Dynamic Programming Based Reverse Frame Selection
for VBR Video Delivery under Constrained Resources

Dayong Tao\textsuperscript{a}, Jianfei Cai\textsuperscript{a}, Haoran Yi\textsuperscript{a}, Deepu Rajan\textsuperscript{a}, Liang-Tien Chia\textsuperscript{a}, King Ngi Ngan\textsuperscript{b}
\textsuperscript{a}School of Computer Engineering, Nanyang Technological University, Singapore 639798
\textsuperscript{b}Dept. of Electronic Engineering, Chinese University of Hong Kong, Hong Kong

Submitted to IEEE Trans. on Circuits and Systems for Video Technology
as Transactions Paper

Contact Author:
Jianfei Cai
Center for Multimedia and Network Technology
School of Computer Engineering
Nanyang Technological University
BLK N4, #02a-32 Nanyang Avenue
Singapore 639798
Phone: (65) 6790-6150
Fax: (65) 6792-6559
Email: asjfcai@ntu.edu.sg

\textsuperscript{*}This research is partially supported by Singapore A*STAR SERC Grant (032 101 0006).
Abstract

In this paper, we investigate optimal frame selection algorithms based on dynamic programming for delivering stored VBR video under both bandwidth and buffer size constraints. Our objective is to find a feasible set of frames that can maximize the video’s accumulated motion values without violating any constraint. It is well known that dynamic programming has high complexity. In this research, we propose to eliminate non-optimal intermediate frame states, which can effectively reduce the complexity of dynamic programming. Moreover, we propose a reverse frame selection (RFS) algorithm, where the selection starts from the last frame and ends at the first frame. Compared with the conventional dynamic programming based forward frame selection, the RFS is able to find all the optimal results for different preloads in one round. We further extend the RFS scheme to solve the problem of frame selection for VBR channels. In particular, we first perform the RFS algorithm offline, and the complexity is modest and scalable with the aids of frame stuffing and non-optimal state elimination. During online streaming, we only need to retrieve the optimal frame selection path from the pre-generated offline results, and it can be applied to any VBR channels as long as the VBR channels can be modelled as piecewise CBR channels. Experimental results show good performance of our proposed algorithms.

Keywords: Optimal frame selection, dynamic programming, VBR video delivery, VBR channels, bandwidth smoothing, motion awareness.
1 Introduction

A variable-bit-rate (VBR) encoded video generally offers improved picture qualities over the corresponding constant-bit-rate (CBR) encoded video give the same average bitrate [1, 2]. However, the VBR video traffic is more difficult to manage because of its significant bit-rate burstiness over multiple time scales [3, 4, 5]. In particular, the high peak and bursty bit rate can substantially increase bandwidth requirement for the continuous playback at the client site. To address this problem, various bandwidth smoothing techniques [6, 7, 8, 9, 10] have been proposed. The basic idea of bandwidth smoothing is to prefetch data ahead of each burst so that large frames can be transmitted earlier at a slower rate. Most existing smoothing techniques focus on either minimizing the bandwidth requirements at a given buffer size [11, 12] or minimizing the buffer requirements under rate-constrained bandwidth conditions [13]. While bandwidth limits the amount of data that can be transmitted per unit time, buffer size regulates the amount of data that can be prefetched [14]. If both bandwidth and buffer size are limited, lossy smoothing is unavoidable.

Given the maximum bandwidth and fixed buffer size, Ng and Song [15] suggested to introduce playback pauses or delete B frames (and subsequently P frames) when the transmission exceeds the rate limit. Their algorithms drop frames without content awareness and have no global optimization criteria. In [14], Zhang et al. proposed an optimal selective frame discard algorithm to minimize the number of frames that must be discarded in order to meet the bandwidth and buffer size limits. However, their algorithm does not take into account semantic frame importance and only considers motion JPEG videos. In [16], Zhou and Liou proposed a nonlinear frame sampling strategy for video streaming under bandwidth and buffer constraints. Their objective is to obtain an optimal set of frames that can maximize the video’s salient scores. Dynamic programming is used to find the optimal path. Nevertheless, the authors did not consider the inter-frame dependency, and they focused on videos with constant frame sizes.

In addition to the individual problems pointed out above, most of the existing lossy smoothing algorithms assume CBR channels during the smoothing process. However, in reality, the network bandwidth such as Internet bandwidth is usually time-varying. In [17], Feng and Liu proposed two methods for streaming stored video over VBR channels (both methods pre-compute a bandwidth smoothing plan assuming fixed buffer size and constant bandwidth): 1) adapt the video stream on-the-fly; 2) run the smoothing algorithm online under the new bandwidth condition for the rest of the frames. However, real-time computation of the transmission plan is too complicated for timely delivery and the situation becomes even worse when there are many concurrent client connections. In [18], Gan et al. proposed a more robust dual-plan bandwidth smoothing method for layer-encoded video streaming. Upon bandwidth renegotiation failure, the scheme
adaptively discards the enhancement layer data to maintain the original frame rate.

Another problem of most existing lossy smoothing algorithms is that they usually do not consider the packet loss problem caused by network congestion or physical-layer bit corruptions. Recently, we have seen extensive studies on rate-distortion optimized (RDO) video streaming over lossy channels [19, 20, 21]. The most representative work is the one in [19], where Chou et al. proposed a framework for streaming packetized media over a lossy packet network in a RDO way. The proposed framework is able to minimize the end-to-end distortion under a rate constraint by choosing the right packets to transmit at a given transmission opportunity. Although the framework is very comprehensive and theoretically sound, it requires an accurate network delay model, which is very difficult to obtain for a network such as Internet. In addition, the proposed optimal packet scheduling in [19] is very complex, which might limit its implementation in practice.

In this paper, we assume the packet loss problem can be well handled by the error control techniques deployed in the transportation layer and the link layer. We only focus on optimal lossy smoothing based on the a priori motion information in video. By lossy smoothing, we mean not all the frames can be selected due to resource constraints. Since high motion objects are usually more perceptible to human eyes, it is desired to select more frames in the high motion segments for better perception. Our goal is to select a set of frames out of the video that can deliver the maximal accumulated motion metrics while being guaranteed transmittable and playable under bandwidth and buffer constraints.

