A 5-Competitive On-line Scheduler for Merging Video Streams

Wun-Tat Chan Tak-Wah Lam Hing-Fung Ting Wai-Ha Wong
Department of Computer Science
University of Hong Kong, Hong Kong
{wtchan,twlam,hfting,whwong}@cs.hku.hk

Abstract

This paper is concerned with an on-line scheduling problem arising from video-on-demand (VOD) systems that support stream merging. Most previous work on this problem focuses on empirical results; Bar-Noy and Ladner [3] are the first to consider worst-case performance and give an on-line algorithm with competitive ratio bounded by $\min(\log_\phi n, \log_\phi \frac{1}{2D}) + O(1)$*, where* $\phi = (1+\sqrt{5})/2$*,* n *is the number of requests, and* $D < 1$ *is the guaranteed startup delay measured as a fraction of the time for a full stream. In this paper we give a new on-line algorithm that improves the competitive ratio to a constant (precisely, 5). Our result implies that the performance does not deteriorate in dealing with a large number of requests and a small startup delay.*

1. Introduction

We study an on-line scheduling problem arising from video-on-demand (VOD) systems that support stream merging. A VOD system often receives many requests for a hot video over a short period of time (say, Friday 7 pm to 9 pm). If the video server uses a dedicated video stream to serve each request, the bandwidth requirement for the server becomes enormous. A simple alternative is to batch [1, 5] the requests so that a single stream is issued to serve requests arrived over a period of time (say, 15 minutes), but the average response time deteriorates much. To reduce the bandwidth requirement without sacrificing the response time, Eager *et al.* [6, 7] proposed a promising technique called stream merging, which takes advantage of the fact that a client can receive data from two streams simultaneously and is equipped with extra buffer. Details are as follows.

Initially, a client receives data from a dedicated stream A for immediate playback. At some time, the client can start to buffer data simultaneously from another stream B initiated earlier (due to another client). Suppose that B is initiated t time units before A. Then after listening to A and B for t time units, the client no longer needs A as it can play back for the next t time units using the data in the buffer, while further buffering from B simultaneously. Conceptually speaking, we say that A *merges* with B when the client starts to count on B only. Note that merging streams allows several clients to be served using only one full stream plus a number of partial streams, thus saving the bandwidth of the server. Figure 1 shows an example illustrating how bandwidth is saved with stream merging.

An on-line scheduler for stream merging is responsible for determining dynamically whether each existing stream should continue to run or merge with another earlier stream. The goal of a scheduler is to minimize the total bandwidth used, i.e., the total duration of all streams. Note that the scheduler determines a schedule without the knowledge about the requests not yet arrived. We analyze the performance of a scheduler using competitive analysis. An on-line scheduler is said to be c-competitive if, for any sequence I of inputs, it uses at most c times of the bandwidth used by the optimal off-line schedule in serving I. In this case, the competitive ratio of the scheduler is said to be at most c.

Previous results: Stream merging was first studied in a primitive form known as patching [4, 8, 9], in which a stream is allowed to merge with a full stream only (the latter will not further merge with other streams). The general stream merging problem was studied by Eager *et al.* [6, 7]; they reported on-line algorithms that look good based on simulation, there is no worst-case performance analysis, though. Bar-Noy and Ladner [3] are the first to investigate the worst-case performance, showing that simple algorithms like best fit and nearest fit are $\Theta(n/\log n)$-competitive, where n is the number of requests. More importantly, they give a non-trivial on-line algorithm with a competitive ratio at most the minimum between $\log_\phi n + O(1)$ and $\log_\phi \frac{1}{2D} + O(1)$, where $\phi = (1+\sqrt{5})/2$ and $D < 1$ is the guaranteed startup delay measured as a fraction of the time for a full stream.

Our contribution: In this paper we give the first on-line algorithm for scheduling stream merging with constant

