A Variation of Minimum Latency Problem

Yung-Hui Huang* Yaw-Ling Lin† Chuan-Yi Tang‡

Abstract

In this paper we study the variation of the *minimum latency problem*(MLP) [2]. The MLP is to find a walk tour on the graph $G(V, E)$ with a distance matrix $d_{i,j}$. Where $d_{i,j}$ indicate the distance between v_i and v_j. Let $\ell(v_i)$ is the latency length of v_i, defined to be the distance traveled before first visiting v_i. The minimum latency tour is to minimize $\sum_{i=1}^{n} \ell(v_i)$. In some message broadcast and scheduling problem [8] the vertex also has latency time and weight. Those problem need to extend the objective function of the minimum latency tour as $\sum_{i=1}^{n} \ell(v_i)w(v_i)$. The definition is equivalent to the MLP with no edge distance but vertex latency time and vertex weight. We give an linear algorithm for the unweighted full k-ary tree or k-path graphs, and $O(n \log n)$ time for general tree graphs. The time complexity in trees is the same as Adolphson's result; however, the algorithm given here is not only simpler, easier to understand, but also more flexible and thus can be easily extended to other classes of graphs.

Keywords: minimum latency problem, linear ordering, broadcast network, trees, k-path graphs

1 Introduction

Consider a worker (a processor,or a repairman) facing a set of jobs with precedence and the latency time to initialize before job start(latency of edges), each of jobs is a vertex with latency time and weight(latency and weight of vertices). The worker must schedule its visits so as to minimize the total latency time the jobs wait before being complete. (We assume that the walk back latency of completed jobs are zero.) This is a simple and natural combinatorial optimization problem faced in realistic world, and may be formalized as follows.

In mobile environment, users retrieve information by portable devices. Since the mobile devices usually have limited power, the issue of minimization the data access latency is important. Periodic broadcasts of frequently requested data can thus reduce the traffics in the air and save the powers of the mobile devices. However, users need to wait for the required data to appear on the broadcast channel. It follows that the more time they wait then the more power devices have to consume. Finding the minimum latency tour can thus help us in solving this kind of problem.

Several variations of this problem was focused on index and data allocation in a single broadcast channel using conventional techniques [4, 5]. In general trees, this problem was solved in $O(n \log n)$ time [1]. The issue of multiple broadcast channels is addressed in [7], and the problem becomes NP-hard [6]. Further, there are discussion using prune strategies for multiple broadcast channel and heuristic answers form single broadcast channel [6].

Given a directed acyclic graph G with a set of job vertices $v_1 \dots v_n$ with weight $w(v_i)$, latency time $t(v_i)$ and latency edges $e_1 \dots e_m$ with distance $d(e_j)$. Let T be a tour which completes the job vertices in precedence order. If the vertex v_i wants to be completed, at least one ancestor of v_i must be finished before. Further, because the walk back cost is zero, T does not need to include the walk back edge and job vertex already been finished. Let the latency $\ell(v_i)$ of a vertex v_i be the sum of the edge and vertex latency time of the tour from the staring point to v_i. We can define the total latency $L(T) = \sum_{i=1}^{n} \ell(v_i)w(v_i)$. We say that a tour T_i is the minimum latency tour if $L(T_i)$ has the smallest value among all possible tours of G.

In this paper, we investigate tree graphs, tree, full k-ary trees and k-path graphs. We propose a polynomial time algorithm for general trees to get the optimal solution in $O(n \log n)$ using a greedy manner, a linear time DFS-like algorithm for full k-ary tree and a $O(n \log k)$ time algorithm for k-path graphs. Figure 1 gives an example of the dags and its minimum latency tour.

2 Basic Notations and Properties

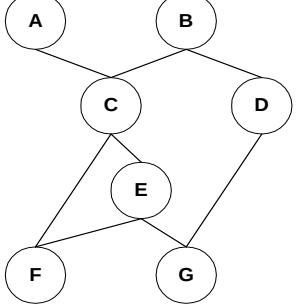
Intuitively, we would want to place the heaviest vertex at the front of visited tour as early as possible. However, at least one ancestors of the heaviest vertex must be placed before, and it might not be wise exploring these vertices too early. This simple observation suggests that we would like to explore the properties of the tree graphs, for short, in a bottom-up manner.