In particular, we first analyze the problem of delivering stored video over CBR channels. In addition to the frame stuffing approach [16], we propose to eliminate non-optimal intermediate frame states in forward frame selection to reduce the computation complexity of dynamic programming. Then, we find that the problem can also be solved by a reverse frame selection (RFS) scheme, where the selection starts from the last frame and ends at the first frame. The major advantage of our proposed RFS is that by running RFS just once, we can easily retrieve any optimal frame selection path starting from any frame at any buffer state. We further extend the RFS scheme to solve the problem of streaming stored video over the VBR channels that can be regarded as piecewise CBR channels. We only need to run RFS \( k \) times, where \( k \) is the number of channel bandwidth samples, and it can apply to any pattern of the VBR channels with bandwidth changes occurring at any time.

The rest of the paper is organized as follows. Section 2 states the problem setting and introduces the related work. Section 3 reviews our previous work for computing the amount of motion in each frame. We describe our proposed forward frame selection and reverse frame selection algorithms for CBR channels in
Section 4 and Section 5. Section 6 describes how to apply the RFS scheme for the VBR channels. In Section 7, we evaluate the performance of our proposed algorithms under both CBR and VBR channels. Finally, Section 8 concludes this paper.

2 Background

2.1 Problem Statement

Our optimal transmission plan is computed based on the system setting shown in Figure 1. Two separate buffers are used at the client side for smoothing and decoding purposes, respectively. The decoding buffer retrieves compressed frames from the receiving buffer and sends the decoded pictures to video sink for display. We assume that once a frame is retrieved from the receiving buffer, its space is immediately made available for future incoming data. In other words, we only need to examine the receiving buffer fullness to avoid buffer overflow and underflow when we compute the transmission plan. In the following, without specification, buffer size means the receiving buffer size.

In addition, for practical video coding, there usually exists inter-frame dependencies in the coded video. For example, most MPEG videos consist of I, P, and B frames. While I frames are intra-coded and can be decoded independently, forward predicted P and bidirectionally predicted B frames need their references for proper decoding. Thus, the encoding order is different from the display order. In this research, we select frames according to the encoding order. In other words, for the receiving buffer, frames are removed one by one in their encoding order at fixed intervals. It is the decoding buffer's responsibility to hold necessary references.

![Diagram of the system setting](image)

Figure 1: The system setting for computing the optimal frame selection path.

The transmission plan consists of the optimal frame selection path and the schedule for frame delivery. The schedule tells the server the time and the period to stop transmitting data. In particular, for a long sequence of small-size frames being transmitted, the client consumes less data than the amount of data being received, which might cause buffer overflow eventually. Under such a circumstance, the server will
have to either stay idle for some time or transmit at a reduced rate to prevent client buffer overflow.

After describing the system setup, now we formulate the problem. For a video sequence with $N$ frames, let $B$ denote the size of the client buffer and $f_i$ denote the frame size for the $i$-th frame, where $i = 1, 2, \ldots, N$. The problem of motion-based optimal frame selection can be expressed as

$$\max_{s_i} M = \sum_{i=1}^{N} m_i \cdot s_i,$$

subject to the bandwidth constraint

$$r_i \leq \frac{\text{bandwidth}}{\text{framerate}},$$

and the buffer constraint, for $k = 1, 2, \ldots, N,$

$$\sum_{i=1}^{k} f_i \cdot s_i - \text{preload} \leq \sum_{i=1}^{k} r_i \leq B - \text{preload} + \sum_{i=1}^{k-1} f_i \cdot s_i,$$

where $m_i$ is the motion metric gain for selecting the $i$-th frame, $s_i$ is the indicator function, which is equal to 1 if the $i$-th frame is selected and equal to 0 otherwise, and $r_i$ is the amount of data sent within the time slot between the $(i - 1)$-th frame and $i$-th frame. Note that in this paper we consider both CBR and VBR channels. In the cases of VBR channels, bandwidth is a variable. However, we assume the bandwidth does not change rapidly and remains constant in a magnitude of a few seconds. This is reasonable for slow-fading channels in mobile networks or using TFRC [22] in Internet, where the application rate is adjusted according to a certain feedback interval. In other words, the VBR channels we consider in this paper can be regarded as piecewise CBR channels so that dynamic programming can be applied.

### 2.2 Related Work on Frame Selection

In this section, we describe three existing frame selection algorithms: the just-in-time algorithm [14], the greedy algorithm [14] and the Z-B diagram algorithm with frame stuffing [16], which will be used for comparison with our proposed algorithms. Since the original algorithms only consider intra-frame video coding, we make some changes in order for them to be applicable with inter-frame dependencies.

#### 2.2.1 Just-In-Time Algorithm

The just-in-time (JIT) algorithm [14] is probably the most intuitive approach for frame selection. It always drops the current frame if client buffer underflow occurs and reduces the transmission rate when buffer overflow occurs. Consequently, the JIT algorithm has no content awareness. Due to inter-frame dependency,
we orderly apply the JIT algorithm to different types of frames: I, P and B, and make sure the references are selected before we select a new frame. The computational complexity of the layered JIT algorithm is $O(N^2)$.

### 2.2.2 Greedy Algorithm

The greedy algorithm proposed in [14] is always trying to make the result look the best at the moment. It selects frames according to their reward metrics, i.e. the one currently having the largest reward metric will get selected first. To overcome inter-frame dependency, we use weighted metric for frame sorting, which is given by

$$m_i^w = \begin{cases} m_i, & \text{if frame } i \text{ is B frame} \\ \sum_{k=i}^j m_k, & \text{if frame } i \text{ is I or P frame} \end{cases}$$

where $j$ is the index for the last frame in the current GOP. In this way, we ensure the reference frames are always given greater weighted metric and thus being considered first. However, we still need to check whether a frame’s references have been selected or not before inserting the frame to the path. The overall computational complexity of the greedy algorithm is also $O(N^2)$. In addition, in this paper we introduce another metric, $m_i^r = m_i / f_i$, instead of using the original motion metric $m_i$ for frame selection since the greedy algorithm is very sensitive to frame size.