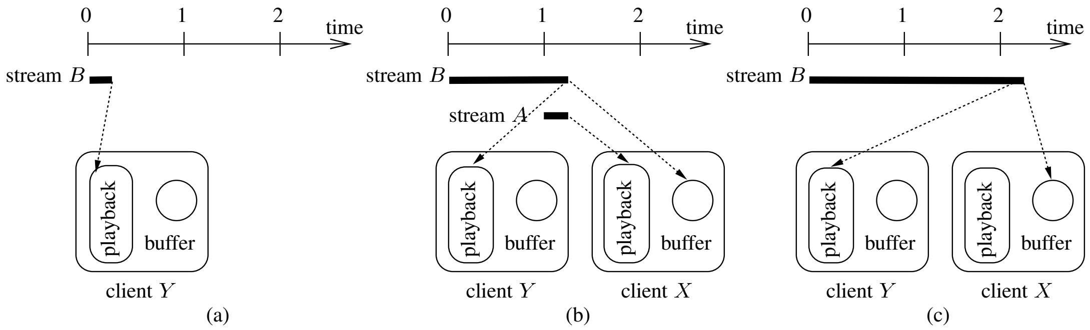


Figure 1. (a) At time 0, client Y is served by a stream B. (b) At time 1, client X arrives. It is served by a new stream A, and it simultaneously buffers data from B. (c) At time 2, A is terminated. Client X plays the content of its buffer and continues buffering from B. Assume the video is of length 120 time units. The total bandwidth for serving both clients is 121 instead of 240.

competitive ratio (precisely, 5), showing the competitive ratio can remain a constant even if there are many requests and the startup delay is arbitrarily small. Our improvement is based on a novel observation that the behavior of any scheduler can be modeled by rectilinear (binary) trees on a triangular grid. This eases both the design and analysis of the algorithm.

Concepts and Notations: We assume that time is divided into intervals of unit length; each unit can be arbitrarily small. Requests arriving within the same interval are treated as a single request as they can all be served by multicasting one stream. Thus, a time unit should be small enough (say, 3 seconds) to ensure a reasonable startup delay. Let L be the length (i.e., in terms of time units) of the video in concern. Following previous work [3], we assume that each client can buffer up to $L/2$ units.

At first glance, the stream merging problem might look simple. In particular, one might expect that every stream initiated should merge with an earlier stream as soon as possible. Figure 2 however shows an example illustrating that it may be more beneficial in some case for a stream to delay merging. Also, merging with the nearest stream may not yield the best schedule.

Organization of the paper: In Section 2 we discuss some preliminary concepts. In Section 3 we describe the novel observation that the behavior of any scheduler can be modeled by rectilinear trees on a triangular grid. In Section 4 we present a general method for constructing a competitive on-line algorithm based on the existence of a rectilinear tree that has good estimates of the optimal rectilinear trees. In Section 5 we show how to construct a rectilinear tree required in Section 4.

2. Preliminaries

2.1. Compact request sequences

A request sequence is said to be *compact* if the last request arrives within $L/2$ time units from the first request. Notice that such a sequence contains at most $L/2 + 1$ requests. Consider any schedule for a compact request sequence I. The first stream initiated must be a full stream. In fact, it suffices to focus on schedules that the first stream is the only full stream and all other streams merge with other streams eventually. It is because Bar-Noy and Ladner [2] have showed that there is an optimal schedule satisfying this property. Since the first stream always contributes a dominating portion in the overall bandwidth usage of any schedule, the overall bandwidth of different schedules may not differ a lot. Thus, we need a more precise comparison. Given any schedule of I, we define the *full cost* to be the bandwidth used by all the streams, and define the *merge cost* to be the bandwidth used by all except the first stream. The full cost and merge cost differ by exactly L and the merge cost is possibly less than L.

Compact request sequences are a key notion for stream merging as Bar-Noy and Ladner [3] observed that a general request sequence can be processed competitively if each consecutive compact request sequence can be processed competitively.

Lemma 1. ([3]) *Let $\mathcal{A}$ be an on-line algorithm such that with respect to any compact request sequence, its merge cost is at most c times of the optimal off-line schedule. Then for any arbitrarily long request sequence, using $\mathcal{A}$ to schedule separately each subsequence spanning an interval of up to $L/2$ units can produce a schedule with an overall bandwidth at most $c + 2$ times of the optimal off-line schedule.*

