT;// Reset for this set of sum norms

      for( int q = 0 ; q < entry_index[position]; q++)
{
  point_variance = sum_norm[stuff[ index[position] + q].x() +
    stuff[ index[position] + q].y()*cols] ;

  MAE_2 = point_variance - variance ;
  MAE_2 = ( MAE_2 < 0) ? -MAE_2 : MAE_2 ;

  if( MAE_2 <= old_MAE_2){

    old_MAE_2 = MAE_2;

    /*if( ( point_monotony <= (monotony + NOISE_ADJ)) &&
      ( point_monotony >= (monotony - NOISE_ADJ)) ){*/
    new_MAE = MAE( stuff[ index[position] + q].x(),
 stuff[ index[position] + q].y(),
 c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = ( best_so_far - pt).mod() ;
  test_b = ( stuff[ index[position] + q] - pt).mod();

  best_so_far = ( test_a > test_b )
    ? stuff[ index[position] + q] : best_so_far ;

/*cout << test_a << " next" << test_b << " pt" << pt
  << " best " << best_so_far << endl ;*/
}
  }
  //if( (position - R) > F_MTD)break;// only search
  //closest points
}
    }
  position++;
}

while( ( (R - old_MAE) <= other_position) &&
       ( (R + old_MAE) >= other_position) &&
       (other_position >= 0 ) && (other_position != position)){
  if( entry_index[other_position] != 0)
    {
      old_MAE_2 = MAX_OUT;// Reset for this set of sum norms

      for( int q = 0 ; q < entry_index[other_position] ; q++)
{
  point_variance =
    sum_norm[stuff[ index[other_position] + q].x() +
    stuff[ index[other_position] +
 q].y()*cols] ;

   MAE_2 = point_variance - variance ;
  MAE_2 = ( MAE_2 < 0) ? -MAE_2 : MAE_2 ;

  if( MAE_2 <= old_MAE_2){

    old_MAE_2 = MAE_2;
    /*if( ( point_monotony <= (monotony + NOISE_ADJ)) &&
      ( point_monotony >= (monotony - NOISE_ADJ)) ){*/
 new_MAE = MAE( stuff[ index[other_position] + q].x(),
 stuff[ index[other_position] + q].y(),
 c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
```

```
    {
      best_so_far = stuff[ index[other_position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = ( best_so_far - pt).mod() ;
  test_b = ( stuff[ index[other_position] + q]
    - pt).mod()  ;

  best_so_far = ( test_b < test_a )
    ? stuff[ index[other_position] + q] :
    best_so_far ;
      }
}
  //if((R - other_position) > F_MTD) break;// only
  //search closest points
}
    }
  other_position--;

  //cout << old_MAE << endl ;

}
//----

//cout << c << " " << r << endl ;
*(vecs + counter ) = best_so_far.x() - c; counter++ ;
*(vecs + counter ) = best_so_far.y() - r; counter++ ;
gen_difference_block( best_so_far.x() - c,
      best_so_far.y() - r,
      c,r,ref, mtn, di, cols,rows) ;
      }
}
//-----------------------------------------*

void
Feature_energy(const unsigned char *ref,
       const unsigned char *mtn ,
       int *di , int *vecs, const int rows ,
       const int cols,const int sub_sample_factor)
{
  int best_x , best_y;
  long int old_MAE , new_MAE;
  int counter = 0 , offset, variance, offset_p,
    R, point_energy;
  int energy ;
  int position , other_position, tmp, tmp_2,
    test_a, test_b, sum;
  int MAE_2 , old_MAE_2 , sq ;

  // make feature using monotony operator
  const int new_rows = rows - 8;
  const int new_cols = cols - 8;

  //--Clear index array...
  for( int init = 0 ; init < LIST_SIZE ; init++)
    {
      index[init] = 0;
      entry_index[init] = 0 ;
      point_bags[init] = NULL ;
    }
  //--

  //First col...
  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += *(ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= new_cols ; i++)
    {
      for( int j = 0 ; j < rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1]  - *(ref + offset - 1)
    + *(ref + offset + 7) ;
}
    }
 //first row ..... col wise.....
  for( int l = 0 ; l < cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += data[k*cols + l];

      sum_norm[l] = sum;
    }
 // Rest of the rows......col wise
  const int seven_cols_up = 7*cols ;

  for( int i = 0 ; i <= new_cols  ; i++)
    {
      for( int j = 1 ; j <= new_rows  ; j++)
{
  offset = i + j*cols ;

  sum_norm[offset] = sum_norm[offset - cols]
    - data[offset - cols]
    + data[offset + seven_cols_up ] ;
}
    }
  //---
  point pt , best_so_far, current_point ;

  for( int i = 0 ; i <= new_cols ; i++)
    // make histogram of values ...
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;
index[ sum_norm[offset] ]++;
      }

  tmp = index[0] ;
  index[0] = 0 ;