### 2.2.3 Z-B Diagram

Figure 2 shows the Z-B diagram proposed in [16]. In particular, a discrete-time model is used at frame level for client buffer management. Each discrete-time point along the horizontal direction is identified by the particular frame fetched out at that moment for decoding, and each buffer fullness level at any time point is called a state indicated by an arrow endpoint in Figure 2. As shown in the figure, all the enqueue lines (slanted lines) are vertically separated by a fixed distance called the step size and every state at each frame is on an enqueue line. This configuration effectively bounds the number of states at each frame by $B / \text{stepsize}$. The larger the step size, the fewer the number of states and hence the less the computational work. To realize the configuration, every frame’s size must be a multiple of the step size. In the case of VBR video, the authors [16] suggested to use the greatest common divisor (GCD) of all the frame sizes as the step size. However, in practice, the GCD is most likely to be a very small value, which results in large number of states at each frame time point and thus significantly increases the computational complexity. Hence, frame stuffing has to be used to increase the step size at the cost of sacrificing bandwidth. For example,
suppose \( \text{stepsize} = 100 \), a frame size of 1009 will be stuffed with 91 dummy data to make its size 1100. The average stuffing for a video of \( N \) frames is \( N \times \frac{(\text{stepsize}-1)}{2} \).

\[
\begin{array}{cccccc}
I_1 & P_1 & B_1 & B_2 & P_2 \\
0.00 & 0.08 & 0.04 & 0.07 & 0.11 \\
\end{array}
\]

Figure 2: The Z-B diagram with frame stuffing.

3 Motion Information Representation

As stated in Section 2.1, a fundamental problem we need to solve is how to represent the amount of motion for each video frame. The common approach is to analyze the motion field and use the motion energy to quantify the amount of motion such as in [23]. In this paper, we apply our previous work, “Pixel Change Map” (PCM) [24, 25], to compute the amount of motion. Compared with the approaches directly based on motion fields, the PCM scheme is of low complexity and very easy to implement. In fact, any frame or content classification scheme can be used in our proposed frame selection algorithms. The PCM scheme by no means is the only or the best way to measure the content importance.

3.1 Pixel Change Map

The perception of motion content for human visual system relies on the intensity of the motion. By intensity, we mean how fast a certain object moves. The faster the object moves, the more perceptible it is to human. As we have observed that a higher intensity of motion would lead to a large number of pixel changes in the video frames, the pixel change map of the frame gives a good characterization of the motion content in the video. This can be also justified from the famous optical flow constraint equation [26]

\[
-d = [p_x, p_y] \cdot [v_x, v_y]^T,
\]
which shows that the velocity of a pixel, \([v_x, v_y]\), in a video signal \([p_x, p_y]\) is directly related to the temporal derivative \(d\), which is approximated by the difference of adjacent frames.

Based on the above observation, we use the simple \textit{Pixel Change Maps} as the measurement of the amount of motion in video signals. In particular, for the current frame \(i\), we compute the frame absolute differences \(d_i(x, y)\), where \(d_i(x, y) = |p_i(x, y) - p_{i-1}(x, y)|\). For each pixel in this frame, if the absolute difference \(d_i\) is greater than a fixed threshold of 10, the corresponding location in the \textit{PCM} is set to 1. The comparison of \(d_i\) with the threshold is simply to undo the effect of any noise associated with the camera or the discretization process when dealing with digital camera. The reason we choose the threshold of 10 is that this threshold has been found to be quite robust to noise in our earlier work [25].

After getting the \textit{PCMs}, we form a 1-D pixel change sequence, where the \(i\)-th component denoted as \(\tilde{d}_i\) is the average pixel change in the \(i\)-th PCM. Then, we filter this pixel change sequence to obtain a more accurate measurement of motion since the human’s perception of motion content in the video has the “smoothening” effect and the human eyes tend to smooth the motion of the video [27]. Besides, the PCMs also contain pixel changes due to other factors in addition to motion such as sudden change of lighting condition. Those pixel changes are regarded as noise that corrupts the true measurement of the amount of motion. To get the accurate measurement of the video motion content according to humans’ perception, we use filters to remove the noise from the pixel change maps. Details on the filter design can be found in [24, 25].

### 3.2 Motion Curve

In this paper, we simply define the motion metric gain \(m_i\) in Eqn. (1) as \(m_i = \tilde{d}_i\), where \(\tilde{d}_i\) is the filtered average pixel change values. Using the proposed PCM scheme, we extract the “motion curves” for four common interchange format (CIF) video sequences: (1) Akiyo, (2) Foreman, (3) Mobile, and (4) Stefan. Figure 3 shows the motion curves for each video separately. We evaluate the effectiveness and accuracy of the “motion curves” by watching the video evolving with the “motion curves”. We find that the extracted motion curves correspond to the motion content in the videos very well. In particular, for the ‘Akiyo’ video sequence, the video contains very little motion and thus the motion curve (Fig. 3(a)) is very close to zero. The ‘Forman’ video sequence contains various amount of motion at different time, especially when there is a large camera panning motion from frame 175 to frame 235, which is represented as a plateau in the motion curve (Fig. 3(b)). For the ‘Mobile’ video sequence, which contains very smooth object motion and camera motion, the extracted motion curve (Fig. 3(c)) is relatively flat. For the ‘Stefan’ video sequence, the motion
content in the video is very high and there is a periodical motion due the rhythm of playing tennis. As shown in Fig. 3(d), we can see that the extracted motion curve manifests this periodical rhythm very well. At the end of the video sequence, the player rushes towards the net and we see a up drift of the motion curve there, which indicates the increasing amount of motion.

Figure 3: The amount of motion per frame before and after the filtering for (a) Akiyo, (b) Foreman, (c) Mobile and (d) Stefan.