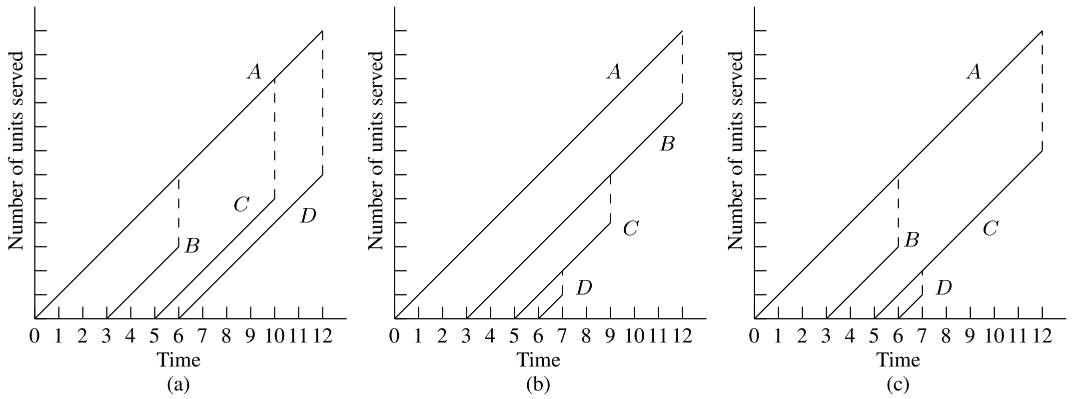


Figure 2. Three schedules for merging four streams A, B, C and D with a total bandwidth of 26, 26 and 23, respectively. They illustrate that sometimes it is more beneficial for a stream to delay merging. (a) B, C and D merge with A in turn. The total bandwidth is 26. (b) B delays the merging with A; C delays the merging with B; and D merges with C, B and then A. The total bandwidth is 26. (c) B merges with A; C delays the merging with A; and D merges with C, which then merges with A. The total bandwidth is 23.

In the rest of this paper, we focus on compact request sequences and give an on-line algorithm whose merge cost is at most three times of that of the optimal off-line algorithm. Then, by Lemma 1, this algorithm is 5-competitive. Unless otherwise specified, all request sequences mentioned in the paper are meant to be compact request sequences. Without loss of generality, we assume that the first request of all compact request sequences arrives at time 0.

2.2. Binary merge trees

In this section we show how to model the schedule produced by any algorithm using binary merge trees. Our binary merge trees attempt to model the occurrences of merging and the status of each stream; this is in contrast to the merge trees used by Bar-Noy and Ladner [3], which model the relationship among the streams. Below, we use the notation $r(s)$ to denote the time when a stream s is initiated.

A binary merge tree is a rooted, ordered binary tree. For an internal node with (out-)degree one, we require the node to have a left son but no right son. Every node u is associated with a stream s_u and a time t_u. In particular, every leaf ℓ corresponds to a distinct stream that is initiated due to a request, and $t_\ell = r(s_\ell)$. For an internal node u, its associated time is always greater than that of its sons, and its stream is always inherited from its left son. If u is of degree two, u and its right son represent the occurrence of a merging as follows.

> Let v be the right son of u. We require that $r(s_v) > r(s_u)$ and $t_u = t_v + r(s_v) - r(s_u)$. At time t_v, the stream s_v (precisely, the clients served by s_v) starts to buffer the content of s_u; after $r(s_v) - r(s_u)$ time units (i.e., at time t_u), s_v merges with s_u and terminates.

No matter u is degree one or two, u and its left son w corresponds to a continuation of the same stream from time t_w to time t_u. That is, the stream has no plan to merge with another earlier stream during this interval. Note that the root as well as all the nodes on the leftmost path of the tree are associated with the same stream, which does not merge with any stream and runs till the end. The time associated with the root is exactly L.

The *cost* of an edge (u, v) is defined as $t_u - t_v$. The *full cost* of a binary merge tree is the sum of the cost over all edges, and the *merge cost* is the full cost minus the cost of the edges on the leftmost path (i.e., L).

Recall that we restrict our attention to compact request sequences and only consider schedules in which all streams merge with the first one. Note that not all such schedules are binary in nature, i.e., some schedule may merge two or more streams with an earlier stream at the same time. Nevertheless, we can easily covert such a schedule into one that can be modeled by a binary merge tree, while preserving the full cost and merge cost. Thus, we have the following lemma. The proof will be given in the full paper.

Lemma 2. *Consider any compact request sequence I. There is a binary merge tree whose full cost and merge cost are equal to those of the optimal off-line schedule.*

Before we leave this section, we show that the optimal binary merge tree (i.e., the tree associated with an optimal off-line schedule) can be assumed to have a more uniform structure.

Lemma 3. *Consider any compact request sequence I. There is an optimal binary merge tree for I such that all the internal nodes, except possibly the root, have both left and right sons.*

Proof. Let T be the optimal binary merge tree with the minimum number of nodes. Assume on the contrary that there is a degree one internal node v. By definition, v has a left son. Let u and w be the parent and the left son of v, respectively.

- Suppose v is the left son of u. We remove v and the two associated edges, and add an edge between u and w. The cost of the resulting binary merge tree is not changed. This contradicts that T has the minimum number of nodes.

