Efficient Algorithms for Optimal Video Transmission

Dexter Kozen Yaron Minsky Brian Smith

Computer Science Department
Upson Hall
Cornell University
Ithaca, New York 14853-7501
{kozen,yminsky,bsmith}@cs.cornell.edu

Abstract

This paper addresses the problem of sending an MPEG-encoded video stream over a channel of limited bandwidth. When there is insufficient bandwidth available for the rate at which the sequence was encoded, some data must be dropped. In this paper we give fast algorithms to determine a prioritization of the data that optimizes the visual quality of the received video sequence in the sense that the *maximum gap of unplayable frames is minimized*.

Our results are obtained in a new model of encoded video data that is applicable to MPEG and other encoding technologies. The model identifies a certain key relationship between the play order and dependence order of frames that allows fast determination of optimal send orders by dynamic programming.

1 Introduction

The transmission requirements for video data and ordinary data are quite different. Ordinary data need not be received in real time, but every bit must be correct. Video data, on the other hand, can tolerate some loss, but must be received, decoded, and played back in real time. Unfortunately, limitations on channel bandwidth may not allow all the data to get through for real-time playback. The question then arises: which data should be sent so as to optimize the perceived quality of the received video stream?

For example, consider a video segment consisting of 120 frames, all the same size and independently decodable. Suppose further that due to bandwidth constraints only 90 frames will get through in the allotted time. The question is: which frames should be sent? Sending the frames in the order of play would leave a 30-frame gap at the end, which at a playback rate of 30 frames/sec would be perceived by the viewer

as a one-second discontinuity. A better choice would be to drop every fourth frame, because the gaps are small and evenly distributed.

Determining which frames to drop is easy if all frames are the same size and there are no dependencies, as with the example above. However, the problem is significantly more complicated without these restrictions.

Such is the case with the MPEG encoding standard [4]. Dependencies between frames arise because frames that are very similar to neighboring frames are not represented in their entirety, but encoded succinctly as modifications to neighboring frames. This gives considerable savings in storage and transmission time, but has the disadvantage that to decode a frame, the receiving process must also have access to all the frames it depends on. Another complicating factor is that the sizes of the encoded frames vary significantly due to dependencies and compression, in practice from 1K to 12K bytes.

When transmitting MPEG data over a traditional network, experiments show that available bandwidth is highly variable, depending on network characteristics such as link capacities and network load. This has lead many researchers to develop *prioritized* transmission schemes. For example, Priority Encoded Transmission (PET) [1] uses a variation on forward error correction where redundant data is introduced to compensate for data loss. Priority is encoded by associating high redundancy with more important frames. Cyclic-UDP [7] uses a retransmission scheme that allows for prioritized delivery. Reservation networks such as Tenet [3] allow users to reserve bandwidth in the network, with statistical guarantees on the transmission properties of the packets. Prioritized delivery can be obtained using several channels. For example, a 1.5 Mbit/sec channel could be divided into a 1.0 Mbit/sec channel that has no packet loss, and a 0.5 Mbit/sec channel that has 20% packet loss. One would then place high priority data on the first channel and low priority data on the second.

An open problem in all of these systems is to choose a prioritization order for MPEG video data sensibly. It is this problem we address in this paper.

Our solution assumes a transport layer that implements some priority-based transmission scheme that when provided with n packets in priority order will transmit the k highest priority packets. For our probabilistic results, we model k as a random variable distributed in the set $\{0, 1, \ldots, m\}$ according to some probability distribution $\mathbf{Pr}$, where m is the total number of packets. For instance, $\mathbf{Pr}\,(k = 0)$ is the probability that no data will be received, while $\mathbf{Pr}\,(k = m)$ is the probability that all packets get through.

1.1 Results

In this paper, we present fast algorithms to determine optimal transmission orders. They are optimal in the sense that they *minimize the maximum gap* for a given bandwidth, where a *gap* is an interval of unplayable frames. A frame is *unplayable* if either it was not received, or it was received but cannot be decoded because it depends on a frame that was not received.