  for( int i = 1 ; i < LIST_SIZE ; i++)
    {
      //index[i] += index[i - 1 ];
      tmp_2 = index[i] ;
      index[i] = ( index[i] > 0 ) ? tmp : 0  ;
      tmp += tmp_2 ;
    }

  for( int i = 0 ; i <= new_cols ; i++)
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;

stuff[ (index[ sum_norm[offset] ]) +
     (entry_index[sum_norm[offset] ]) ]
  =  point(i,j);

entry_index[ sum_norm[offset] ]++ ;
      }
  //--
  // reusing sum_norm here
  for( int i = 0 ; i <= new_rows ; i++)
    {
      offset = i*cols ;

      for( int j = 0 ; j <= new_cols; j++)
{
  variance = 0;
  sq = 0;
  for( int k = 0; k < 8 ; k++)
    {
      offset_p = k*cols + offset + j ;

      for( int l = 0 ; l < 8 ; l++)
sq += ref[l + offset_p]*ref[l + offset_p] ;
    }

  sum_norm[offset + j] =  sq ;
}
    }
  //-------------------------------------------------

  for( int r = 0 ; r <= new_rows ; r += 8 )
    for( int c = 0 ; c <= new_cols ; c += 8)
      {
best_x = 0 ;
best_y = 0 ;
old_MAE = MAX_OUT;
old_MAE_2 = MAX_OUT;
offset = c + r*cols ;
R = 0;
variance = 0;
sq = 0;

//cout << c << "\t" << r << endl ;
```

```
for( int k = 0; k < 8 ; k++)
  {
    offset_p = k*cols + offset ;

    for( int l = 0 ; l < 8 ; l++)
      {
sq += mtn[ l + offset_p]*mtn[ l + offset_p];

R += mtn[offset_p + l] ;
      }
  }

energy = sq ;
position = R;
other_position = R - 1;
//------
while( ( (R - old_MAE) <= position) &&
       ( (R + old_MAE) >= position) && (position < LIST_SIZE )){

  if( (entry_index[position] != 0) /*&& ( position < LIST_SIZE)*/)
    {
      old_MAE_2 = MAX_OUT;// Reset for this set of sum norms

      for( int q = 0 ; q < entry_index[position]; q++)
{
  point_energy = sum_norm[stuff[ index[position] + q].x() +
    stuff[ index[position] + q].y()*cols] ;

  MAE_2 = point_energy - energy ;
  MAE_2 = ( MAE_2 < 0) ? -MAE_2 : MAE_2 ;

  if( MAE_2 <= old_MAE_2){

    old_MAE_2 = MAE_2;

    new_MAE = MAE( stuff[ index[position] + q].x(),
 stuff[ index[position] + q].y(),
 c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = ( best_so_far - pt).mod() ;
  test_b = ( stuff[ index[position] + q] - pt).mod();

  best_so_far = ( test_a > test_b )
    ? stuff[ index[position] + q] : best_so_far ;
}
  }
}
    }
  position++;
}

while( ( (R - old_MAE) <= other_position) &&
       ( (R + old_MAE) >= other_position) &&
       (other_position >= 0 ) && (other_position != position)){
  if( entry_index[other_position] != 0)
    {
      old_MAE_2 = MAX_OUT;// Reset for this set of sum norms

      for( int q = 0 ; q < entry_index[other_position] ; q++)
{
  point_energy =
    sum_norm[stuff[ index[other_position] + q].x() +
    stuff[ index[other_position] +
q].y()*cols] ;

   MAE_2 = point_energy - energy ;
  MAE_2 = ( MAE_2 < 0) ? -MAE_2 : MAE_2 ;

  if( MAE_2 <= old_MAE_2){

    old_MAE_2 = MAE_2;

  new_MAE = MAE( stuff[ index[other_position] + q].x(),
stuff[ index[other_position] + q].y(),
c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[other_position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = ( best_so_far - pt).mod() ;
  test_b = ( stuff[ index[other_position] + q]
    - pt).mod()  ;

  best_so_far = ( test_b < test_a )
    ? stuff[ index[other_position] + q] :
    best_so_far ;
      }
}

}
    }
  other_position--;
}
//----

//cout << c << " " << r << endl ;
*(vecs + counter ) = best_so_far.x() - c; counter++ ;
*(vecs + counter ) = best_so_far.y() - r; counter++ ;
gen_difference_block( best_so_far.x() - c, best_so_far.y() - r,
      c,r,ref, mtn, di, cols,rows) ;
      }
}
//-----------------------------------------*
const int sqrt_2 = 141 ;

inline int sine( const int u , const int k)
{
  static double my_pi = acos(-1) ;
  static int log = 0 ;

  static int values[64] ;

  if( log == 0)
    {
      for( int a = 0 ; a < 8 ; a++ )
for( int b = 0 ; b < 8 ; b++)
  {
    values[a*8 + b] = ( (int)( 100*sin( (2*a + 1)*my_pi*b/16 ) )) ;
  }
    }