4 Forward Frame Selection for CBR channels

After obtaining the motion metrics, in this section, we discuss how to maximize the reward function shown in Eqn. (1) by selecting a feasible set of frames, which satisfies the fixed network bandwidth and the buffer constraints. Since each video frame is either selected or discarded, this problem can be considered as a 0-1 knapsack problem [28], which can be solved by dynamic programming. The Viterbi algorithm is a dynamic programming algorithm often used for solving optimization problems whose solutions depend on
their subproblems [28]. It avoids overlap computation by solving each subproblem once and saves the answer to a table for future usage. At the final stage, it performs back tracing to find the optimal path that reaches the optimal solution.

Without using the Viterbi algorithm, theoretically the number of possible path is $2^N$, which means the computational complexity grows exponentially with the number of frames. By using the Viterbi algorithm, the complexity can be greatly reduced to $O(BN)$, where $B$ represents the client buffer size. In this section, we introduce our proposed dynamic programming based forward frame selection algorithm, which consists of three basic components: non-optimal state elimination, virtual states and optimal preload. The component of non-optimal state elimination is for reducing the complexity of dynamic programming. The complexity can be further decreased by combining with the frame stuffing approach mentioned in [16]. The component of virtual state is to deal with the issue of inter-frame dependency, and the component of optimal preload is to find the lowest preload value for the global optimal result.

### 4.1 Viterbi Trellis

Similar to the Z-B diagram algorithm [16], we also use the common discrete-time model at frame level for client buffer management, as shown in Fig. 4. Let $b_j^i$ denote the buffer fullness level at the $j$-th state at frame $i$. If state $l$ at frame $q$ is created by state $k$ at frame $p$, or in other words state $l$ at frame $q$ is directly lined with state $k$ at a previous frame $p$, then we have the following relation:

$$b_q^l = \min\{b_p^k + (q - p) \times (\text{bandwidth/ framerate}) - f_q, B - f_q\}, \quad (6)$$

where $\text{bandwidth/ framerate}$ is the amount of data that can be transmitted in one frame time-slot period, $B$ is buffer size, and $f_q$ is the size of frame $q$. Note that $b_q^l$ becomes a full buffer state ($b_q^l = B - f_q$) if buffer overflow occurs during state transition from $b_p^k$ to $b_q^l$. In this case, the server has to stay idle for some time or transmit at a reduced rate in order to avoid client buffer overflow. In addition, there is a preload level at the initial stage just before playback starts. It is the amount of data that has been prefetched into the client buffer. The time required to buffer preload is called startup delay:

$$\text{startup delay} = \frac{\text{preload}}{\text{bandwidth}}. \quad (7)$$

Depending on the tolerable startup delay, preload can vary in the range of $[0, B]$.

Let $M_j^i$ denote the accumulated motion metrics at $b_j^i$. For state transition from $b_p^k$ to $b_q^l$, $M_q^l = M_p^k + m_q$, where $m_q$ is the motion metric associated with frame $q$. If another state $b_n^s$ also leads to state $b_q^l$, we resolve
the “collision” with:

$$M_q^j = \max\{M_q^l, M_s^l + m_q\}.$$  

Clearly, we always try to maximize the accumulated motion metrics at each state. In case of a “tie”, when

$$M_q^l = M_s^l + m_q,$$

we can arbitrarily choose one path without affecting global optimality. Alternatively, we may want to choose the one that selects more frames as an additional metric. In summary, a state can be completely characterized by a five-tuple vector \((f_i, m_i, b_j^i, M_j^i, p_j^i)\), where \(p_j^i\) is a pointer pointing back to the state that creates the current one and is used for back tracing the path at the final stage.

### 4.2 Non-Optimal States

Theoretically, the number of possible states increase in an exponential manner with the increase of frame number, which makes the dynamic programming algorithm computationally prohibitive. However, in this research, we find the number of states at each frame can be largely reduced by three factors:

1. buffer size because no state can fall outside the given buffer range,

2. inter-frame dependencies because P and B frames need their reference frames for proper decoding,

3. non-optimal states described below.

**Lemma 1** For any two optimal states \(b_j^i\) and \(b_k^i\) at frame \(i\) (an optimal state means a state that could be included in the final optimal path), if \(b_k^i < b_j^i\), then \(M_k^i \geq M_j^i\), and vice versa. In other words, for optimal states at frame \(i\), \(M_j^i\) increases monotonically as \(b_j^i\) decreases.
Proof: Suppose \( b^k_i < b^l_i \) and \( M^k_i < M^l_i \), and \( b^k_i \) is on the final optimal frame selection path \( OP_k \) (see Figure 5). Obviously, \( b^l_i \) and \( b^k_i \) are mutually exclusive because frame \( i \) can only be selected once. Because \( b^l_i \) is at a higher buffer state, the least \( b^l_i \) can do is to select the same frames that \( b^k_i \) has selected from frame \( i \) until the end. The accumulated motion values of the new path \( OP_j \) is larger than that of \( OP_k \) by \( M^l_i - M^k_i \). In fact, \( b^l_i \) has a better chance to select additional frames such as path \( OP_{j+1} \). Whatever the case is, \( b^l_i \) can make the accumulated motion values larger by at least \( M^l_i - M^k_i \), which contradicts the claim that \( OP_k \) is the optimal path. Hence, \( b^k_i \) can never be an optimal state. For example, the state \( b^k_2 \) at \( P_5 \) with \( M^k_2 = 0.23 \) in Figure 4 is a non-optimal state.

![Figure 5: An illustration of a non-optimal state](image)

The elimination of non-optimal states can dramatically reduce the computation complexity because it not only reduces the number of states at each individual frame but also prevents those non-optimal states from propagating into subsequent frames.