- Suppose v is the right son of u. Notice that s_v continues running from time t_w to t_v and starts merging with s_u at t_v. We can modify the merge schedule by making s_v starts merging with s_u earlier, say at t_w. This modification decreases the merge cost of the schedule and contradicts the optimality of T.

Therefore, all internal nodes, except possibly the root, have both left and right sons and the lemma follows. □

3. Rectilinear trees

This section discusses a more powerful tool called *rectilinear trees* to model schedules of stream merging. The new tool is more useful for visualizing the difference between an arbitrary schedule and the optimal off-line schedule. Roughly speaking, rectilinear trees are binary merge tree embedded on a triangular grid. Let $K = L/2+1$. Consider a $K \times K$ grid with K *grid points* (i.e., with $K-1$ *grid arcs*) on each boundary. The grid is distorted into a rectangle in the following sense: The horizontal grid arc has a length of two and the vertical grid arc a length of one.

Notations: A *grid line* is a sequence of consecutive horizontal grid arcs or consecutive vertical grid arcs. A *grid path* is a sequence of grid arcs connecting any two grid points. The rows of the grid are numbered 0 to $K-1$ from top to bottom, and the columns are numbered 0 to $K-1$ from left to right. The *triangular grid* in concern is the upper right triangle of the rectangular grid; precisely, the hypotenuse comprises the grid points $(0,0), (1,1), \cdots, (K-1, K-1)$.

A *rectilinear tree* is a binary tree on the triangular grid satisfying the following properties.

- The root is at the top rightmost corner of the triangular grid and the leaves are on the hypotenuse of the triangular grid.

- Any node and its parent are either on the same row or on the same column. I.e., any edge is either a horizontal or a vertical grid line.

- The grid path from any leaf to the root is *monotone*—As we traverse from the leaf to the root, the rectilinear distance to the root is strictly decreasing.

For any rectilinear tree T, we define the *length* of an edge in T to be the length of the corresponding horizontal or vertical grid line. Furthermore, we define the *full length* of T to be the total length of all its edges, and define the *branch length* to be the total length of all edges except those edges along the leftmost path of the tree. Notice that two rectilinear trees with the same set of grid arcs have the same branch length even if the set of nodes may differ. Branch length is a notion heavily used below; to simplify our discussion, we denote by $|T|$ the branch length of T.

In the next theorem we show that any binary merge tree T can be embedded on a triangular grid as a rectilinear tree T_R in such a way that the cost of each edge in T is exactly the same as the length of the corresponding edge in T_R. Consequently, the branch length of T_R equals the merge cost of T.

Theorem 4. *Any binary merge tree can be embedded on a triangular grid as a rectilinear tree whose branch length equals the merge cost of the binary merge tree.*

Proof. For any binary merge tree T, we embed it into the triangular grid as follows. Without loss of generality, we assume that the first stream is initiated at time 0. A node u in T is associated with a stream s_u and a time t_u as described in Section 2.2. We create a node u_R at the grid point $(r(s_u), (t_u + r(s_u))/2)$. If u is a leaf, u_R is on the hypotenuse at $(r(s_u), r(s_u))$. The root of T becomes a node at the grid point $(0, (L+0)/2) = (0, L/2)$, which is the upper rightmost corner of the grid.

For any internal node u in T, let w be the left son of u. The node w_R is at the grid point $(r(s_w), (t_w + r(s_w))/2)$. Since s_u is the same as s_w and $t_u > t_w$, u_R lies on the right of w_R on the same row. Therefore, the edge (u, w) becomes a horizontal grid line between u_R and w_R. The length of this grid line is $2 \times (t_u + r(s_u) - t_w - r(s_w))/2 = t_u - t_w$, which equals the cost of the edge (u, w) in T.

If u has a right son v in T, the node v_R is at the grid point $(r(s_v), (t_v + r(s_v))/2)$. Since $t_u = t_v + r(s_v) - r(s_u)$, and $r(s_u) < r(s_v)$, u_R lies above v_R on the same column. Therefore, the edge (u, v) is mapped to a vertical grid line between u_R and v_R. The length of this grid line is $r(s_v) - r(s_u) = t_u - t_v$, which equals the cost of the edge (u, v) in T.