We give two algorithms for two different situations. The first algorithm is deterministic and assumes that we know the channel bandwidth exactly. This would be

the situation for example with a reservation-based ATM network, where the network server provides a guaranteed bandwidth to the sending process for an interval of time. The algorithm produces a table that can be calculated *offline in advance* and stored with the video data. The table gives for each $k \leq m$ a set of frames of total weight at most k that minimizes the maximum gap. The complexity of the algorithm to create the table is $O(mn^3)$. When the video stream is to be transmitted, the frames to transmit are determined by table lookup.

The second algorithm does not assume known bandwidth, but assumes that the bandwidth is uniformly distributed in an interval with known endpoints. This assumption is realistic, since there exist techniques to measure the channel bandwidth at any instant of time reliably [7]. The algorithm computes an optimal order to transmit the frames, where p is the size of the interval. The order is optimal in the sense that it minimizes the *expected* maximum gap. The complexity of the algorithm is $O(pmn^3)$.

We remark that the obvious greedy algorithm (in each step, choose the enabled frame that minimizes the maximum gap of the remaining set of frames; continue until the cutoff k is exceeded) is not optimal in either case.

1.2 Comparison with Recent Work

The problem we solve in this paper is part of an approach to the general problem called *temporal decimation*, in which selected frames are dropped from the encoded sequence of frames. The missing frames are filled in by replaying the latest decoded frame or by interpolation. As more frames are dropped, the image degrades by developing a jerkiness, depending on the sizes of the gaps between playable frames and the quality of the interpolation process. While temporal decimation is standard practice for some kinds of compressed video, it has generally not been used with MPEG streams because of the frame dependencies. Interpolation methods for MPEG are currently under investigation [6], which could be used in conjunction with our algorithms.

A more common approach, popular with the signal processing community, is *spatial decimation*. This takes the form of dropping the least significant coefficients of the Fourier or cosine transform of the image [2]. As more data are dropped, the image degrades by developing a fuzziness around the edges of objects, called the *corona effect*. This approach is highly dependent on the coding method and data format, whereas the temporal approach is relatively independent of the particular coding method. We would expect that a combination of the two approaches would produce the best results in practice.

The remainder of this extended abstract contains a more detailed technical development. Proofs are omitted from this abstract for lack of space. Full proofs and further details can be found in [5]. A sample run is given in the Appendix.

2 A Formal Model of Video Sequences

We model an MPEG-encoded video sequence by:

- a finite set U of *frames*,
- a total order $\sqsubseteq$ on U called the *play order*, intuitively the chronological order of the frames,
- a partial order $\preceq$ on U called the *dependence order*, intuitively the order giving the dependence of one frame on another as described in the introduction. We write $u \preceq v$ to indicate that frame v depends on u, *i.e.* cannot be decoded without knowledge of u; and
- a weight function $\omega : U \to \mathbb{N}$ giving the number of data units (nominally "packets") comprising each frame.

We also postulate the following key condition describing the relationship between the two orders $\sqsubseteq$ and $\preceq$:

Condition 2.1 *For any frame u, the set of frames depending on u form an $\sqsubseteq$-interval. In other words,*

$$\forall u\ \exists x\ \exists z\ \{v \mid u \preceq v\} = \{y \mid x \sqsubseteq y \sqsubseteq z\}\ .$$

This condition is satisfied by all MPEG sequences.

Although in reality if $u \preceq v$ then u is typically larger than v, we do not assume any formal relationship between $\preceq$ and ω in the model; nor do we take advantage of any special structure of MPEG other than Condition 2.1.