### 4.3 Virtual States

Referring to Fig. 4, we need to consider \( P_2, B_3, \) and \( B_4 \) in order to obtain all possible states at \( P_5 \). This works fine for a small set of frames within a GOP. However, it is not suitable for I frames in a long video sequence with numerous GOPs since any state transition to a I frame from any previous frame is allowed. The number of possible state transitions to examine for the I frame in the next GOP can be potentially huge especially when the I frame is near to the end of the video sequence. In order to avoid this inconvenient multi-step “look back”, we introduce the concept of “virtual state”. A virtual state at a frame \( i \) is a state where a frame selection path passes through the frame \( i \) time point without selecting frame \( i \), shown as empty endpoints in Fig. 6. With virtual states, we only need to look back the states at frame \( i \) to get all possible states for frame \( i + 1 \).
In particular, a state \( j \) at frame \( i \) is carried forward and becomes the \( j \)-th virtual state at frame \( i + 1 \) with the following relations:

\[
\begin{align*}
    b_{i+1}^j &= \min\{b_i^j + \text{bandwidth/framerate}, B\} \\
    M_{i+1}^j &= M_i^j \\
    p_{i+1}^j &= \begin{cases} 
        p_i^j, & \text{if } b_i^j \text{ is a virtual state;} \\
        \text{pointer to } b_i^j, & \text{if } b_i^j \text{ is an actual state.}
    \end{cases}
\end{align*}
\]

From the virtual states, we can easily find the corresponding actual states by:

\[
\begin{align*}
    b_{i+1}^a &= b_{i+1}^v - f_{i+1}, \\
    M_{i+1}^a &= M_{i+1}^v + m_{i+1}, \\
    p_{i+1}^a &= p_{i+1}^v,
\end{align*}
\]

while taking into account inter-frame dependencies. For example, in Fig. 6, the first virtual state at \( B_3 \) points back to the actual state at \( I_1 \), and hence it can not create an actual state at \( B_3 \) because \( P_2 \) is not selected. However, this virtual state cannot be discarded because it might create an optimal state at future I frames. After obtaining all the virtual and actual states at frame \( i + 1 \), they are jointly verified for state optimality. In other words, states at frame \( i + 1 \), virtual or actual, must all satisfy \textsc{Lemma} 1. For instance, in Figure 6, the non-optimal virtual state \( b_4^{3v} \) is removed from \( B_4 \) since \( (M_4^{3v} = 0.12) < (M_4^{1a} = 0.15) \). Note that eliminating \( b_4^{3v} \) also prevents it from propagating to \( P_5 \).

Special actions need to be taken when we apply \textsc{Lemma} 1 for I frames. Considering the following scenario:

\[
\ldots P_{i-3} B_{i-2} B_{i-1} I_i B_{i+1} B_{i+2} P_{i+3} \ldots
\]
we divide all the I-frame states into two sets: set 1 contains the states pointing back to an actual state before $P_{i-3}$, and set 2 contains the states pointing back to $P_{i-3}$, $B_{i-2}$, or $B_{i-1}$. Obviously, only the actual states in set 2 are allowed to create actual states in $B_{i+1}$ and $B_{i+2}$. If we directly apply LEMMA 1 to all the states in both set 1 and set 2, it is possible that an actual state from set 2 is eliminated by a state from set 1. However, the discarded state might create a potentially optimal state at $B_{i+1}$ or $B_{i+2}$. Therefore, for I frames, we treat the two sets separately and apply LEMMA 1 within each set.

4.4 Optimal Preload

In the previous subsections, we have shown how to obtain the optimal result given bandwidth, buffer size and preload. In this subsection, we study the case where the preload is not fixed. It is clear that for different preload values the optimal results will be different. Depending on the client’s tolerable startup delay, the preload can vary in the range of $[0, B]$. Intuitively, a larger preload should yield a larger or at least equal optimal result. The question is: given bandwidth and buffer constraints, what is the minimum preload required to obtain the maximal global optimal result? In other words, there exists a certain preload level, above which we can not get a better optimal result.

Obviously, $\text{preload} = B$ is able to give the maximal global optimal result. But the problem is how to bring the preload down to the optimal level. In this research, we propose a two-step approach to bring down the preload level from $B$. In particular, in the first step we perform global reduction (GR). GR is defined as the distance between the lowest state on the maximal optimal path and the empty buffer line (Fig. 7). It is clear that we can bring the entire maximal optimal path down by GR without changing the global optimal result. In the second step, we perform local reduction (LR). The idea of LR comes from the observation that in the cases of buffer overflow we have to waste some channel bandwidth. In fact, through reducing the preload level, we can avoid wasting channel bandwidth or reduce the amount of bandwidth wasted. Consider two adjacent states $b_i^j$ and $b_k^l$ on the maximal optimal path, where $b_k^l$ is a full buffer state. If buffer overflow occurs during the state transition from $b_i^j$ to $b_k^l$, the amount of LR at frame $k$ is defined as:

$$LR_k = \min\{b_i^j + (k - i) \cdot \left(\frac{\text{bandwidth}}{\text{framerate}}\right) - B, b_s^l\},$$

where $b_s^l$ is the lowest state on the maximal optimal path up to frame $k$. It is clear that we can bring down the optimal path from the first frame to frame $k$ by $LR_k$ without affecting the global optimal result. Note that this local reduction has no effect on the optimal path after frame $k$. Therefore, by jointly applying GR and applying LR at the places of buffer overflow, we are able to bring down the preload to the optimal level.
5 Reverse Frame Selection for CBR Channels

In this section, we propose a reverse frame selection (RFS) scheme, which selects frames starting from the last frame of a video sequence until its beginning, to solve the problem of video streaming over CBR channels.

As shown in Fig. 8(a), the symbols above the full buffer line tell the frame type (I, P, or B) as well as its frame number in the sequence. The real numbers below the empty buffer line are the motion metrics associated with each frame. It is obvious that after the client consumes the last frame, the buffer should become empty. Hence, the first state at the last frame is positioned at the empty buffer line, which we refer to as an empty buffer state. A state is represented by the starting point of an arrow, and all arrows are pointing upward because we can record the accumulated motion metrics only if the arrow end is within the full buffer line or its “buffer resource need” can be satisfied. An upward arrow actually means consumption of the buffer data. In contrast, reception of data during each frame slot is reflected by the downward slanted lines between frames. In case of “buffer underflow”, such as from $B_N$ to $P_{N-1}$, an empty buffer state will be created. It means that the amount of data transmitted during the period is more than enough to reach the current state, and the server needs to stay idle for some time or reduce the transmission rate. Because an path can terminate at any frame, if there is no empty buffer state at a frame, we will create one such as that at $B_{N-2}$.