Denote T_R as the rectilinear tree defined above. The branch length of T_R equals the merge cost of T. □

4. The on-line algorithm

4.1. Framework

A crucial issue in designing on-line algorithms is to estimate the cost of the optimal solution. We give a simple way to estimate the cost of the optimal solutions of *all* compact request sequences using only one rectilinear tree, called the *complete rectilinear tree*. This rectilinear tree when restricted to any compact request sequence I gives an attainable estimate of the branch length of the optimal rectilinear tree for I. Our on-line algorithm makes reference to this tree to schedule the merging of streams; the resulting schedule corresponds to a rectilinear tree with branch length always upper bounded by the estimated branch length of the optimal rectilinear tree.

Define a *complete rectilinear tree* T to be a rectilinear tree whose leaves comprise all the grid points along the hypotenuse of the triangular grid. For any compact request sequence I, let V_I be the subset of grid points along the hypotenuse corresponding to the requests in I. The rectilinear tree T when restricted to I is called the *partial rectilinear tree* of T with respect to I. It is a rectilinear tree denoted by $T\|_I$ whose set of leaves is V_I and set of grid arcs is a subset of that of T. In other words, for any grid arc e in T, if e is along a path of T from the root to some leaves in V_I, e is in $T\|_I$; otherwise, e is not in $T\|_I$. Note that $T\|_I$ is unique for any given I.

In Section 4.2 we describe a methodology which makes reference of a given complete rectilinear tree T to derive an on-line algorithm $\mathcal{A}_T$. For any compact request sequence I, $\mathcal{A}_T$ gives a schedule whose corresponding rectilinear tree, denoted by $\mathcal{A}_T(I)$, has a branch length at most that of $T\|_I$. That means for any given T and I we can produce a schedule with merge cost at most the branch length of $T\|_I$.

Consider any compact request sequence I. There is an optimal binary merge tree for I satisfying the property stated in Lemma 3. By Theorem 4, this optimal binary merge tree can be embedded on the triangular grid as a rectilinear tree, denoted by OPT_I. In Section 5 we describe a particular complete rectilinear tree S that can provide a very close estimate of the branch length of OPT_I for all compact request sequences I. Precisely, we show that for any I, the branch length of $S\|_I$ is at most three times of that of OPT_I, and hence the merge cost of the schedule produced by $\mathcal{A}_S$ is at most three times of that of the optimal off-line algorithm. Together with Lemma 1, we can conclude the following theorem.

Theorem 5. *$\mathcal{A}_S$ is 5-competitive.*

4.2. The on-line algorithm

In this section we assume T is a chosen complete rectilinear tree. We give an on-line algorithm $\mathcal{A}_T$ which constructs a rectilinear tree based on T and a compact request sequence. The algorithm makes reference to T to determine how the streams serving the requests merge with each other. Details of $\mathcal{A}_T$ are as follows.

When the first request arrives (at time 0), the scheduler initiates a stream for this request and this stream will not merge with any stream. The initial rectilinear tree contains only two nodes, namely, the root at the top rightmost corner and a leaf at the top leftmost corner, of the triangular grid. These two nodes are connected by a horizontal grid line. All the grid points along this grid line are associated with the first stream.

When a subsequent request arrives, it is assigned a dedicated stream X immediately. We also determine when X should begin buffering data from which earlier stream. Let (r, r) denote the leaf corresponding to current request where r is the arrival time. We locate in T the least common ancestor a of (r, r) and the leaf corresponding to the request just before the current request. A new grid path is added to the current rectilinear tree as follows.

- If a lies on column r, the new grid path is a vertical grid line running upwards from (r, r). The path stops at the first grid point it intersects with the current rectilinear tree. In this case, X immediately buffers data from the stream associated with the intersecting grid point.
- If a does not lie on column r, it must lie on the right of column r because of the monotone property of the paths in T. Suppose a lies on column c_a. The new grid path comprises of a horizontal grid line from (r, r) to (r, c_a) and a vertical grid line from (r, c_a) to the first grid point it intersects with the current rectilinear tree. In this case, $2(c_a - r)$ time after X is initiated, X starts buffering from the stream associated with the intersecting grid point.

All the grid points along the new path, except the intersecting point, are associated with the stream X. The above decision is made every time when a request arrives.

For any compact request sequence I, recall that $|\mathcal{A}_T(I)|$ and $|T\|_I|$ denote the branch length of $\mathcal{A}_T(I)$ and $T\|_I$, respectively. The next lemma bounds $|\mathcal{A}_T(I)|$ by $|T\|_I|$.

Lemma 6. *For any complete rectilinear tree T and compact request sequence I, we have $|\mathcal{A}_T(I)| \leq |T\|_I|$.*

Proof. Suppose I has k requests. For $1 \leq i \leq k$, let q_i denote the arrival time of the i-th request and consider the following definitions.