  return values[ u + k*8 ];
}

inline int sine( const int arg)
{
  static double my_pi = acos(-1) ;
  static int log = 0 ;

  static int values[64] ;

  if( log == 0)
    {
      for( int a = 0 ; a < 8 ; a++ )
for( int b = 0 ; b < 8 ; b++)
  {
    values[a*8 + b] = ( (int)( 100*sin( arg*my_pi/16 ) ));
  }
    }

  return values[ arg%16 ];
}

inline int cosine( const int u , const int k)
{
  static double my_pi = acos(-1) ;
  static int log = 0 ;

  static int values[64] ;

  if( log == 0)
    {
      for( int a = 0 ; a < 8 ; a++ )
for( int b = 0 ; b < 8 ; b++)
  {
    values[a*8 + b] = ( (int)( 100*cos( (2*a + 1)*my_pi*b/16 ) )) ;
  }
    }

  return values[ u + k*8 ];
}

inline int cosine( const int arg)
{
  static double my_pi = acos(-1) ;
  static int log = 0 ;

  static int values[64] ;
```

```
  if( log == 0)
    {
      for( int a = 0 ; a < 8 ; a++ )
for( int b = 0 ; b < 8 ; b++)
  {
    values[a*8 + b] = ( (int)( 100*cos( arg*my_pi/16 ))) ;
  }
    }

  return values[ arg%16 ];
}

inline void
row_dct( const unsigned char *start , int place[] ,
 const int coefficients)
{
  for( int k = 0 ; k < coefficients ; k++)
    {
      place[k] = 0;
      for( int u = 0 ; u < 7 ; u++)
{
  place[k] += start[u]*cosine(u,k);
}
      place[k] =
( k == 0) ? (place[k] + sqrt_2/2)/(sqrt_2) : place[k]/100 ;
    }
}

inline void
col_dct( const unsigned char *start , int place[] ,
 const int cols, const int coefficients   )
{
  for( int k = 0 ; k < coefficients ; k++)
    {
      place[k*8] = 0;
      for( int u = 0 ; u < 7 ; u++)
{
  place[k*8] += start[u*cols]*cosine(u,k);
}
      place[k*8] =
( k == 0) ? (place[k*8] + sqrt_2/2)/(sqrt_2) : place[k*8]/100 ;
    }
}

inline void
row_dst( const unsigned char *start , int place[] ,
 const int coefficients)
{
  for( int k = 0 ; k < coefficients ; k++)
    {
      place[k] = 0;
      for( int u = 0 ; u < 7 ; u++)
{
  place[k] += start[u]*sine(u,k);
}
      place[k] =
( k == 0) ? (place[k] + sqrt_2/2)/(sqrt_2) : place[k]/100 ;
    }
}

inline void
col_dst( const unsigned char *start , int place[] ,
 const int cols, const int coefficients   )
{
  for( int k = 0 ; k < coefficients ; k++)
    {
      place[k*8] = 0;
      for( int u = 0 ; u < 7 ; u++)
{
  place[k*8] += start[u*cols]*sine(u,k);
}
      place[k*8] =
( k == 0) ? (place[k*8] + sqrt_2/2)/(sqrt_2) : place[k*8]/100 ;
    }
}
void
feature_search_DCT(const unsigned char *ref ,
   const unsigned char *mtn ,
   int *di , int *vecs, const int rows ,
   const int cols,const int sub_sample_factor)
{
  int offset ;
  int R = 0;
  int old_MAE = MAX_OUT , new_MAE = MAX_OUT ;
  int new_feature_MAE , old_feature_MAE = MAX_OUT ;
  int position , other_position , tmp_2 , test_a , test_b ;

  //static vector<int,VECTOR_LENGTH>
  //sine_features[FRAME_SIZE_MAX] ;

  //long int dist_old , dist_new ;//<><><><

  unsigned int counter = 0;

  const int new_cols = cols - 8 , new_rows = rows - 8 ;

  int block[64] ;
  //int q1 , q2, q3 , q4 ;

  int tmp ;

  //--Clear index array...
  for( int init = 0 ; init < LIST_SIZE ; init++)
    {
      index[init] = 0;
      entry_index[init] = 0 ;
    }

  // making feature-----------

  //const int N = 4;
  // const int N_1_cols_up = ( N - 1)*cols;
  //------
  for( int j = 0 ; j <= new_rows ; j++)
    for( int i = 1 ; i <= new_cols ; i++)
      {
offset = i + j*cols ;

for(int k = 0 ; k < 8; k++)
  row_dct( (ref + k*cols + offset) , &block[k*8] , 3);

for(int k = 0 ; k < 2; k++) // only first 4 features around
    // origin used
  col_dct( (ref + k + offset) , &block[k] , cols, 3);