Fig. 8(a) is not intuitive to interpret. We use a simple “buffer mirroring” technique to make the computation easier and more intuitive. Imagine there is a mirror aligned with the empty buffer line in Fig. 8(a). If we look from the full buffer line side, we shall see a mirrored buffer model as shown in Fig. 8(b). The buffer mirroring effect makes the computation just like what we do in the forward frame selection scheme.
except that the selection is from the end of the video sequence to the beginning. All the concepts, including Lemma 1, discussed for forward frame selection can also be applied to the mirrored buffer model.

Note that at the first frame, RFS generates multiple accumulated motion metric gains at different states and each gain corresponds to the optimal result that we can get at that preload level. In other words, RFS runs only once and gets all the optimal results for different start-up delays, which is very useful for the case of multiple clients with different start-up delay requirements. For those preloads that are not exactly matched in the RFS results, we use the results of their nearby lower matched preloads. On the contrary, the forward frame selection scheme has to run many times in order to obtain all the optimal results for multiple start-up delay requirements, which is very time-consuming.

6 Reverse Frame Selection for VBR Channels

For the VBR channels (piecewise constant channels), the optimal frame selection becomes extremely difficult since we do not know when and how the channel is going to change in the future. i.e., the channel variation is unpredictable. Suppose we know the range of bandwidth variation, one possible approach is to offline compute the optimal frame selection path according to the middle value of the bandwidth variation range. Note that this middle value is not the average bandwidth, which we do not know. We can also compute the optimal path according to the minimum (or maximal) bandwidth, but it will lead to high channel bandwidth wasting (or high occurrence of buffer underflow). During transmission, the just-in-time (JIT) algorithm [14] is applied for on-the-fly adaptation in response to bandwidth changes. If the current bandwidth is larger, JIT raises buffer occupancy, which reduces the probability of future buffer underflow.
In case of buffer underflow, JIT drops the current frame right the way. When overflow occurs, JIT reduces the transmission rate. Clearly, all these approaches use the pre-computed frame selection path and the actual path cannot be better than the pre-computed one. Another possible approach is to use JIT directly without any pre-computed frame selection path, which tries to send all the frames without content awareness.

In this paper, we propose to use the RFS scheme for video streaming over VBR channels. In particular, we sample the bandwidth variation range into a finite sequence of channel rates. For a given client buffer size, we run the RFS scheme for each sampled channel rate. During transmission, if the starting buffer occupancy status is $b_1$ and the current bandwidth is $C$, we will first classify it into one of the pre-selected channel rates $C_1$ and then retrieve the optimal frame path for the channel rate $C_1$ with a starting state of $b_1$. If at frame $N_2$ the buffer state is $b_2$ and the bandwidth is changed to $C_2$, we will retrieve the optimal frame path starting at frame $N_2$ for the channel rate $C_2$ with a starting state of $b_2$. Fig. 9 shows such an example. In this way, the global optimality is approximately preserved under dynamic changing network conditions. The key advantage here is that we only need to run RFS $k$ times, where $k$ is the number of channel bandwidth samples, and it can apply to any pattern of piecewise constant channels occurring at any time as long as the changes are within the variation range.

![Figure 9: Optimal path switching for video streaming over VBR channels.](image)

### 7 Experimental Results

#### 7.1 Experimental Results of Short Videos

We conduct experiments to compare six algorithms: “OFS”, “OFS+OP”, “Z-B”, “JIT”, “Greedy” and “Greedy+WR”, where “OFS” stands for our proposed forward optimal frame selection algorithm without optimal preload, “OFS+OP” stands for our proposed forward optimal frame selection with optimal preload,
“Z-B” represents the Z-B diagram algorithm with frame stuffing, “Greedy” represents the greedy algorithm with weighted motion metric, and “Greedy+WR” represents the greedy algorithm with weighted ratio metric $m_i$. Note that we have also obtained the results of the Reverse Viterbi algorithm. Since they are the same as “OFS” and “OFS+OP” for the respective cases, we do not list them out for the conciseness of this paper.

We choose four representative MPEG-4 CIF video traces to evaluate the performance of the various algorithms. Each trace contains 300 frames with a frame rate of 30 frames/s. They are both encoded in the pattern: $\ldots I B B P B B P \ldots$ with a GOP size of 90. The akiyo video trace contains little motion information and is coded at a very low bit rate. The foreman video trace containing moderate motion is coded at a relatively lower bit rate. The mobile video trace containing nearly flat high motion and complex texture is coded at a high bit rate. The stefan video trace containing nearly flat high motion and complex texture is coded at the highest average bit rate (see Table 1). The table also shows the setting of the buffer sizes and the bandwidth ranges for each video sequence. Note that except for the “OFS+OP”, all the other algorithms use fixed preloads, i.e. half of the buffer sizes.

<table>
<thead>
<tr>
<th>Video Title</th>
<th>akiyo</th>
<th>foreman</th>
<th>mobile</th>
<th>stefan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitrate (kbps)</td>
<td>47.52</td>
<td>288.25</td>
<td>745.10</td>
<td>977.00</td>
</tr>
<tr>
<td>Motion Metric</td>
<td>3.05</td>
<td>63.2</td>
<td>109.28</td>
<td>127.0</td>
</tr>
<tr>
<td>Buffer size (kbytes)</td>
<td>9.6</td>
<td>45</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>Bandwidth Range (kbps)</td>
<td>[15, 52.5]</td>
<td>[120, 300]</td>
<td>[420, 780]</td>
<td>[330, 1050]</td>
</tr>
</tbody>
</table>

