Multi-stream Synchronization for 3D Tele-immersive and Collaborative Environment

Zhenyu Yang
School of Computing and Information Sciences
Florida International University
11200 SW 8th, Miami, FL, USA
yangz@cis.fiu.edu

ABSTRACT

The 3D tele-immersive and collaborative environment provides a virtual space for the interaction of remotely dispersed users. To achieve multi-perspective rendering and realistic 3D visual effect, it is needed to transmit multiple semantically correlated 3D video streams from the source to the destination with stringent synchronization requirement. In this paper we discuss the issue of multi-stream synchronization from the general context of the multicast routing with delay and delay variation constraints, which was proved as an NP-complete problem. Then we propose a heuristic to construct a multicast network on an overlay content dissemination architecture for the solution, and show that our algorithm is asymptotically more advanced in the time complexity than existing ones. Empirical studies further verify the performance of our algorithm regarding to the temporal efficiency in various sizes of input data.

Keywords

3D tele-immersion, delay and delay variation constraints, multicast

1. INTRODUCTION

The 3D tele-immersive and collaborative environment provides a virtual space for the interaction of remotely dispersed users [6]. In such an environment, multiple 3D cameras are mounted to capture a physical scene from various spatial points, with each 3D camera corresponding to one 3D video stream [8]. With an overlay content dissemination architecture [9], those streams are transmitted to the destination where a joint virtual space immersing the participants is rendered from multiple perspectives in real time. The proposed multicast-based overlay architecture allows a more efficient multi-stream content delivery [9]. Under such mechanism, a user can dynamically manipulate his/her viewpoint and the system automatically responses by selecting an appropriate subset of streams to be transmitted. Further, users of similar viewpoints may share the streams of the intersection instead of requesting them directly from the source (Figure 1).

It is observed that there is a tight semantic correlation among multiple streams generated from the same scene acquisition source as the underlying 3D cameras are calibrated and highly synchronized. To extend the general concept of a video frame, we define a set of n 3D video frames captured from n cameras bearing the same timestamp as a *macro-frame* [8]. In this paper, we are interested in the problem of synchronized transmission of macro-frames. That is, each individual frame of one macro-frame must be received by the rendering sites within a bounded delay window (the δ in Figure 1). As pointed out in [4], the synchronization issue of macro-frame delivery is very crucial for 3D tele-immersive systems based on the multi-stream model, which has strong impact on the 3D rendering and the support for real-time interaction.

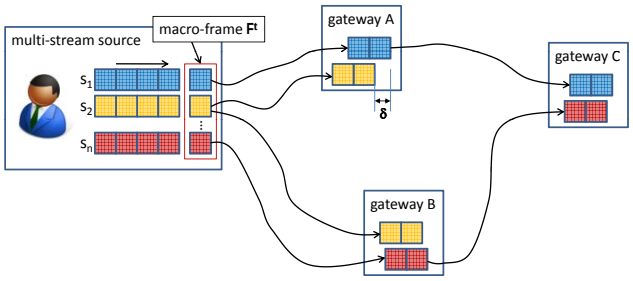

Figure 1: Multi-stream distribution architecture

The synchronization issue of the overlay and multicast networks has been brought to the attention of the community with the problem formulated as delay and delay variation bounded multicast network (DVBMN) [5, 2, 1]. The DVBMN problem concerns about building a multicast network spanning from one source node to a set of destination nodes to satisfy the quality-of-service (QoS) requirements on not only the maximum delay of every path but also the maximum delay variation between any pair of paths.

Due to the NP-completeness of the DVBMN problem [5], several heuristics are proposed in the literature [5, 2, 7, 1]. As shown in recent research trends, one promising approach for solving the problem is to first derive the k shortest paths from the source node to every destination node as the candidate set where the maximum delay bound is imposed. This first step is carried out using the most efficient k shortest paths algorithm (e.g., [3]). If we define m as the number of

IMMERSCOM '09, May 27-29, 2009, Berkeley, CA USA.

destination nodes, the candidate set will contain at most mk paths. Then in the second step, paths from the candidate set are compared with each other to search for an optimal subset of paths with the minimum delay variation. By far, the most efficient algorithm in the second step achieves the tightest delay variation bound under the time complexity of $O(m^2k)$ with a somewhat complicated procedure [1].