(features[offset])[0] = block[0];
(features[offset])[1] = block[1];
(features[offset])[2] = block[8];
(features[offset])[3] = block[9];

sum_norm[offset] = (features[offset])[0];;

cout << " ij:" << i << " " << j <<  endl ;
      }

  //---
  point pt , best_so_far;

  for( int i = 0 ; i <= new_cols ; i++)
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;
index[ sum_norm[offset] ] += 1;
      }

  tmp = index[0] ;
  index[0] = 0 ;

  for( int i = 1 ; i < LIST_SIZE ; i++)
    {
      //index[i] += index[i - 1 ];
      tmp_2 = index[i] ;
      index[i] = tmp ;
      tmp += tmp_2 ;
    }

  for( int i = 0 ; i <= new_cols ; i++)
    for( int j = 0 ; j <= new_rows ; j++)
      {
offset = i + j*cols ;

stuff[ index[ sum_norm[offset] ] +
     entry_index[sum_norm[offset] ] ] =
  pt(i,j);

entry_index[ sum_norm[offset] ] =
  entry_index[ sum_norm[offset] ] + 1;
      }

  cout << "end section make feature!" << endl ;

  //  const int SUB_WIDTH = 10;
  const int THRESHOLD = 300;

  for( int r = 0 ; r < rows ; r += 8)
    for( int c = 0 ; c < cols ; c += 8)
      {

cout << c << "\t" << r << endl ;
```

```
offset = c + r*cols ;

// Calc R
R = 0 ; // Reset R.

for(int k = 0 ; k < 8; k++)
  row_dct( (mtn + k*cols + offset) , &block[k*8] , 3);

for(int k = 0 ; k < 2; k++) // only first 4 features around
    // origin used
  col_dct( (mtn + k + offset) , &block[k] , cols, 3);

R = block[0] ;
//---Find mtn vectors----

old_MAE = MAX_OUT ;
new_MAE = 0;
old_feature_MAE = MAX_OUT ;
new_feature_MAE = 0;

best_so_far(c,r);
pt( c, r );
//dist_old = rows*rows + cols*cols;//<><><><

position = R;
other_position = R - 1;

/*/ do an initial search around the origin of the search block
for( int yy = r - SUB_WIDTH ; yy <= (r + SUB_WIDTH) ; yy++)
  {
    if((yy < 0) || ( yy > new_rows)) break;

    for( int xx = c - SUB_WIDTH ; xx <= (c + SUB_WIDTH) ; xx++)
      {
if( (xx < 0) || ( xx > new_cols)) break;

new_MAE = MAE( xx, yy, c,r, ref , mtn , cols ,
       rows);
if( new_MAE < old_MAE)
  {
    best_so_far( xx , yy );
    old_MAE = new_MAE;

  }else if( (new_MAE <= old_MAE) &&
    (new_MAE != MAX_OUT) &&
    ( (point( xx , yy ) - pt).mod() <
      (best_so_far - pt).mod() ) )
    {
      best_so_far( xx , yy );
      //old_MAE = new_MAE;
    }
      }
  }*/

/*old_MAE = MAE( best_so_far.x(),
       best_so_far.y(), c,r, ref , mtn , cols , rows);*/

//-------------------------------------

if( old_MAE > THRESHOLD){

  //cout << old_MAE << " " << R << " "
  //    << position << " " << other_position << endl;

while( ( (R - old_MAE) <= position) &&
       ( (R + old_MAE) >= position) && ( position < LIST_SIZE)

       /*( (R - old_MAE) <= other_position) &&
       ( (R + old_MAE) >= other_position)*/ ){

  if( (entry_index[position] != 0) /*&&
      ( position < LIST_SIZE)*/){

    for( int q = 0 ; q < entry_index[position]; q++)
      {
new_feature_MAE =
  feature_MAE( stuff[ index[position] + q].x(),
       stuff[ index[position] + q].y(),
       c,r,
       features,cols , rows);

if( new_feature_MAE <= old_feature_MAE ){

  old_feature_MAE = new_feature_MAE;

  new_MAE = MAE( stuff[ index[position] + q].x(),
 stuff[ index[position] + q].y(),
 c,r, ref , mtn , cols , rows);

  if( ((index[position] + q) < 0) ||
      ((index[position] + q) > (512*512)))
      {
cerr << "bounds!" << endl ;
      }

  if( new_MAE < old_MAE)
    {
      best_so_far = stuff[ index[position] + q];
      old_MAE = new_MAE ;
    }else
      if( new_MAE == old_MAE)
{
  test_a = (best_so_far - pt).mod() ;
  test_b =
    (stuff[ index[position] + q] - pt).mod() ;
  best_so_far = ( test_a > test_b )
    ? stuff[ index[position] + q] :
    best_so_far ;
}
  }
      }
  }
  position++;
}
while( /*( (R - old_MAE) <= position) &&
       ( (R + old_MAE) >= position) &&*/