Let $\leq$ be any partial order on U. We write $u < v$ if $u \leq v$ but not $v \leq u$. For $u \in U$ and $A \subseteq U$, define

$$\begin{aligned} \uparrow_{\leq}(u) &= \{v \mid u \leq v\} \\ \uparrow_{\leq}(A) &= \bigcup_{u \in A} \uparrow_{\leq}(u) \quad = \quad \{v \mid \exists u \in A\ u \leq v\}\ . \end{aligned}$$

$\downarrow_{\leq}(u)$ and $\downarrow_{\leq}(A)$ are defined similarly. For $V \subseteq U$, a $\leq$-*suffix* of V is a subset $X \subseteq V$ such that $X = V \cap \uparrow_{\leq}(X)$, *i.e.* it is closed upward under $\leq$. A $\leq$-*prefix* of V is a subset $X \subseteq V$ such that $X = V \cap \downarrow_{\leq}(X)$, *i.e.* it is closed downward under $\leq$. An $\leq$-*interval* of V is an intersection of a suffix of V and a prefix of V.

In this notation, Condition 2.1 says that for all frames u, $\uparrow_{\preceq}(u)$ is an $\sqsubseteq$-interval.

For any set A, denote the cardinality of A by $|A|$. For $A \subseteq U$, define $\omega(A) = \sum_{u \in A} \omega(u)$. A *send order* is a map $\pi : \{1, 2, \ldots, \omega(U)\} \to U$ such that for any u, $|\pi^{-1}(u)| = \omega(u)$. Intuitively, at time i, we send the next packet from frame $\pi(i)$.

For a send order π, $u \in U$, $A \subseteq U$, and $0 \leq i \leq \omega(U)$, define

$$\begin{aligned} \mathrm{last}_{\pi}(u) &\stackrel{\mathrm{def}}{=} \max \pi^{-1}(u) && (1) \\ \mathrm{last}_{\pi}(A) &\stackrel{\mathrm{def}}{=} \max_{u \in A} \mathrm{last}_{\pi}(u) && (2) \\ \mathrm{received}_{\pi}(i) &\stackrel{\mathrm{def}}{=} \{u \mid \mathrm{last}_{\pi}(u) \leq i\} && (3) \\ \mathrm{playable}_{\pi}(i) &\stackrel{\mathrm{def}}{=} \{u \mid \downarrow_{\preceq}(u) \subseteq \mathrm{received}_{\pi}(i)\} \end{aligned}$$

$$
\begin{aligned}
&= \{u \mid \mathrm{last}_\pi(\downarrow_\preceq(u)) \le i\} && (4)\\
\mathrm{unplayable}_\pi(i) &\stackrel{\mathrm{def}}{=} U - \mathrm{playable}_\pi(i) && (5)\\
\mathrm{maxint}(A) &\stackrel{\mathrm{def}}{=} \max\{|I| \mid I \text{ is an } \sqsubseteq\text{-interval of } A\} && (6)\\
M_\pi(i) &\stackrel{\mathrm{def}}{=} \mathrm{maxint}(\mathrm{unplayable}_\pi(i)) && (7)\\
\mathcal{E}(M_\pi(k)) &\stackrel{\mathrm{def}}{=} \sum_{i=0}^{\omega(U)} M_\pi(i)\mathbf{Pr}\left(k = i\right) . && (8)
\end{aligned}
$$

The number $\mathrm{last}_\pi(u)$ represents the time that the last packet of u is sent under the send order π, and $\mathrm{last}_\pi(A)$ is the time that the last packet of some frame in A is sent under the send order π. When π is the send order and i is the cutoff, the set $\mathrm{received}_\pi(i)$ is the set of frames received; the set $\mathrm{playable}_\pi(i)$ is the set of playable frames (the set of frames u such that u and all frames on which u depends are received); the set $\mathrm{unplayable}_\pi(i)$ is the set of unplayable frames; and the number $M_\pi(i)$ is the length of the longest unplayable interval. The quantity $\mathcal{E}(M_\pi(k))$ is the expected length of the longest unplayable interval when π is the send order. A send order π is *optimal* if $\mathcal{E}(M_\pi(k))$ is minimum.