The solution we propose in this paper for the multi-stream synchronization follows the similar two-step framework. However, for the second step, we apply a different procedure for comparing candidate paths. Not only is the new procedure much simpler, it also achieves the same level of minimum delay variation as in [1] but much more efficiently with the time complexity of $O(mk\log(m))$.

The paper is organized as follows. The DVBMN problem is formulated in Section 2. The solution is given in Section 3. The evaluation of performance is presented in Section 4. The conclusion and future work are discussed in Section 5.

2. PROBLEM DEFINITION

The DVBMN problem is defined as follows. There is a directed graph $G = \langle V, E\rangle$ where V denotes the set of vertices and E the set of edges. An edge delay function $\mathcal{D} : E \rightarrow \Re^+$ is defined which assigns a non-negative delay to each edge. Given a single source vertex $s \in V$ and a set of destination vertices $M \subseteq V - \{s\}$ (let $m = |M|$), the contents are transmitted through a multicasting subgraph $T = \langle V_T, E_T\rangle$. The subgraph T spans between the source vertex s to every vertex $v \in M$. Let $p_T(s, v)$ denote the path from the source vertex s to one destination vertex v through the edges in E_T. The delay along the path is expressed as $\Sigma_{e\in p_T(s,v)}\mathcal{D}(e)$.

The problem has two bounding parameters, namely the *delay tolerance* Δ and the *delay variation tolerance* δ. The goal is to construct a particular T such that constraints (1) and (2) as shown below are satisfied.

$$\forall v \in M, \quad \sum_{e\in p_T(s,v)} \mathcal{D}(e) \leq \Delta \tag{1}$$

$$\forall v, u \in M, \quad \left| \sum_{e\in p_T(s,v)} \mathcal{D}(e) - \sum_{e\in p_T(s,u)} \mathcal{D}(e) \right| \leq \delta \tag{2}$$

The DVBMN problem as described above is shown to be NP-complete [5], which has aroused lots of attention to search for efficient heuristics [5, 2, 7, 1]. The interested readers may refer to [1] for a brief survey. Meanwhile, to evaluate the performance of those algorithms, a third parameter δ_T is defined as in (3),

$$\delta_T = \max_{\forall u,v\in M} \left(\left| \sum_{e\in p_T(s,v)} \mathcal{D}(e) - \sum_{e\in p_T(s,u)} \mathcal{D}(e) \right| \right) \tag{3}$$

which is called the *maximum delay variation*. To simplify the expression, from now on we use $\mathcal{D}(p)$ to denote the additive delay of path p (i.e., $\mathcal{D}(p) = \sum_{e\in p}\mathcal{D}(e)$). Thus, we can rewrite 3 as below.

$$\delta_T = \max_{\forall u,v\in M} \left(|\mathcal{D}(p_T(s, u)) - \mathcal{D}(p_T(s, v))| \right) \tag{4}$$

3. SOLUTION

By far, one promising approach for the solution of DVBMN problem was based on the k shortest paths algorithm (e.g., [3]) with two major steps. In the first step, the k shortest paths algorithm is executed which returns a path list $\mathcal{P}(v)$ for every $v \in M$. Each path list $\mathcal{P}(v)$ contains at most k shortest paths from the source vertex s to the destination vertex v (i.e., $p_G(s, v)$). Also, all the paths in $\mathcal{P}(v)$ satisfy the delay tolerance (i.e., Δ) and are sorted in the ascending order of path delay. Note, the value of k is pre-selected according to certain criteria such as the graph size, edge density and the number of destinations [1].

The k shortest paths algorithm generates m path lists with each list containing at most k paths. Thus, there are at most mk paths which forms the candidate set. The second step examines those paths of the candidate set to derive an optimal set of paths (i.e., T).

In [1], Banik et al. proposed an algorithm to search for the optimal T in terms of minimizing δ_T. Clearly, a naive approach would exhaustively search all the possible combinations with a time complexity of $O(k^m)$. Hence, one important contribution made by Banik et al. was their search algorithm that returned the optimal set of paths with the time complexity of $O(m^2k)$.

The solution we propose in this paper follows the similar framework as above. However, in the second step we use an even more efficient search algorithm which achieves the same result as [1] but with the time complexity of $O(mk\log(m))$. Meanwhile, our algorithm is much simpler compared to theirs.