- T^i is the partial rectilinear tree of T with respect to the first i requests in I;
- p_T^i is the grid path in T^i from leaf (q_i, q_i) to the least common ancestor of the leaf (q_i, q_i) and another leaf (q_{i-1}, q_{i-1});
- $\mathcal{A}_T^i$ is the rectilinear tree constructed by algorithm $\mathcal{A}_T$ for serving the first i requests in I; and
- $p_{\mathcal{A}}^i$ is the grid path added to $\mathcal{A}_T^{i-1}$ by algorithm $\mathcal{A}_T$ for serving the i-th request.

We prove this lemma by proving the following two claims. For the i-th request where $1 \leq i \leq k$,

1. the length of $p_{\mathcal{A}}^i$ is shorter than or equal to the length of p_T^i;
2. if there is a grid point $(r, c) \in T^i$, there exists a grid point $(r', c) \in \mathcal{A}_T^i$ with $r' \geq r$.

We prove the claims by induction on i. It is obvious that both claims are true after serving the first request. Consider the i-th request. Let (r_a, c_a) be the least common ancestor of (q_i, q_i) and (q_{i-1}, q_{i-1}) in T. Since (r_a, c_a) must be in T^{i-1}, by the second claim, there is a grid point $(r^*, c_a) \in \mathcal{A}_T^{i-1}$ with $r^* \geq r_a$, and without loss of generality, we assume r^* is the maximum among those grid points in T satisfying the property. Therefore, the grid path $p_{\mathcal{A}}^i$ which runs from (q_i, q_i) to (r^*, c_a) has a shorter length than p_T^i.

Since both grid paths p_T^i and $p_{\mathcal{A}}^i$ are monotone, they span from column q_i to column c_a only. Hence, we need to prove the second claim for grid points within these columns only. Recall that $p_{\mathcal{A}}^i$ contains the horizontal grid line from (q_i, q_i) to (q_i, c_a). Since all grid points in both $\mathcal{A}_T^{i-1}$ and T^{i-1} lie on or above row q_i, the second claim follows.

As a result, we have proved that $|\mathcal{A}_T(I)| \leq |T\|_I|$ because they are the sum of length of $p_{\mathcal{A}}^i$ and sum of length of p_T^i for $1 \leq i \leq k$, respectively. $\square$

5. A good complete rectilinear tree

In this section we describe a complete rectilinear tree S on a triangular grid Δ of size $K \times K$, which provides a good estimate of all optimal rectilinear trees. Precisely, we require $|S\|_I| \leq 3\,|OPT_I|$ for all compact request sequences I.

S has a simple recursive structure. For any integer n, let $f(n) = \lfloor (2/3)(n-1) \rfloor + 1$. Like any complete rectilinear trees, S contains K leaves along the hypotenuse and the root of S is at grid point $(0, K-1)$. The left subtree of the root contains the top $f(K)$ leaves along the hypotenuse and the right subtree contains the rest. The roots of these two subtress are located at grid points $(0, f(K)-1)$ and $(f(K), K-1)$, which are connected to the root of S by a horizontal grid line with $K - f(K)$ grid arcs and a vertical grid line with $f(K)$ grid arcs, respectively. Each subtree is further divided recursively until the subtree contains one leaf only.

It is worth-mentioning that S does not *cross* the rectangle R whose top rightmost corner is at $(0, K-1)$ and bottom leftmost corner at $(f(K), f(K)-1)$.[1] Precisely, S *crosses* R means that there is a path in S entering R at a certain boundary grid point, going through at least one interior grid point, and leaving R at a distinct boundary grid point. Similarly, in each step of recursion, we can also identify a rectangle that S does not cross.

The rest of this section is devoted to showing that $|S\|_I| \leq 3|OPT_I|$ for any compact request sequence I. The key idea is to transform OPT_I into another rectilinear tree O_I with I as leaves and using edges of S only. This transformation is very conservative in replacing the edges of OPT_I so as to ensure that $|O_I| \leq 3|OPT_I|$. On the other hand, by definition, $S\|_I$ contains the minimal subset of grid arcs in S to form a rectilinear tree with with I as leaves and using edges of S only. Thus, O_I contains every grid arc in $S\|_I$ and $|O_I| \geq |S\|_I|$. We conclude that $|S\|_I| \leq |O_I| \leq 3|OPT_I|$.