       ( (R - old_MAE) <= other_position) &&
       ( (R + old_MAE) >= other_position) &&
       (other_position >= 0)){
  if( (entry_index[other_position] != 0) &&
      /*(other_position >= 0) && */(other_position != position) ){

    for( int q = 0 ; q < entry_index[other_position] ; q++)
      {
new_feature_MAE =
  feature_MAE( stuff[index[other_position]
    + q].x(),
      stuff[index[other_position]
    + q].y(),
      c,r,
      features,cols , rows);
if( new_feature_MAE <= old_feature_MAE ){

  old_feature_MAE = new_feature_MAE;

  if( ((index[other_position] + q) < 0) ||
     ((index[other_position] + q) > (512*512)))
      {
cerr << "bounds!" << endl ;
      }
  new_MAE =
    MAE( stuff[ index[other_position] + q].x(),
 stuff[ index[other_position] + q].y(),
 c,r, ref , mtn , cols , rows);

    if( new_MAE < old_MAE)
      {
best_so_far = stuff[ index[other_position] + q];
old_MAE = new_MAE ;
      }else
if( new_MAE == old_MAE)
  {
    test_a = (best_so_far - pt).mod() ;
    test_b = (stuff[ index[other_position] +
   q] - pt).mod() ;

    best_so_far = (test_a > test_b )
      ? stuff[ index[other_position] + q] :
      best_so_far ;
  }
    }
      }
  }
  //position++;
  other_position--;
}
}
*(vecs + counter ) = best_so_far.x() - c ; counter++ ;
*(vecs + counter ) = best_so_far.y() - r ; counter++ ;

gen_difference_block( best_so_far.x() - c ,
      best_so_far.y() - r ,c,r,ref, mtn,
      di, cols,rows) ;

      }
}
//-------------------------------------------
void
```

```
tile_search(const unsigned char *ref ,
    const unsigned char *mtn ,
    int *di , int *vecs, const int rows ,
    const int cols , const int dummy)
{
  int offset ;
  long int sum = 0, R = 0;

  int old_MAE = MAX_OUT , new_MAE ;

  long int position, other_position ;

  long int test_a , test_b ;

  int counter = 0, pos = 0, largest, tile_offset;

  const int new_cols = cols - 8 , new_rows = rows - 8 ;

  static short int search_pos_x[1600] , search_pos_y[1600] ;
  const int THRESHOLD = 500 ;

  //First col...
  for( int l = 0 ; l < rows ; l++)
    {
      offset = l*cols ;
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += *(ref + k + offset);

      data[offset] = sum;
    }
  //rest of the cols...row-wise...

  for( int i = 1 ; i <= new_cols  ; i++)
    {
      for( int j = 0 ; j < rows ; j++)
{
  offset = i + j*cols;

  data[offset] = data[offset - 1] - *(ref + offset - 1)
    + *(ref + offset + 7) ;
}
    }
 //first row ..... col wise.....
  for( int l = 0 ; l < cols ; l++)
    {
      sum = 0;

      for( int k = 0 ; k < 8 ; k++)
sum += data[k*cols + l];

      sum_norm[l] = sum;
    }
 // Rest of the rows.....col wise
  const int seven_cols_up = 7*cols ;

  for( int i = 0 ; i <= new_cols  ; i++)
    {
      for( int j = 1 ; j <= new_rows  ; j++)
{
  offset = i + j*cols ;

  sum_norm[offset] = sum_norm[offset - cols]
    - data[offset - cols]
    + data[offset + seven_cols_up ] ;
}
    }
  //---
  point pt , best_so_far;

   //cout << "HERERERERERERERERERERERE " << R << endl ;

  for( int r = 0 ; r < rows ; r += 8)
    for( int c = 0 ; c < cols ; c += 8)
      {
offset = c + r*cols ;

// Calc R
R = 0 ; // Reset R.

for( int m = 0 ; m < 8 ; m++)
  for( int n = 0 ; n < 8 ; n++)
    R += *( mtn + offset + m + n*cols );

//---Find mtn vectors----

pt( c,r );
pos = 0;

old_MAE = MAX_OUT ;
position = R;
other_position = R - 1;

largest = ( cols < rows ) ? rows : cols ;

//cout << c << " " << r << endl ;////////////

for( int log = 0 ; log < largest/8; log++){

  pos = 0 ;

  for( int i = 1 ; i < log ; i++ )
    {
      pos++;
      search_pos_x[pos] = c + i*8 ;
      search_pos_y[pos] = r + log*8 ;
      // lower right hand cnr across
      pos++;
      search_pos_x[pos] = c - i*8 ;
      search_pos_y[pos] = r + log*8 ;
      // lower right hand cnr across
      //------------
      pos++;
      search_pos_x[pos] = c + i*8 ;
      search_pos_y[pos] = r - log*8 ;
      // upper right hand cnr across
      pos++;
      search_pos_x[pos] = c - i*8 ;
      search_pos_y[pos] = r - log*8 ;
      // upper right hand cnr across
      //------------
      pos++;
      search_pos_x[pos] = c + log*8 ;
      search_pos_y[pos] = r - i*8 ;
      // upper right hand cnr down

      pos++;
      search_pos_x[pos] = c + log*8 ;
      search_pos_y[pos] = r + i*8 ;
      // upper right hand cnr down

      //------------
      pos++;
      search_pos_x[pos] = c - log*8 ;
      search_pos_y[pos] = r - i*8 ;
      // lower right hand cnr down

      pos++;
      search_pos_x[pos] = c - log*8 ;
      search_pos_y[pos] = r + i*8 ;
      // lower right hand cnr down
    }
  search_pos_x[0] = c ;
  search_pos_y[0] = r ;
  // always search centre tile.