For the rest of the section, we introduce our algorithm in two parts. In the first part, we focus on illustrating the main idea by using a more straightforward but a *little slower* algorithm with the time complexity of $O(m^2k)$. Note that, even this slower algorithm has already matched the best performance so far. Next, we describe how to evolve this algorithm to achieve the time complexity of $O(mk\log(m))$.

3.1 The Slower Algorithm

We present the algorithm by referring to a sample graph in Figure 2. To simplify the graph, we assume each edge is bi-directional and has symmetric delay. The source vertex is v_s and the set of destination vertices is $M = \{v_2, v_6, v_8\}$.

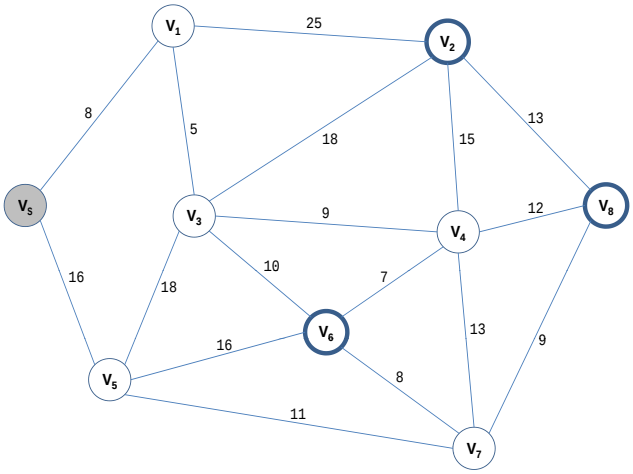

Figure 2: A sample graph

Path List	k Shortest Paths (k = 4)	Path Delay
$P(v_2)$	$v_s \to v_1 \to v_3 \to v_2$	31
	$v_s \to v_1 \to v_2$	33
	$v_s \to v_1 \to v_3 \to v_4 \to v_2$	37
	$v_s \to v_1 \to v_3 \to v_6 \to v_4 \to v_2$	45
$P(v_6)$	$v_s \to v_1 \to v_3 \to v_6$	23
	$v_s \to v_1 \to v_3 \to v_4 \to v_6$	29
	$v_s \to v_5 \to v_6$	32
	$v_s \to v_5 \to v_7 \to v_6$	35
$P(v_8)$	$v_s \to v_1 \to v_3 \to v_4 \to v_8$	34
	$v_s \to v_5 \to v_7 \to v_8$	36
	$v_s \to v_1 \to v_3 \to v_6 \to v_7 \to v_8$	40
	$v_s \to v_1 \to v_3 \to v_6 \to v_4 \to v_8$	42

Figure 3: The results of k shortest paths

Then, the k shortest paths algorithm is executed with $k = 4$. The results are shown in Figure 3. There are three points to be noted here. First, as mentioned earlier all the paths in the same path list are sorted in the ascending order of delay. Second, generally some path lists may contain less than k paths. It will not affect the correctness of our algorithm. However, to ease the description we assume every resultant path list contains k paths. Third, the value of delay tolerance Δ is not specified here but the readers should be aware that during the execution of k shortest paths algorithm the constraint of Δ is already applied.

We denote the candidate sets from the k shortest paths algorithm as $\mathcal{P}(v)$ for $v \in M$. For convenience, we apply a total order for all the vertices in M (i.e., $M = \{v_1, v_2, ..., v_m\}$). Then we can simplify the notation of $\mathcal{P}(v_i)$ as $\mathcal{P}_i$. Given a set of path lists $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_m\}$, we say that a path set P is *derived* from $\mathcal{P}$ if and only if $P = \{p_1, p_2, ..., p_m\}$ and $p_i \in \mathcal{P}_i$ for $1 \le i \le m$. The goal of the second step is to find an optimal set of paths T derived from $\mathcal{P}$ such that the maximum delay variation δ_T is minimized (as shown in (5)).

$$T = \underset{P \text{ derived from } \mathcal{P}}{\arg\min} \left(\max_{\forall p_i, p_j \in P} (|\mathcal{D}(p_i) - \mathcal{D}(p_j)|) \right) \quad (5)$$

The pseudo-code of the search algorithm is given in Table 1. After the execution of the program, the optimal set of paths is saved in T where $T[i]$ stores the path from s to v_i for $v_i \in M$ (note that, $T[i] \in \mathcal{P}_i$), and the related minimum δ_T is saved in the variable of min_δ_T. Recall that the set $\mathcal{P}_i$ is sorted in ascending order of path delay. Therefore, the first element (denoted as p_i) always holds the path of minimum delay in the current set of $\mathcal{P}_i$ and we can keep removing the first element without violating this property (as in Line 4, 18, and 22 of the pseudo-code).