The send order π defines a total order on frames according to the time at which their last packets are sent:

$$u \trianglelefteq_\pi v \iff^{\mathrm{def}} \mathrm{last}_\pi(u) \le \mathrm{last}_\pi(v) .$$

2.1 Acceptable Send Orders

Define a send order π to be *contiguous* if for all $u \in U$, $\pi^{-1}(u)$ is an $\le$-interval. In other words, all packets of one frame are sent together. Define a send order π to be *consistent with the dependence order* $\preceq$ if $u \preceq v \Rightarrow u \trianglelefteq_\pi v$; that is, if $\trianglelefteq_\pi$ is a total extension of $\preceq$. We say that a send order is *acceptable* if it is both contiguous and consistent with $\preceq$.

Without loss of generality, we can restrict our attention to acceptable send orders:

Lemma 2.2 *For any send order π, there is a contiguous send order π' such that $\mathcal{E}(M_{\pi'}(k)) \le \mathcal{E}(M_\pi(k))$.*

Lemma 2.3 *For any contiguous send order π, there is a contiguous send order π' consistent with $\preceq$ such that $\mathcal{E}(M_{\pi'}(k)) \le \mathcal{E}(M_\pi(k))$.*

Intuitively, a frame is not considered received until its last packet is received; and it does not make sense to send a frame unless all frames on which it depends have already been sent, since it cannot be decoded without them anyway.

Note that an acceptable send order π is uniquely determined by its induced total order $\trianglelefteq_\pi$ on U; for this reason we henceforth omit the π and speak of send orders $\trianglelefteq$.

2.2 Basic Properties

Let $\leq$ be a partial order on U. Two sets $V, W \subseteq U$ are said to be $\leq$-*independent* if no pair of elements $v \in V$ and $w \in W$ are $\leq$-comparable; *i.e.*, if for all $v \in V$ and $w \in W$, neither $v \leq w$ nor $w \leq v$. A family of subsets of U is said to be *pairwise* $\leq$-*independent* if all pairs of sets in the family are $\leq$-independent.

Let $V \subseteq U$. We say that a set A is *enabled in* V if A is a $\preceq$-prefix of V. A set is *enabled* if it is enabled in U. A single element u is *enabled in* V if $\{u\}$ is enabled in V, *i.e.* if u is a $\preceq$-minimal element of V.

The following independence property is the crucial observation that allows optimal send orders to be computed efficiently. Intuitively, if w is an element of an enabled set A, there are no precedence constraints between two frames in $U - A$ on opposite sides of w. Thus no remaining frame occurring before w in the play order sent in the future can cause a frame occurring after w in the play order to become enabled, or vice versa. In other words, the sets of frames occurring before and after w in the play order are $\preceq$-independent. This will allow us to process the $\sqsubseteq$-intervals to the left and right of w independently to get optimal send orders on those subsets, then merge them to get an optimal send order on the union.

Lemma 2.4 *If A is enabled, $w \in A$, $u, v \in U - A$, and $u \sqsubset w \sqsubset v$, then u and v are $\preceq$-incomparable. In other words, if A is enabled and $w \in A$, then $\downarrow_{\sqsubseteq}(w) \cap (U - A)$ and $\uparrow_{\sqsubseteq}(w) \cap (U - A)$ are $\preceq$-independent.*

Corollary 2.5 *The set of gaps of any enabled set are pairwise $\preceq$-independent.*

3 Algorithms for Known Cutoff

Algorithm 3.1 is a recursive algorithm for computing an enabled set A minimizing maxint(A) subject to $\omega(A) \leq k$ for known cutoff k. The order of transmission does not matter. The algorithm actually computes a table of such sets for all values of k by dynamic programming.

The algorithm is called by GAP(I), where I is a gap of some enabled set. It returns a list of subsets $B(d) \subseteq I$ and a list of numbers $C(d)$, one for each $0 \leq d \leq \omega(I)$, such that

(i) $B(d)$ is enabled in I

(ii) $\omega(B(d)) \leq d$

(iii) maxint$(I - B(d))$ is minimum among all subsets of I satisfying (i) and (ii)

(iv) $C(d) =$ maxint$(I - B(d))$.

The algorithm is called at the top level by GAP(U).

Algorithm 3.1 GAP(I):

If $I = \varnothing$, return lists with one element $B(0) = \varnothing$ and $C(0) = 0$.