  //cout << "gudvhicvihcvihcvqihvc" << endl ;

  for( int i = 0 ; i <= pos ; i++)
    {
      for( int y = -4 ; y < 3 ; y++)
{
  if( (search_pos_y[i] + y) < 0) continue ;
  else
    if( (search_pos_y[i] + y) > rows) break ;

  for( int x = -4 ; x < 3 ; x++)
    {
      if( (search_pos_x[i] + x) < 0) continue ;
      else
if( (search_pos_x[i] + x) > cols) break ;

      tile_offset = search_pos_x[i] + x +
( search_pos_y[i] + y)*cols ;

      if( ((R - old_MAE) <= sum_norm[tile_offset]) &&
  ((R + old_MAE) >= sum_norm[tile_offset]))
{
  new_MAE = MAE( search_pos_x[i] + x,
 search_pos_y[i] + y,
 c,r, ref , mtn , cols , rows);

  if( new_MAE < old_MAE)
    {
      best_so_far = point(search_pos_x[i] + x,
  search_pos_y[i] + y );
      old_MAE = new_MAE ;
    }else
```

```
       if( new_MAE == old_MAE)
  {
    test_a = ( best_so_far - pt).mod() ;
    test_b = ( point(search_pos_x[i] + x,
     search_pos_y[i] + y) -
       pt).mod();

    best_so_far = ( test_a > test_b )
      ?  point(search_pos_x[i] + x,
       search_pos_y[i] + y) :
      best_so_far ;
  }
  }
      }
  }
      }

    if( old_MAE < THRESHOLD) largest = log + 2 ;
    // might be  that a closer/ point exists
}
//----
*(vecs + counter ) = (best_so_far.x() - c) ; counter++ ;
*(vecs + counter ) = (best_so_far.y() - r) ; counter++ ;

gen_difference_block( best_so_far.x() - c,
       best_so_far.y() - r,
       c,r,ref, mtn,
       di, cols,rows) ;

       }
}
//----------------------------------------


//----------------------------------------

/* Housekeeping  stuff at EOF****/

const int number_of_functions = 33 ;

struct function_data
{
  char *name ;
  void (*function)(const unsigned char *,
   const unsigned char *,
   int *, int *, const int,
   const int , const int ) ;
};

const function_data functions_listed[number_of_functions] =
{
  { "Conjugate_Search" , conjugate_search},/////////1
  { "Cross_Search" , cross_search},
  { "Full_Search" , fsa },
  { "Modified_Conjugate_Search" , modified_conjugate_search},
  { "Modified_SEA" , modified_SEA },/////////////5
  { "Modified_SEA_v2" , modified_SEA_v2 },
  { "Modified_TSS" , modified_TTS },
  { "One_Dimensional_Full_Search" , IDFS },
  { "Parallel_Hierarchical_One_D_Search" , PHODS },
  { "Sampled_Pts" , sampled_pts}, ////////10
  { "Subblock_Full_Search" , subblock_fsa} ,
  { "SEA" , SEA },
  { "TSS" , TTS },
  { "2Dlog_Search" , twoD_log_search },
  { "Cross_Pattern_Search" , CPS },/////////15
  { "Plus_Pattern_Search" , PPS },
  { "Feature_Search" , feature_search},
  { "Modified_Feature_Search" , feature_search_2},
  { "PBIL", Pop_BIL},
  { "Monotony", Feature_Monotony}, /////////20
  { "Modified_Monotony", Feature_Modified_Monotony},
  { "Conjugate_Directional_Search", CDS},
  { "Four_Step_Search", Four_SS},
  { "New_DMD",new_DMD }, //////////24
  { "Block_Based_Gradient_Descent_Search" , BBGDS},//////25
  { "Prefiltered_Sampled_Points" , sampled_pts_prefilt},
  { "Projection_Search", projection_search_2},//////27
  { "Dynamic_SW_Adjustment", DSWA_IS},
  { "Improved_TSS" , Improved_TSS},
  { "Variance" , Feature_variance},///////30
  { "Energy" , Feature_energy},
  { "DCT" , feature_search_DCT},
  { "Tile_search" , tile_search}///////33
};