Before we examine the algorithm in more detail, let us first get an intuitive sense of how the program actually runs using the aforementioned sample graph (Figure 2) and the output from the k shortest paths (Figure 3). Figure 4 illustrates the runtime snapshot of the search algorithm with each grid representing one iteration of the while loop (Line 7 to 24 in Table 1). For example, in the grid [1] each column of the table shows the initial elements of path list $\mathcal{P}_i$ and the paths of minimum delay of each path list are located in

Table 1: The pseudo-code of searching for the optimal combination of paths to minimize δ_T

```
1  search(P_1, P_2, ..., P_m)
2      min_δ ← ∞
3      for i = 1 to m do
4          p_i ← the 1st element of P_i
5      end for
6      done ← false
7      while not done do
8          p_min ← arg min_{p∈{p_1,p_2,...,p_m}} (D(p))
9          p_max ← arg max_{p∈{p_1,p_2,...,p_m}} (D(p))
10         D ← D(p_max) − D(p_min)
11         if D < min_δ then
12             min_δ ← D
13             for i = 1 to m do
14                 T[i] ← p_i
15             end for
16         end if
17         let l be the index of p_min s.t. p_min ∈ P_l
18         P_l ← P_l − {p_min}
19         if P_l = ∅ then
20             done ← true
21         else
22             p_l ← the 1st element of P_l
23         end if
24     end while
```

the first row.* Since $p_{min} = 23$ and $p_{max} = 34$, the current min_δ is 11 (as shown beside the right arrow). After that, the smallest delay (i.e., 23) is removed and the program evolves to the next iteration until one of the path lists becomes empty. The minimum δ_T is 2, which is found the first time in the fourth iteration and the program terminates at the seventh iteration where $\mathcal{P}_2 = \emptyset$. It is noted that several combinations achieve the same minimum δ_T (e.g., the fifth and the seventh iteration). In that case, only the one with minimum delay is selected as in the fourth iteration, achieving small *stretch*. Figure 5 shows the selected paths spanning from v_s to v_2, v_6 and v_8.

Time Complexity

For the search algorithm listed in Table 1, the time complexity of Line 3 to 5 (i.e., the for loop) is $O(m)$. Within the while loop, Line 8 takes $O(m)$ to find the minimum element within m elements and the same for Line 9. The for loop to keep track of the optimal set (Line 13 to 15) takes $O(m)$. The rest lines within the while loop take $O(1)$. Each iteration removes one path (Line 18). Since there are at most mk paths to be removed, the maximum number of iterations is mk. Therefore, the total time complexity is $O(m^2k)$. The performance matches the best algorithm proposed so far but is significantly simpler.

Correctness

The correctness of the slower algorithm is stated as the following theorem. We leave the proof in Appendix.

THEOREM 3.1. *When the search algorithm terminates, it finds the minimum delay variation (i.e., δ_T) for any set of paths P derived from the given set of path lists $\mathcal{P}$.*

* To save space, only the delay is shown and the actual path is omitted.