Otherwise, for each element $u \in \text{enabled}(I)$, set

$$\begin{aligned} I_{\sqsubset u} &= I \cap \{v \mid v \sqsubset u\} \\ I_{\sqsupset u} &= I \cap \{v \mid v \sqsupset u\} \ . \end{aligned}$$

It can be shown that $I_{\sqsubset u}$ and $I_{\sqsupset u}$ are $\preceq$-suffixes of U. Recursively compute GAP($I_{\sqsubset u}$) and GAP($I_{\sqsupset u}$). This gives $B_{\sqsubset u}$, $C_{\sqsubset u}$, $B_{\sqsupset u}$, and $C_{\sqsupset u}$. Create the lists $B_{I,u}$ and $C_{I,u}$ that give the optimal sets to send and their max gap sizes given that u is one of the frames sent, as follows. For each $d < \omega(u)$, set

$$\begin{aligned} B_{I,u}(d) &= \varnothing \\ C_{I,u}(d) &= |I| \ . \end{aligned}$$

For each d such that $\omega(u) \le d \le \omega(I)$, set

$$\begin{aligned} B_{I,u}(d) &= B_{\sqsubset u}(i) \cup \{u\} \cup B_{\sqsupset u}(j) \\ C_{I,u}(d) &= \max\ C_{\sqsubset u}(i),\ C_{\sqsupset u}(j) \end{aligned}$$

where i, j are such that $0 \le i + j + \omega(u) \le d$ and max $C_{\sqsubset u}(i),\ C_{\sqsupset u}(j)$ is minimum. It can be shown that the set $B_{I,u}(d)$ is enabled in I. Using the fact that $C_{\sqsubset u}(i)$ and $C_{\sqsupset u}(i)$ are monotonically nonincreasing in i, one can scan the lists $C_{\sqsubset u}$ and $C_{\sqsupset u}$ linearly to determine $B_{I,u}$ and $C_{I,u}$ in time $O(\omega(I))$.

Now for each d, $0 \le d \le \omega(I)$, set $B_I(d) = B_{I,u}(d)$ and $C_I(d) = C_{I,u}(d)$, where $C_{I,u}(d)$ is minimum over all enabled $u \in I$. This requires linear time in the sizes of the lists.

This algorithm can be implemented in a dynamic programming style by computing GAP(I) for all $\preceq$-upward closed $\sqsubseteq$-intervals I of length 1, then the same for all such I of length 2, and so on. Alternatively, the algorithm can be implemented in a recursive style with intermediate results cached so that subsequent calls with the same parameters take constant time.

There are at most $O(n^2)$ intervals on which the procedure can be called. A call on interval I takes time $O(\omega(I) \cdot |I|)$ exclusive of recursive subcalls, giving a total time bound of $O(\omega(U) \cdot n^3)$.

4 Algorithms for Unknown Cutoff

In this section, we do not assume that the cutoff k is known, but is uniformly distributed on some interval $[a, b]$. We give the algorithm here for $a = 0$ and $b = \omega(U)$, then discuss how to modify it for an arbitrary interval at a cost of an extra linear factor in complexity. In fact, we can give a polynomial-time algorithm for any step function with a constant number of steps, although this is not very interesting because the dependence is exponential in the number of steps of the step function.

Assume then that k is uniformly distributed in the interval $[0, \omega(U)]$. Thus $\mathbf{Pr}\,(k = i) = \frac{1}{\omega(U)+1}$. In this case, (8) simplifies to

$$\mathcal{E}(M_\pi(k)) \;=\; \frac{1}{\omega(U)+1} \sum_{u \in U} \mathrm{maxint}(\uparrow_{\trianglelefteq}(u))\omega(u)\ .$$

Since all we care about is minimization, we can drop the normalization term and just minimize $\theta(U)$ over all choices of send order $\trianglelefteq$, where for any gap I of an enabled set,

$$\theta(I) \;=\; \sum_{u \in I} \mathrm{maxint}(\uparrow_{\trianglelefteq}(u) \cap I)\omega(u)\ .$$

4.1 Greedy Merge

The following result allows us to treat $\preceq$-independent sets separately, then combine the results to give an optimal solution for the union of the two sets.