Figures 10 shows the frame selection results using the six different algorithms under different channel bandwidth. It can be seen that the “OFS+OP” outperforms all the other algorithms especially at low bandwidth regions since it fully explores the buffer capacity. Under the fixed preload levels, our proposed “OFS” algorithm always gives the optimal results. For the “Z-B” algorithm, we choose step size = 100 bytes for all the video traces. The “Z-B” performance of stefan is only slightly worse than “OFS” while the gap is very large for akiyo. This is because stefan has a very high bit rate and the stuffed data only occupies a small percentage of the total bandwidth whereas for akiyo the stuffed data severely degrades bandwidth utilization. As expected, the selection results of the “Greedy+WR” algorithm are very close to the optimal results except at low bandwidth regions. This is because at low bandwidth some P frames that have small motion metrics still tend to be selected due to the large weights assigned to them. As a result, the “Greedy+WR” performs
as poor as the “Greedy” algorithm and the “JIT” algorithm at low bandwidth regions. Note that for the “JIT” algorithm I and P frames are always considered first, which is equivalent to assign them “weights”. It is surprising that the JIT that has no content awareness performs better than the “Greedy” algorithm for stefan. The reason is perhaps that for stefan high motion frames have very large frame sizes and the “Greedy” algorithm completely ignores the cost of consuming large frames.

Figure 10: The selected motion metric gains under different algorithms for (a) Akiyo, (b) Foreman, (c) Mobile, and (d) Stefan.

Figure 11 shows the number of states at each frame for stefan using the “OFS” algorithm with and without the non-optimal state elimination, and the “Reverse Viterbi” algorithm. The bandwidth is 600 kbps and the buffer size is 150 kbytes. For “OFS”, the preload is set to 75 kbytes. As shown in the figure, the initial linear increment indicates an exponential growth of states with the increase of the frame number (note that the log-scale is used at the vertical axis). The curves then become relatively constant because of the buffer size constraint. Those discontinuous points are mainly due to inter-frame dependencies. Comparing the cases with and without the non-optimal state elimination, we can see that there exist a huge number of
non-optimal states. By applying the proposed non-optimal state elimination, we reduce the number of states at each frame by approximately 100 times. In addition, we can observe that the number of states for the “Reverse Viterbi” algorithm is more or less the same as that for the “OFS”.

![Graph showing the number of states at each frame.](image)

Figure 11: The number of states at each frame.

### 7.2 Experimental Results of Long Video

The purpose of the previous experiments is to prove the concepts of our proposed algorithms, where the short test video sequences and the small client buffer are being used. However, in practical applications such as VoD, the video is typically much longer and the buffer size even in today’s mobile devices can be much bigger. Therefore, in this section, we study the performance of our proposed algorithms in the cases of long video and large client buffer size.

We create a longer video sequence, where we equally choose 15 times of each of the four video traces in Table 1 and randomly shuffle these sixty 300-frame traces. The generated mixed sequence is encoded into MPEG-4 bitstream with an average bitrate of 542.33 kbps, a pattern of IPBBPBBP..., a GOP size of 60, and an accumulated motion values of 4598.21. The buffer size we consider is in the range from 256 kbyte to 2048 kbyte. For such a long video sequence and large buffer size, with only non-optimal state elimination, the computational complexity is still too high. Thus, in addition to non-optimal state elimination, frame stuffing is used to further reduce complexity at the cost of sacrificing some bandwidth resources. Note that, in the following, we use our proposed reverse frame selection scheme for experiments due to its low complexity and flexibilities, and hereafter “OFS” stands for the proposed reverse frame selection scheme.
7.2.1 Impact of Preload and Frame Stuffing

Figure 12 shows the optimal accumulated motion values that we can achieve under different frame stuffing sizes and preloads with a fixed bandwidth of 300 kbps. It is obvious that the smaller the stuffing size we use, the better performance we achieve since less bandwidth is wasted. Comparing Figure 12(a) and (b), we find that large preloads for the smaller buffer do not lead to as proportionate gains in the accumulated motion metrics as those for larger buffer. In addition, in the case of 512 kbytes buffer and the frame stuffing size of 1 byte (i.e. no stuffing), the accumulated motion results stop at the preload time of a little over 4 seconds, which is less than half of the largest allowable preload time, 13.65 seconds \((8 \times 512/300)\). On the contrary, the corresponding results in the case of 1024 kbytes buffer stop around 18 seconds. The stop point is actually the point of the optimal preload, after which increasing preload will not change the performance. The reason to have a shorter optimal preload for 512 kbytes buffer is that a smaller buffer is more likely to incur buffer overflow at an early stage, and once buffer overflow occurs, increasing the preload becomes useless (see Section 4.4).

![Figure 12: The results of the accumulated motion metrics for delivering the long VBR video under different frame stuffing sizes and preloads with a buffer size of (a) 512 kbytes and (b) 1024 kbytes.](image)

7.2.2 Effectiveness of Complexity Reduction

In this section, we evaluate the effectiveness of frame stuffing as well as non-optimal state elimination for reducing the complexity of dynamic programming. Figure 13 shows the average number of states per frame for different frame stuffing step sizes under different buffer conditions. As we can see, the number of states per frame reduces with increasing stuffing sizes. For instance, for buffer = 1024 kbytes, the number of states reduces from over 128k \((2^{17})\) at stuffing = 1 byte (i.e. no stuffing) down to a little over 4k \((2^{12})\) at stuffing
= 200 bytes, a dramatic reduction of 32 times. However, as the number of states becomes less, further reduction by increasing the stuffing size appears to be less significant.

Figure 13: Frame stuffing vs average number of states per frame with bandwidth = 300 kbps.

The effectiveness of non-optimal state elimination can also be evaluated from Figure 13. In particular, let $2^T$ represent the theoretical number of states per frame after taking the contribution of frame stuffing into account. For example, for buffer = 512 kbytes and stuffing = 200 bytes, the theoretical value is $2^T = (512k/200) = 2^{11.36}$. Let $2^R$ represent the recorded average number of states in Figure 13. It is clear that the percentage calculated by $(2^T - 2^R)/2^T$ indicates the contribution from non-optimal state elimination. Table 2 and Table 3 show this percentage of state reduction due to non-optimal state elimination at different buffer sizes. It can be seen that with no frame stuffing the reduction percentage can be as high as 95.79% for buffer = 512 kbytes. However, as we increase the stuffing size, the reduction becomes less effective. This is not surprising because frame states are spaced out by at least a distance equal to the stuffing size. As the stuffing size increases, it becomes less likely to create non-optimal states. Comparing Table 2 and Table 3, we can conclude that a larger buffer creates a smaller percentage of non-optimal states, and non-optimal state elimination is more effective for small stuffing sizes and small buffers.