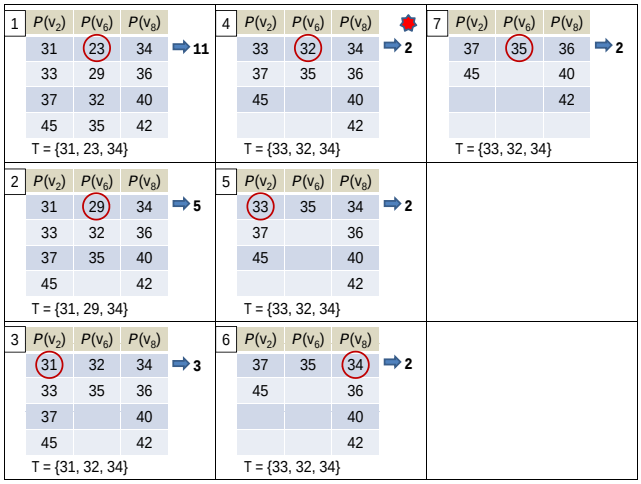

Figure 4: The snapshot of the search algorithm

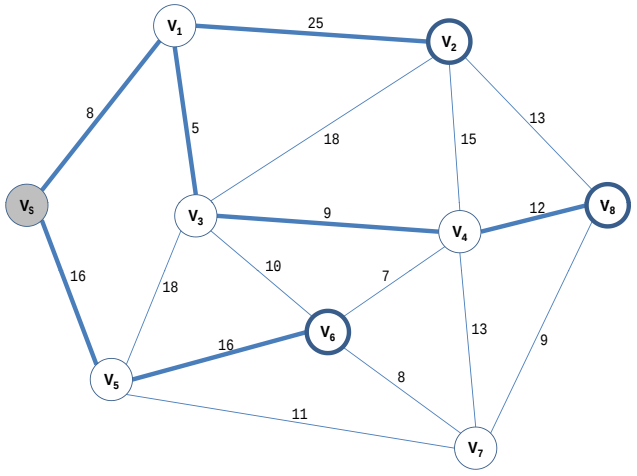

Figure 5: The selected multicast paths with the minimum δ_T

3.2 The Faster Algorithm

As mentioned earlier, the slower algorithm has the time complexity of $O(m^2k)$. The slower algorithm is used to illustrate the idea and as the basis for the improvement. If we examine the pseudo-code more carefully, we assume Line 8 and 9 use linear search to find the path of minimum and maximum delay which has the time complexity of $O(m)$. Obviously, we could improve this part by using more advanced searching algorithm such as the priority queue. The faster algorithm is listed in Table 2.

The faster algorithm is essentially the same as the slower algorithm except the usage of priority queue to accelerate the searching. Note that, we only use priority queue to search for p_{min} and we simply use one variable p_{max} to keep track of the current maximum delay path. For brevity, we will not spend more space to discuss its correctness. For the time complexity, Line 4 to 10 take $O(m\log(m))$ to insert m elements into the priority queue. Inside the while loop (Line 12 to 30), each iteration takes $O(\log(m))$ for the enqueue and dequeue operations. The total number of iterations is $O(mk)$. Therefore, the total complexity is $O(mk\log(m))$.

Table 2: The pseudo-code of searching for the optimal combination of paths to minimize δ_T

```
1   search(P_1, P_2, ..., P_m)
2     min_δ ← ∞
3     p_max ← 0
4     for i = 1 to m do
5       p ← the 1st element of P_i
6       insert p to a priority queue: q.enqueue(p, D(p))
7       if D(p) > D(p_max) then
8         p_max ← p
9       end if
10    end for
11    done ← false
12    while not done do
13      p_min ← q.head()
14      D ← D(p_max) − D(p_min)
15      if D < min_δ then
16        min_δ ← D
17      end if
18      let l be the index of p_min s.t. p_min ∈ P_l
19      P_l ← P_l − {p_min}
20      if P_l = ∅ then
21        done ← true
22      else
23        q.dequeue()
24        p ← the 1st element of P_l
25        q.enqueue(p, D(p))
26        if D(p) > D(p_max) then
27          p_max ← p
28        end if
29      end if
30    end while
```

It is observed that the faster algorithm does not generate the optimal combination of paths (i.e., T). In the slower algorithm, the step of keeping track of T takes $O(m)$ and it could happen at each iteration in the worst case. A simple way to fix it is to run the faster algorithm twice. In the first run we record δ_T, and in the second run we use δ_T to decide when to record T (for only once). Since each run takes $O(mk\log(m))$, the total complexity is still $O(mk\log(m))$.

The time complexity of the most efficient k shortest paths algorithm is $O(|E|+|V|k\log(|E|/|V|))$ such as in [3]. Adding the cost of Banik's algorithm, the total time complexity of minimum delay variation algorithm would be $O(|E| + |V|k\log(|E|/|V|) + m^2k)$ [1]. Thus, with our new algorithm, the time complexity is reduced to $O(|E|+|V|k\log(|E|/|V|)+ mk\log(m))$, which can be simplified as $O(|E|+|V|k\log(|E|))$.