Let V, W be $\preceq$-independent $\preceq$-suffixes of U such that any $\sqsubseteq$-interval intersecting both V and W also intersects $U - (V \cup W)$. In our application, V and W will be adjacent gaps of some enabled set.

We say that a total order $\leq$ on $V \cup W$ is the *greedy merge* of its restrictions to V and W if for all $u \in V \cup W$,

$$u \in V \iff \mathrm{maxint}(\uparrow_{\leq}(u) \cap V) \geq \mathrm{maxint}(\uparrow_{\leq}(u) \cap W) \tag{9}$$

$$u \in W \iff \mathrm{maxint}(\uparrow_{\leq}(u) \cap V) < \mathrm{maxint}(\uparrow_{\leq}(u) \cap W) \tag{10}$$

(the two conditions are equivalent). The greedy merge of two send orders on V and W can be computed by taking in each step either the least remaining element $v \in V$ or the least remaining element $w \in W$, depending on whether $\mathrm{maxint}(\uparrow_{\trianglelefteq}(v)) \geq \mathrm{maxint}(\uparrow_{\trianglelefteq}(w))$ or $\mathrm{maxint}(\uparrow_{\trianglelefteq}(v)) < \mathrm{maxint}(\uparrow_{\trianglelefteq}(w))$, respectively.

Theorem 4.1 *Any greedy merge of optimal send orders on V and W is an optimal send order on $V \cup W$.*

This implies that the optimal acceptable send order for an $\sqsubseteq$-interval I of U is of the form u followed by a greedy merge of optimal send orders on $I_{\sqsubset u}$ and $I_{\sqsupset u}$ for some u enabled in I. This gives rise to the following recursive algorithm for computing an optimal send order for a given interval I. For each enabled element u of I, separate the interval into $I_{<u}$ and $I_{>u}$ and recursively compute $\theta(I_{\sqsubset u})$ and $\theta(\sqsupset u)$ and optimal send orders on $I_{<u}$ and $I_{>u}$. Set

$$\theta(I) \;=\; \min_{u \in \mathrm{enabled}(I)} |I| \cdot \omega(u) + \theta(I_{\sqsubset u}) + \theta(I_{\sqsupset} u)\ .$$

For the $u \in \mathrm{enabled}(I)$ giving the minimum, greedily merge the optimal send orders on $\theta(I_{\sqsubset u})$ and $\theta(I_{\sqsupset} u)$ and put u in front to get an optimal send order on I.

As in Algorithm 3.1, the algorithm can be implemented either in a bottom-up dynamic programming style or in a recursive style in which intermediate results are

cached. There are $O(n^2)$ possible intervals on which the algorithm can be called, and a call on interval I takes time $O(|I|)$ exclusive of recursive subcalls. This gives a total time bound of $O(n^3)$.

A slight modification of the algorithm works for an arbitrary interval distribution $[a, b]$. We actually compute recursively, for each gap of an enabled set, a table giving the optimal send order for each possible interval distribution $[a', b']$. The optimal send orders on subintervals are greedily merged as above. The merge that gives the best result for each interval distribution $[a, b]$ is retained, and all others are discarded.

References

[1] A. Albanese, J. Blomer, J. Edmonds, M.Luby, and M. Sudah. Priority encoding transmission. *Symposium on Foundations of Computer Science*, pages 604–612, October 1994.

[2] A. Eleftheridas, S. Pejhan, and D. Anastassiou. Algorithms and performance evaluation of the XPhone multimedia communication system. In *Proceedings of ACM Multimedia 93*, pages 311–320, New York, 1993. Association for Computing Machinery Press. (Anaheim, CA, August 1-6, 1993).

[3] D. Ferrari, A. Banerjea, and H. Zhang. Network support for multimedia—a discussion of the Tenet approach. *Computer Networks and ISDN Systems*, 26(10):1267–1280, July 1994.