Note that although the number of states after frame stuffing and non-optimal state elimination is still large, we find that at each frame many consecutive states point to the same frame to be selected next. Therefore, in our implementation, we group those states leading to the same next destination together and only store the ranges of different groups. In this way, the resulted storage overhead is actually not much.
Table 2: Effectiveness of non-optimal state elimination at buffer = 512 kbytes.

<table>
<thead>
<tr>
<th>Number of States per Frame (log2)</th>
<th>Frame Stuffing Step Size (byte)</th>
<th>1</th>
<th>2</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theory</td>
<td></td>
<td>19</td>
<td>18</td>
<td>13.36</td>
<td>12.36</td>
<td>11.36</td>
<td>10.36</td>
<td>9.36</td>
</tr>
<tr>
<td>Record</td>
<td></td>
<td>14.43</td>
<td>14.34</td>
<td>12.51</td>
<td>11.81</td>
<td>10.98</td>
<td>10.09</td>
<td>9.25</td>
</tr>
<tr>
<td>Reduction (%)</td>
<td></td>
<td>95.79</td>
<td>92.09</td>
<td>44.52</td>
<td>31.70</td>
<td>23.16</td>
<td>17.07</td>
<td>7.34</td>
</tr>
</tbody>
</table>

Table 3: Effectiveness of non-optimal state elimination at buffer = 1024 kbytes.

<table>
<thead>
<tr>
<th>Number of States per Frame (log2)</th>
<th>Frame Stuffing Step Size (byte)</th>
<th>1</th>
<th>2</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theory</td>
<td></td>
<td>20</td>
<td>19</td>
<td>14.36</td>
<td>13.36</td>
<td>12.36</td>
<td>11.36</td>
<td>10.36</td>
</tr>
<tr>
<td>Record</td>
<td></td>
<td>17.25</td>
<td>16.88</td>
<td>14.03</td>
<td>13.11</td>
<td>12.17</td>
<td>11.22</td>
<td>10.30</td>
</tr>
<tr>
<td>Reduction (%)</td>
<td></td>
<td>85.13</td>
<td>77.00</td>
<td>20.45</td>
<td>15.91</td>
<td>12.34</td>
<td>9.25</td>
<td>4.07</td>
</tr>
</tbody>
</table>

7.2.3 Performance Comparison

Figure 14(a) shows the results of the accumulated motion metrics of different frame selection algorithms under different bandwidth conditions, where “OFS+n” refers to our proposed optimal reverse frame selection algorithm with \( n \) bytes of frame stuffing. The observations are similar to those described in Section 7.1.

Figure 14(b) shows the results under different buffers. It can be seen that all the OFS+n algorithms outperform the other three algorithms under all the buffer conditions. We can also observe that a larger buffer such as 2048 kbytes does not yield significant gain. This is due to the bandwidth constraint. In addition, it is interesting to see that the Greedy algorithm has a worse performance when buffer increases from 1024 kbytes to 2048 kbytes. The reason is perhaps that a larger buffer allows the Greedy algorithm to select more large-size frames at the beginning, which consumes most of the bandwidth and thus compromises the overall gain. Note that we did not compare with the Z-B diagram algorithm since its accumulated motion results are the same as our proposed OFS at the same stuffing size.
Figure 14: The results of the accumulated motion metrics of different frame selection algorithms with a fixed preload of 375 kbytes (300kbps × 10 seconds). (a) Under different bandwidth conditions with buffer = 1024 kbytes (b) Under different buffers with bandwidth = 300 kbps.

7.2.4 Performance of VBR Channels

We use piecewise-CBR channels to approximate the bandwidth variations of VBR channels. Particularly, we divide the time into consecutive $T$-second intervals and at the beginning of each interval the bandwidth is randomly chosen from the set: \{100, 200, 300, 400, 500\} kbps. The time interval $T$ are set to 2 and 10 seconds, representing a fast-changing VBR channel and a slow-changing channel, respectively.

Figure 15 shows the frame selection results of different algorithms over the VBR channels, where “Ave” is the algorithm using the middle bandwidth to compute the optimal path (see Section 6), and “UPB” is the upper bound that achieves the global optimization by assuming the channel bandwidth variation is known a priori. It can be seen that both AVE+200 and OFS+200 outperform JIT significantly. For the case of the fast-changing VBR channel in Figure 15(a), AVE+200 has a better performance than OFS+200. This is because OFS is optimal on the condition that the new bandwidth will remain constant until the end of the sequence. With the bandwidth varies so frequently, the global optimality of the OFS is severely deviated. On the contrary, for the case of the slow-changing VBR channel in Figure 15(b), OFS+200 outperforms AVE+200. This is because with less frequent bandwidth changes, the OFS can better preserve the global optimality over longer CBR channel segments while the AVE+200 algorithm is severely degraded by the long-term low bandwidth CBR channel segments.
8 Conclusion

In this paper, we have studied the problem of optimal frame selection for streaming stored video over both CBR and VBR channels using dynamic programming. Our major contributions are threefold. First, we have proposed the elimination of non-optimal states, and combining with the frame stuffing it can effectively reduce the computational complexity of dynamic programming, especially in the cases of small stuffing sizes and small buffers. Second, we have proposed the RFS algorithm, which can find the optimal results for any preload in one round for CBR channels. Third, our proposed RFS algorithm has been smartly extended for the VBR channels, which can be modelled as piecewise CBR channels. Experimental results have demonstrated that with modest complexity our proposed algorithm achieves much better performance than the common JIT algorithm, especially in the cases of slow-changing VBR channels.

References