4. EVALUATION

We implement the slower and faster version of the search algorithm. For comparison, we also implement a variant from the algorithm proposed by Banik et al., called *Chains* [1]. The time complexity of Chains is $O(m^2k)$, similar to the slower search algorithm. However, it is more complicated which involves several scans. In the first scan, all path lists (i.e., $\{\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_m\}$) are merged into one list in the ascending order of path delay with the time complexity of $O(mk\log(m))$.[†] After that, some data structures are ini-

[†] In their paper, the authors claimed the merge can be

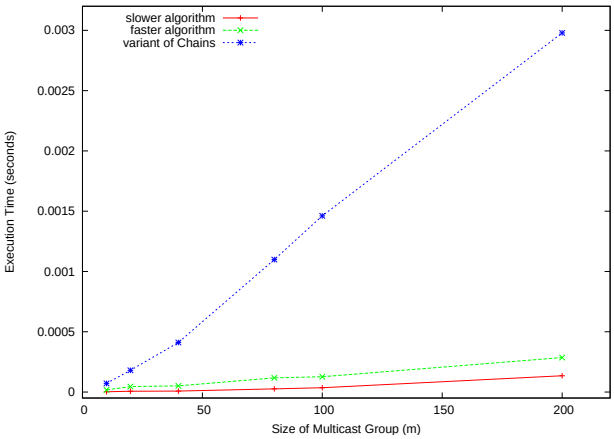

Figure 6: A comparison of execution time of slower algorithm, faster algorithm and Chains ($k = 6$)

tialized in a second scan over the merged list with the time complexity of $O(m^2k)$. Then, a third scan is performed to finally output the optimal set of paths. Compared with their approach, the slower program only takes one scan.

From the description in their paper, we are not very sure about the details of the second scan of Chains. Thus, we substitute it with a routine that establishes somewhat similar data structures but with the time complexity of $O(mk)$. By taking this approach, we feel confident that although they are not exactly the same, nevertheless the variant shall not run slower than the Chains algorithm to justify the comparison.

Since both our algorithm and the Chains algorithm assume the output from the k shortest paths algorithm, to facilitate the experiment we generate random path lists with various combinations of m and k for the performance evaluation. We set k to be 6 and select m ranging from 10 up to 1000. The results are plotted in Figure 6 where each point is calculated over an average of 6 runs.

Figure 6 demonstrates the advantage of the slower algorithm over the faster algorithm with small to medium size of input. Meanwhile, although asymptotically similar, the slower algorithm runs much faster than the Chains algorithm due to its simplicity. For larger input size, Table 3 illustrates the performance of the faster algorithm as compared with that of the slower algorithm.

Table 3: A comparison of execution time of slower algorithm and faster algorithm with large input size (Unit: second)

	$m = 400$	$m = 600$	$m = 800$	$m = 1000$
slower	0.000186	0.001157	0.001684	0.002095
faster	0.000319	0.000811	0.000930	0.001029

As shown in Table 3, the faster algorithm starts to win over its slower counterpart at the point of $m = 600$. At the current state, a multicast group for collaborative work may not involve such big size. Therefore, it seems that the slower algorithm would be a better choice in most of the situations. However, we envision that in the near future with the increasing popularity of multi-site/multi-stream 3D tele-immersive and collaborative environments, we may need to consider the synchronization issue for large scale collaborative group. In those scenarios, a multicast session involving hundreds of nodes would become more usual than bizarre and the faster algorithm will certainly show its potential.

done in $O(mk)$ which we believe is incorrect.

5. CONCLUSION AND FUTURE WORK

In this paper, we investigate the issue of multi-stream synchronization for the support of 3D tele-immersive and collaborative environment. The problem is modeled under the context of delay and delay variation bounded multicast network (DVBMN), which is NP-complete. Thus, we study one promising approach based on the k shortest paths algorithm. The approach leverages on the execution of the k shortest paths algorithm to derive a candidate set of mk shortest paths from one single source to m destinations. Later, a search procedure is applied to find a set of paths out of the candidate set to minimize the maximum delay variation. By far, the best searching algorithm achieved the time complexity of $O(m^2k)$. As an improvement, we propose two versions of a new searching algorithm. For the slower version, its asymptotic complexity matches that of the previously proposed algorithm but is much faster in practical experiments due to its simplicity. Furthermore, the faster version achieves the time complexity of $O(mk \log(m))$ and shows its potential over the slower version with larger input size of data.

So far, the issue of synchronizing multicast group has only involved with single source. As our next step, we will consider the synchronization problem with multiple sources, which is very much needed for the research community of immersive and collaborative applications. Also, we will investigate the embedding of the synchronization component into the current multi-stream delivery architecture (e.g., [9]) and the issue of dynamics.