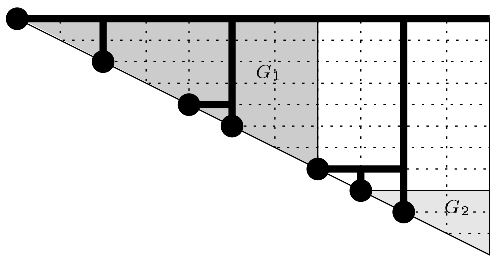
5.1. The recursive transformation

In this section we describe a procedure called TRANSFORM that transforms OPT_I into another rectilinear tree O_I using edges in S only. In the next section we prove that the branch length of O_I is bounded as stated above.

Procedure TRANSFORM is recursive in nature; it takes two parameters T and G. Initially, $T = OPT_I$ and $G = \Delta$ is a triangular grid of size $K \times K$. In general, G is a smaller triangular grid of size $k \times k$ and T is a rectilinear tree embedded on G with its root located at the top rightmost corner of G.

Note that the top rightmost grid point of G is $(0, k-1)$. Let R be the rectangle with the top rightmost corner and bottom leftmost corner at grid points $(0, k-1)$ and $(f(k), f(k)-1)$. I.e., R contains $f(k)+1$ rows and $k - f(k) + 1$ columns. Intuitively, S does not use any edges in R, so Procedure TRANSFORM checks whether T crosses R or not. If so, we remove the grid arcs within R as follows. If T contains a path crossing R from the bottom boundary of R, we expand T to include all the arcs on the bottom and right boundaries of R, excluding the arc on the bottom boundary adjacent to the bottom leftmost corner. Similarly, if T contains a path crossing R from the left boundary of R, we include arcs on the left and top boundaries of R, excluding the arc on the left boundary adjacent to the bottom

[1] R, excluding its bottom leftmost corner, is enclosed entirely in Δ.



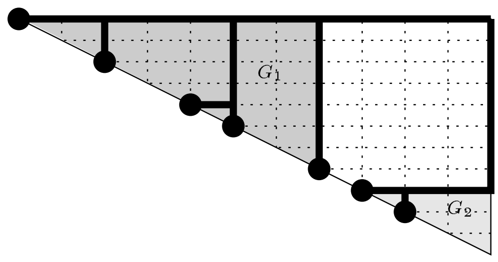


Figure 3. A 12×12 triangular grid G is divided into an 8×8 triangular grid G_1 and a 4×4 triangular grid G_2. (a) An optimal rectilinear tree for a given set of compact request sequence with seven requests. (b) After a recursive step the edges within the rectangle (unshaded area) are removed and the grid arcs on the four boundaries are included.

leftmost corner. Afterwards, we remove all grid arcs of T that are within R. See Figure 3 for an example. Note that new intersections of grid arcs are probably introduced and new nodes will be created on these points. The new tree formed is T'.

We further transform T' recursively as follows. Ignoring the interior grid points of R, we partition G into two disjoint triangular sub-grids: an $f(k) \times f(k)$ triangular grid G_1 to the left of R and an $(k - f(k)) \times (k - f(k))$ triangular grid G_2 below R. See Figure 3 for an example. Let T_1' and T_2' be the subtrees of T' that are embedded on G_1 and G_2, respectively. Due to the way T' is constructed, all paths in T_1' starting from the leaves pass through the top rightmost corner of G_1. Note that T_1' may or may not have a node at the top rightmost corner of G_1; if not, we add a node there. If G_1 is of size at least 3×3 and T_1' contains at least a leaf, we recursively transform T_1' with respect to G_1; otherwise, T_1' remains the same. Similarly, we perform the same operation to transform T_2'.

Note that Procedure TRANSFORM removes all grid arcs from *OPT_I* that are not in S and introduces only grid arcs in S to *OPT_I*. As a result, when the transformation is done, O_I contains only grid arcs in S.

5.2. Analysis of TRANSFORM

Consider a recursive step of Procedure TRANSFORM in which there is a path crossing the rectangle in concern. Let $\mathcal{E}$ and $\mathcal{E}'$ denote the set of grid arcs removed and added , respectively, in the step. Note that $\mathcal{E}$ comprises the grid arcs of the rectilinear tree *OPT_I*, and different $\mathcal{E}$ will contain interior arcs of different non-overlapping rectangles.

Now, we show that for each step, $|\mathcal{E}'| \leq 3\,|\mathcal{E}|$. Let G be the $k \times k$ triangular grid on which we work in a particular step, and R be the corresponding rectangle. The following lemma shows that if *OPT_I* crosses a rectangle R, *OPT_I* has a path crossing R vertically or horizontally.