//-------------------------------------------
```

Bibliography

[1] Dimitris Anastassiou. Digital television. *Preceedings of the IEEE*, 82(4):510 – 519, April 1994.

[2] Oscar C. Au, Yiu-Hung Fok, and Ross D. Murch. Novel fast block motion estimation in feature space. In *ICIP 94 Volume III of III*, volume 3 of *ICIP 94*, page 209 212, 1994.

[3] Shumeet Baluja and Rich Caruana. Removing the genetics from the standard genetic algorithm. Unpublished paper.

[4] John S. Baras, Sohail Zafar, and Ya-Qin Zhang. Predictive block -matching motion estimation for tv coding – part 1: Inter-block prediction. *IEEE Transactions on Broadcasting*, 37(3):97 – 101, September 1991.

[5] Lynne Billard and Raoul LePage, editors. *Exploring the limits of Bootstrap.* John Wiley and Sons, INC, 1992.

[6] Chris Chatfield. *Problem Solving: A Statisticians Guide.* Chapman and Hall, 2 edition, 1995.

[7] Liang-Gee Chen, Wia-Ting Chen, Yeu-Shen Jehng, and Tzi-Dar Chiueh. An efficient parallel motion estimation algorithm for digital image processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 1(4):378 – 385, December 1991.

[8] Liang-Gee Chen, Tzi-Dar Chiueh, and Her-Ming Jong. Accuracy improvement and cost reduction of 3-step search block matching algorithm for video coding. *IEEE Transactions on Circiuts and Systems for Video Technology*, 4(1):88 – 90, February 1994.

[9] Mei-Juan Chen, Liang-Gee Chen, and Tzi-Dar Chiueh. One-dimensional full search motion estimation algorithm for video coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 4(5):504 – 509, October 1994.

[10] Wen-Hsiung Chen, C. Harrison Smith, and S. C. Fralick. A fast computational algorithm for the discrete cosine transform. *IEEE Transactions on Communications*, COM-25(9):1004 – 1009, September 1977.

[11] Keith Hung-Kei Chow and Ming L. Liou. Genetic motion search algorithm for video compression. *IEEE Transactions on Circuits and Systems for Video Technology*, 3(6):440 – 445, December 1993.

[12] Cuthbert Daniel and Fred S. Wood. *Fitting Equations to Data, Computer Analysis of Multifactor Data.* John Wiley and Sons, 1980.

[13] I. R. L. Davies, D. Rose, and R. J. Smith. Automated image quality assessment. In *Proceedings of the SPIE - INternational Society for Optical Engineering.*, volume 1913, pages 27–36, 1993.

[14] Frederic Dufaux and Fabrice Moscheni. Motion estimation techniques for digital tv: A review and new contribution. *Proceedings of the IEEE*, pages 858 – 876, June 1995.

[15] G. Dunn and B. S. Everitt. *Advanced Methods of Data Exploration and Modelling.* Heinemann Educational Books Ltd., 1983.

[16] B. Efron and R. Tibshirani. *An introduction to the Bootstrap.* Chapman Hall, 1993.

[17] Bradley Efron. *The Jackknife, the Bootstrap and other resampling plans.* Society for Industrial and Applied Mathematics., 1982.

[18] Luc De Fos and Michael Stegherr. Parameterizable vsli architectures for the full search block-matching algorithm. *IEEE Transactions on Circuits and Systems*, 36(10):1309 – 1316, October 1989.

[19] M. Ghanbari. The cross-search algorithm for motion estimation. *IEEE Transactions on Communications.*, 38 No 7:950 – 953, July 1990.

[20] Andrew S. Glasnner. *Principle of Digital Image Synthesis , Vol I*, volume 1. Morgan Kuafmann Publishers, Inc, 340 Pine St, Sixth Floor, San Francisco, CA 94104-3205, 1995.

[21] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Addison-Wesley Publishing Company., 1993.

[22] Paul E. Green. *Analysing Multivariate Data.* Dryden Press, 1978.

[23] Caspar Horne, T. Naveen, and Ali Tabatabai. Study of the characteristics of the mpeg2 4:2:2 profile - application of the mpeg2 in studio enviroment. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(3):251 – 272, June 1996.

[24] Fu huei Lin and Russel M. Mercerau. An optimization of mpeg to maximize subjective quality. In *Proceedings ICIP-95.* IEEE Signal Processing Society., IEEE Computer Society Press., 1995.

[25] International Organisatioin for Standardisation. *Generic coding of moving pictures and associated audio.*, h.262 edition.

[26] Bernd Jahne. *Digital Image Processing : Concepts, Algorithms and Scientific Applications.* Springer-Verlag, Berlin Heidelberg, 2 edition, 1993.

[27] Jaswant R. Jain and Ankil K. Jain. Displacement measurement and its application in interframe coding. *IEEE Transactions on Communications*, COM-29(12):1799 - 1807, December 1981.

[28] Alan Jeffrey. *Linear Algebra and Ordinary Differential Equations.* Blackwell Scientific., 1990.