6. REFERENCES

[1] S. M. Banik, S. Radhakrishnan, and C. N. Sekharan. Multicast routing with delay and delay variation constraints for collaborative applications on overlay networks. *IEEE Transactions on Parallel and Distributed Systems*, 18(3):421–431, 2007.

[2] S. Kapoor and S. Raghavan. Improved multicast routing with delay and delay variation constraints. In *GLOBECOM '00: Proceedings of the Global Telecommunication Conference*, volume 1, pages 476–480, Irvine, CA, USA, 2000.

[3] V. Manuel, J. Pelayo, and A. M. Varo. Computing the k shortest paths: A new algorithm and an experimental comparison. In *WAE '99: Proceedings of the 3rd International Workshop Algorithm Engineering*, pages 15–29, London, UK, 1999.

[4] D. E. Ott and K. Mayer-Patel. Coordinated multi-streaming for 3d tele-immersion. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 596–603, New York, NY, USA, 2004. ACM Press.

[5] G. N. Rouskas and I. Baldine. Multicast routing with end-to-end delay and delay variation constraints. *IEEE Journal on Selected Areas in Communications*, 15(3):346–356, 1997.

[6] R. Sheppard, M. Kamali, R. Rivas, M. Tamai, Z. Yang, W. Wu, and K. Nahrstedt. Distributed virtual collaboration through tele-immersive dance (ted): A symbiotic creativity and design environment for art and computer science. In *MULTIMEDIA '08: Proceedings of the 16th annual ACM international conference on Multimedia*, pages 579–588, Vancouver, Canada, 2008.

[7] P.-R. Sheu and S.-T. Chen. A fast and efficient heuristic algorithm for the delay- and delay variation bound multicast tree problem. In *ICOIN '01: Proceedings of the 15th International Conference on Information Networking*, pages 611–618, Beppu City, Oita, Japan, 2001.

[8] Z. Yang, K. Nahrstedt, Y. Cui, B. Yu, J. Liang, S. hack Jung, and R. Bajscy. Teeve: the next generation architecture for tele-immersive environments. In *ISM '05: Proceedings of the 7th IEEE International Symposium on Multimedia*, pages 112–119, Irvine, CA, USA, 2005.

[9] Z. Yang, W. Wu, K. Nahrstedt, G. Kurillo, and R. Bajcsy. Viewcast: view dissemination and management for multi-party 3d tele-immersive environments. In *MULTIMEDIA '07: Proceedings of the 15th international conference on Multimedia*, pages 882–891, New York, NY, USA, 2007. ACM Press.

APPENDIX

A. PROOF OF THEOREM 3.1

We prove the theorem based on the techniques of loop invariant and induction. To distinguish the state of variables from one iteration to another, during the proof we use the *superscript notation*, for example $\mathcal{P}^j$ indicates the content of $\mathcal{P}$ in the jth iteration. To start with, denote p^j as the path of minimum delay in the jth iteration of the while loop (actually, it should be p^j_{min} to be more consistent but we simplify the notation here). From the pseudo-code (Table 1), it is easy to observe that $\mathcal{D}(p^{j_1}) \leq \mathcal{D}(p^{j_2})$ for any $j_1 < j_2$.

For convenience, given P derived from $\mathcal{P}$ define an *index* function id for $p \in P$ where $id(p) = i$ if and only if $p \in \mathcal{P}_i$. Also, define $min(P) = \arg\min_{p \in P}(\mathcal{D}(p))$ and so as $max(P)$. We will notify in case such usage of min and max may cause confusion. First, we prove the following lemma.

LEMMA A.1. *Given a set of path lists $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_m\}$, the jth iteration finds min_$\delta = min(\delta_P)$ where P is any set of paths derived from $\mathcal{P}$ and P includes at least one path from $\{p^1, p^2, ..., p^j\}$.*

PROOF. The proof is by the induction on j, the number of iterations. First, consider the base case of $j = 1$. Define $P^1 = \{min(\mathcal{P}^1_1), min(\mathcal{P}^1_2), ..., min(\mathcal{P}^1_m)\}$. Thus, $p^1 = min(P^1)$, which is the path of minimum delay in the first iteration. According to Line 10, the min_δ is calculated as follows.

$$\texttt{min\_}\delta = \mathcal{D}(max(P^1)) - \mathcal{D}(p^1) \tag{6}$$

We only need to show that min_$\delta \leq min(\delta_P)$ for any set of paths P derived from $\mathcal{P}$ which includes p^1. By contradiction, suppose there is another path set Q with Q derived from $\mathcal{P}$ and Q includes p^1, and $\delta_Q <$ min_δ.