[4] D. Le Gall. MPEG: a video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.

[5] Dexter C. Kozen, Yaron Minsky, and Brian Smith. Efficient algorithms for optimal video transmission. Technical Report 95-1517, Computer Science Department, Cornell University, May 1995.

[6] Daniel Scharstein. Synthesizing new views from stereo data. In *IEEE Workshop on Representations of Visual Scenes (in conjunction with ICCV'95)*, 1995. Submitted.

[7] Brian Christopher Smith. *Implementation Techniques for Continuous Media Systems and Applications*. PhD thesis, University of California at Berkeley, 1994.

Appendix

Here is a sample run. The program asks for an MPEG sequence consisting of I, P, and B frames. The I frames are independently coded and are usually the largest. Each P frame depends on the previous I or P frame. Each B frame depends on the previous and next I or P frame.

```
sequence? ibbpbbpbbibbpbbbpbbibppbbpbbb
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 position
  I  B  B  P  B  B  P  B  B  I  B  B  P  B  B  B  P  B  B  I  B  P  P  B  B  P  B  B  B type
  0  1  2  1  4  5  4  7  8  7 10 11 10 13 14 15 13 17 18 17 20 20 22 23 24 23 26 27 28 left
  8  1  2  8  4  5  8  7  8 18 10 11 18 13 14 15 18 17 18 28 20 28 28 23 24 28 26 27 28 right
  2  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1  1 weight
total weight = 32
  0: ----------------------------
  1: ----------------------------
  2: ---------I------------------
  3: ---------I--P----------------
  4: ---------I---------I---------
  5: ---------I--P------I---------
  6: ---------I--P------I-P-------
  7: I--------I--P------I---------
  8: I--------I--P------I-P-------
  9: I--P-----I--P------I-P-------
 10: I--P-----I--P------I-PP------
 11: I--P-----I--P---P--I-PP------
 12: I--P-----I--P---P--I-PP--P---
 13: I--P--P--I--P---P--I-PP--P---
 14: I--P--P--I--P---P--I-PP--P-B-
 15: I--P--P--I--P-B-P--I-PP--P-B-
 16: I--P--P--I--P-B-P--I-PPB-P-B-
 17: I--P--P--I--P-B-PB-I-PPB-P-B-
 18: I--P--P--IB-P-B-PB-I-PPB-P-B-
 19: I--P--PB-IB-P-B-PB-I-PPB-P-B-
 20: I--PB-PB-IB-P-B-PB-I-PPB-P-B-
 21: IB-PB-PB-IB-P-B-PB-I-PPB-P-B-
 22: IB-PB-PB-IB-P-B-PB-I-PPB-PBB-
 23: IB-PB-PB-IB-P-B-PB-I-PPB-PBBB
 24: IB-PB-PB-IB-P-B-PB-I-PPBBPBBB
 25: IB-PB-PB-IB-P-B-PB-IBPPBBPBBB
 26: IB-PB-PB-IB-P-B-PBBIBPPBBPBBB
 27: IB-PB-PB-IB-PBB-PBBIBPPBBPBBB
 28: IB-PB-PB-IB-PBBBPBBIBPPBBPBBB
 29: IB-PB-PB-IBBPBBBPBBIBPPBBPBBB
 30: IB-PB-PBBIBBPBBBPBBIBPPBBPBBB
 31: IB-PBBPBBIBBPBBBPBBIBPPBBPBBB
 32: IBBPBBPBBIBBPBBBPBBIBPPBBPBBB
```

The program determines the endpoints of the interval influenced by each frame. For example, the I frame at position 9 influences all frames between position 7 and 18, inclusive. The weights of the frames are arbitrarily assigned 1 (B and P frames) or 2 (I frames) for simplicity for purposes of illustration.

The total weight of all frames in this example is 32. The table that is produced contains a row for each weight. Row i gives a set of frames of total weight at most i which minimize the maximum gap over all sets of weight at most i. For example, if we can only get 14 packets through, we should send the frames indicated in row 14. In this case the maximum gap is 3, and we can do no better.