[29] S. Kappagantula and K. R. Rao. Motion compensated interframe image prediction. *IEEE Transactions on Communications*, COM-33(9):1011 - 1015, September 1985.

[30] G. Keesman, A. Cotton, and D. Kessler. Study of the subjective performance of a range of mpeg-2 encoders. In *Proceedings ICIP-95*, volume 2. IEEE Signal Processing Society., IEEE Computer Society Press., 1995.

[31] M. K. Kim and J. K. Kim. Efficient motion estimation algorithm for bidirectional prediction scheme. *Electronics Letters*, 30(8):632 - 633, April 1994.

[32] S. C. Kwatra, Chow-Ming Lin, and W. A. Whyte. An adaptive algorithm for motion compensated color image coding. *IEEE Transactions on Communications*, COM-35(7):747 - 753, July 1987.

[33] Liang-Wei Lee, Jau-Yien Lee, and Jhing-Fa Wang. Dynamic search-window adjustment and interlaced search for block-matching algorithm. *IEEE Transactions on circiuts and systems for video technology*, 3(1):85 - 87, February 1993.

[34] Renxiang Li, Ming L. Liou, and Bing Zeng. A new three-step search algorithm for block motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 4(4):438 - 442, August 1994.

[35] W. Li and E. Salari. Successive elimination algorithm for motion estimation. *IEEE Transactions On Image Processing*, 4(1):105 - 107, January 1995.

[36] Lunrng-Kuo Liu and Ephraim Feig. A block-based gradient descent search algorithm for block motion estimation in video coding. *IEEE Transactions on Circuits and Systems for Video Technology.*, 6(4):419 - 422, August 1996.

[37] J. T. McClave and R. L. Schaeffer. *Probability and Statistics for Engineers.* Intenational Thomson Publishers, 1995.

[38] William S. Meisel. *Computer-Orianted Approaches to Pattern Recognition.*, volume 83 of *Mathematics in Science and Engineering.* Academic Press, New York, San Francisco, London, 1972.

[39] D. S. Moore and G. P. McCabe. *Introduction to the Practice of Statistics.* W.H Freeman and Company, New York, 2nd edition, 1993.

[40] F. Mosteller and J. W. Tukey. *Data Analysis and Regression.* Addison-Wesley Publishing Company., 1977.

[41] H. G. Musmann, P. Pirsh, and H. J. Grallert. Advances in picture coding. *Proceedings of the IEEE*, 73(4):523 - 531, April 1985.

[42] A. N. Netravali and J. D. Robbins. Motion-compensated television coding:part1. *The BELL System technical Journal*, 58(3):631 - 669, March 1979.

[43] Bjorn Olstad. Adaptive temporal decimation for video compression algorithms. *Journal of Electronic Imaging*, 2(1):5 - 18, January 1993.

[44] Stephen Orr. Digital video spearheads tv and video conferencing applications. *Computer Design*, pages 59 - 70, December 1994.

[45] Lai-Man Po and Wing-Chung Ma. A novel four-step search algorithm for fast block motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology.*, 6(3):313 - 317, June 1996.

[46] Elisa Sayrol, Antoni Gasull, and Javier R. Fonollosa. Motion estimation using higher order statistics. *IEEE Transactions on Image Processing.*, 5(6):1077 - 1084, June 1996.

[47] B. E. Shi, T. Roska, and L. O. Chua. Design of linear cellular neural networks for motion sensitive filtering. *IEEE Transactions of Circuits and Systems-II:Analogue and Digital Signal Processing*, 40(5):320 - 331, May 1993.

[48] Ajit Singh. *Optic Flow Computation: A Unified Perspective.* IEEE Computer Society Press Monograph, 1991.

[49] Ranjan Kumar Som. *A Manual of Sampling Techniques.* Heinemann, 1973.

[50] Ram Srinivasan and K. R. Rao. Predictive coding based on efficient motion estimation. *IEEE Transactions on Communications*, COM-33(8):888 - 896, August 1985.

[51] F. G. Stremler. *Introduction to communications systems.* Addison-Wesley, 1990.

[52] Earl W. Swokowski. *Calculus with Analytic Geometry.*, volume Second Alternate Edition. PWS-KENT, 1988.

[53] S.D. Voran and S. Wolf. The development and evaluation of an objective video quality assesmsnt system that emulates human visual panels. In *IBC 1992 International Broadcasting Convention*, volume 358, pages 504 – 508, 1992.

[54] Mark Allen Weis. *Data Structures and Algorithm Analysis in C++.* Benjamin Cummings, 1994.

[55] G. Barrie Wetherhill. *Intermediate Statistical Methods.* Chapman and Hall, 1981.

[56] *Proceedings ICIP-94*, 1994.

[57] Kun-Min Yang, Ming-Ting Sun, and Lancelot Wu. A family of vsli designs for the motion compensation block-matching algorithm a. *IEEE Transactions on Circuits and Systems*, 36(10):1317 – 1325, October 1989.

[58] Sohail Zafar and Ya-Qin Zhang. Predictive block matching motion estimation for tv coding – part2: Inter-frame prediction. *IEEE Transactions on Broadcasting*, 37(3):102 – 105, September 1991.