Since Q includes p^1 and p^1 is the path of minimum delay over all paths in $\mathcal{P}$. The calculation of δ_Q must include p^1 as the minimum delay path according to its definition, such that $\delta_Q = \mathcal{D}(max(Q)) - \mathcal{D}(p^1)$. Let $q_{max} = max(Q)$ and $p^1_{max} = max(P^1_{min})$. Therefore, we have that

$$\mathcal{D}(q_{max}) - \mathcal{D}(p^1) < \mathcal{D}(p^1_{max}) - \mathcal{D}(p^1) \tag{7}$$

or

$$\mathcal{D}(q_{max}) < \mathcal{D}(p^1_{max}). \tag{8}$$

Obviously, $id(q_{max}) \neq id(p^1_{max})$. Since $\mathcal{D}(p^1_{max})$ is the path of minimum delay in its path list, its delay should not be bigger than another path in the same path list. Let $i_1 = id(q_{max})$ and $i_2 = id(p^1_{max})$ (so $i_1 \neq i_2$). Let $q \in Q$ and $id(q) = i_2$. Then, the following must hold.

$$\mathcal{D}(q_{max}) \geq \mathcal{D}(q) \geq \mathcal{D}(p^1_{max}) \tag{9}$$

This contradicts to (8).

Next, we assume the induction hypothesis (IH) that the Lemma A.1 is true for $j = n - 1$ and consider the case of $j = n$. Again, let $P^n = \{min(\mathcal{P}^n_1), min(\mathcal{P}^n_2), ..., min(\mathcal{P}^n_m)\}$, $p^n = min(P^n)$ and $p^n_{max} = max(P^n)$. There are two possibilities regarding to min_δ (case (a) and (b) below).

(a) min_$\delta \leq \mathcal{D}(p^n_{max}) - \mathcal{D}(p^n)$. In this case, min_$\delta$ remains unchanged. Now consider any set of paths P derived from $\mathcal{P}$ and P includes at least one path from $\{p^1, p^2, ..., p^{n-1}, p^n\}$. (a.1) Suppose P only includes path from $\{p^1, p^2, ..., p^{n-1}\}$, then by the IH min_$\delta < \delta_P$. (a.2) Suppose P only includes p^n but none of $\{p^1, p^2, ..., p^{n-1}\}$, which is actually the same situation as $j = 1$ since p^n is *currently* the path of minimum delay. Thus, we have $\mathcal{D}(p^n_{max}) - \mathcal{D}(p^n)$ as the minimum variation for any set of paths P derived from $\mathcal{P}$ which includes p^n but none of $\{p^1, p^2, ..., p^{n-1}\}$. Combining (a.1) and (a.2), it is proved that min_δ is the minimum delay variation for any set of paths P derived from $\mathcal{P}$ which includes at least one path from $\{p^1, p^2, ..., p^{n-1}, p^n\}$.

(b) min_$\delta > \mathcal{D}(p^n_{max}) - \mathcal{D}(p^n)$. In this case, min_$\delta$ is assigned the new value of $\mathcal{D}(p^n_{max}) - \mathcal{D}(p^n)$. The rest of the proof is similar to that of (a) and is omitted for brevity.

Therefore, Lemma A.1 is proved. □

Theorem 3.1 is now straightforward given the previous lemma. First, the search algorithm checks one path in every iteration and will never check it again. Thus, it will terminate either after all the paths are checked or one of the path lists, namely $\mathcal{P}_l$, is exhausted (i.e., $\mathcal{P}^j_l = \emptyset$). Second, due to Lemma A.1, the algorithm guarantees to find the minimum delay variation for any derived set of paths which includes at least one path from $\{p^1, p^2, ..., p^j\}$ in the j-th iteration. Therefore, when the search terminates it will find the minimum delay variation for any derived set of paths P, since any such P would include at least one path from the whole set of paths (i.e., $\mathcal{P}$) or $\mathcal{P}_l$. □