Lemma 7. *Consider a compact request sequence I and the corresponding* OPT_I. *For any rectangle R, if* OPT_I *crosses R,* OPT_I *contains a path crossing R vertically or horizontally, i.e., the path passes through a single column or a single row of R.*

Proof. By Lemma 3 and Theorem 4, all internal nodes, except possibly the root, of *OPT_I* have two sons. Let R be the rectangle in concern. Suppose *OPT_I* has a path P from a leaf to the root that crosses R. The path P leaves R at either the right boundary or the top boundary. If P leaves R at the right boundary, say on row i, then P or another path in *OPT_I* must go through R horizontally on row i. Otherwise, there exists an arc on row i not involved in *OPT_I*; suppose the rightmost such arc occurs between the grid points (i, x) and $(i, x + 1)$. Note that $(i, x + 1)$ corresponds to a node with at most one son. A contradiction occurs. If P leaves R through the top boundary, say on column j, a similar argument can show that P or another path in *OPT_I* must go through R vertically on column j. Therefore, the lemma follows. □

Since G is a $k \times k$ triangular grid, by construction, the length of the vertical boundary and horizontal boundary of R is $f(k)$ and $2(k - f(k))$, respectively. Lemma 7 implies that we will remove a path that crosses R either vertically or horizontally. Thus $|\mathcal{E}| \geq \min\{f(k), 2(k - f(k))\}$. Note that $|\mathcal{E}'|$ must be smaller than the total length of the four boundaries of R within G, and thus we have $|\mathcal{E}'| \leq 2(f(k) + 2(k - f(k))) - 3 = 4k - 2f(k) - 3$.[2] Based on these two inequalities, a simple calculation will show that

[2]Since the bottom leftmost corner of R is outside G, the total length of the four boundaries of R within G is three less than the perimeter of R.

$|\mathcal{E}'| \leq 4|\mathcal{E}|$. In fact, we can derive a better bound by a more careful analysis.

Lemma 8. *For each recursive step j, we have $|\mathcal{E}'| \leq 3|\mathcal{E}|$.*

Proof. To be given in the full paper. □

Corollary 9. *Given any compact request sequence I, $|O_I| \leq 3\,|\mathrm{OPT}_I|$.*

We conclude the paper by the following theorem.

Theorem 10. *Given any compact request sequence I, we have $|\mathcal{A}_S(I)| \leq 3\,|\mathrm{OPT}_I|$.*

Proof. Recall that $|S\|_I| \leq |O_I|$. By Lemma 6 and Corollary 9, we have $|\mathcal{A}_S(I)| \leq |S\|_I| \leq |O_I| \leq 3\,|OPT_I|$ and the theorem follows. □

References

[1] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. On optimal batching policies for video-on-demand storage servers. In *Proc. 1996 ICMCS*, pages 253–258, Hiroshima, Japan, 17–23 June 1996.

[2] A. Bar-Noy and R. E. Ladner. Efficient algorithms for optimal stream merging algorithms for medio-on-demand. Manuscript, 2000.

[3] A. Bar-Noy and R. E. Ladner. Competitive on-line stream merging algorithms for medio-on-demand. In *Proc. 12th ACM-SIAM SODA*, Washington, DC, 7–9 Jan. 2001.

[4] Y. Cai, K. A. Hua, and K. Vu. Optimizing patching performance. In *Proc. MMCN*, pages 204–215, San Jose, CA, 1999.

[5] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *Proc. 2nd ACM Multimedia*, pages 15–23, San Francisco, CA, 15–20 Oct. 1994.

[6] D. Eager, M. Vernon, and J. Zahorjan. Minimizing bandwidth requirements for on-demand data delivery. In *Proc. 5th MIS*, pages 80–87, Indian Wells, CA, 1999.

[7] D. Eager, M. Vernon, and J. Zahorjan. Optimal and efficient merging schedules for video-on-demand servers. In *Proc. 7th ACM Multimedia*, pages 199–202, Orlanda, FL, Nov. 1999.

[8] L. Gao and D. Towsley. Supplying instantaneous video-on-demand services using controlled multicast. In *Proc. 1999 ICMCS*, pages 117–121, Florence, Italy, 7 - 11 June 1999.

[9] K. A. Hua, Y. Cai, and K. Vu. Patching: A multicast technique for true video-on-demand services. In *Proc. 6th ACM Multimedia*, pages 191–200, Bristol, U.K., Sept. 1998.