Without loss of generality, we can add the weight $w(v)$ and latency time $t(v)$ of vertex v into all edges

*Department of Computer Science, National Tsing-Hua University, Taiwan, R.O.C. E-mail: yeong@cs.nthu.edu.tw.

†Department of Computer Science and Information Management. Providence University. E-Mail: yllin@pu.edu.tw. This work was partially supported by NSC 89-2213-E-126-003.

‡Department of Computer Science, National Tsing-Hua University, Taiwan, R.O.C. E-mail: cytang@cs.nthu.edu.tw.



1. Weight of vertices

	A	B	C	D	E	F	G
w(vi)	30	0	45	50	42	65	101

2. Work Time of vertices t(vi) is zero.
3. All edge delay d(ej) is 1

Result:
T = {A,C,E,G,F,B,D}
L(T) = 1*30+2*45+3*42+4*101+5*65+6*0+7*50
= 1325

Figure 1: The minimum latency tour of dags

which incident to v. Let $w(e) = w(v)$ and $d(e) = d(e) + t(v)$.

Given a tree T, we say a subtour $T_k = \langle e_i, \ldots, e_j \rangle$ is *inseparable* if and only if there exists a minimum latency tour on T containing this subsequence T_k contiguously. That is, $T_k \subseteq T$, and all adjacent edges of T_k are also adjacent to each other in T. Let $d(T_k)$ denote the sum of distances $d(e)$'s for each edge e in T_k, and $w(T_k)$ donate the sum of weights $w(e)$'s. Further, we define the *ratio cost* of T_K by $c(T_k) = \frac{w(T_k)}{d(T_k)}$. By definition, each edge e of G is initially inseparable subtour and have $c(e) = \frac{w(e)}{d(e)}$.

The idea here is to iteratively *glue* the child and parent vertices and find its minimum latency tour for the new sub-graph. Taking advantage of the nice properties of trees, we can glue the inseparable subtour that has the maximum ratio cost with its parent such that edges of the trees are grouped into a new inseparable subtour. If an inseparable subtour T'_k is the root edge and has maximum ratio cost then T'_k can be output directly.

From above idea, we found that there are some interesting properties of trees. Section 3 introduce the $O(n \log n)$ algorithm for tree graph, and provides some properties for trees. Those properties are also used in Section 4 to provide a DFS-like algorithm for the special case "full k-ary tree". The algorithm in Section 4 is $O(n)$. After the tree and its special case, we discuss another case called "k-path graphs" in Section 5.

3 Algorithm for General Trees

In this section we prove some property of general tree graphics. According to those properties an $O(n \log n)$ time algorithm for tree naturally evolves.The idea here is iterative select a maximum ratio cost edge and shrinking with is parent or output the selected subtour until all vertex is be output; totally it will spend $O(n)$ time to obtain a Fibonacci heap. Second, we uses $O(\log n)$ time to extract maximum and decrease the ratio cost of the parent edge exactly n times. The output result is then the minimum latency tour.

First we show when to glue two inseparable subtours into one:

Lemma 3.1 *Let a and b be two inseparable subtours of G, and a is the parent of b. Further, b has the largest ratio cost among all others. It follows that ab is inseparable subtour.*

Proof. By contradiction, assume that there is a nonempty subsequence α of edges of the trees such that $L(a\alpha b) < \min\{L(\alpha ab), L(ab\alpha)\}$. First we assume $c(\alpha) < c(ab)$. Note that $d(a), d(b) > 0$ and $c(a) \leq c(b)$, we have

$$\begin{aligned} c(\alpha) &< c(ab) \\ &< \frac{w(a)+w(b)}{d(a)+d(b)} \\ &< \frac{d(a)c(a)+d(b)c(b)}{d(a)+d(b)} \\ &\leq \frac{d(a)c(b)+d(b)c(b)}{d(a)+d(b)} \\ &= c(b) \end{aligned} \tag{1}$$

then

$$\begin{aligned} & L(ab\alpha) - L(a\alpha b) \\ = \; & L(b\alpha) - L(\alpha b) \\ = \; & d(b)[c(\alpha)d(\alpha)] - d(\alpha)[c(b)d(b)] \\ = \; & d(b)d(\alpha)[c(\alpha) - c(b)] \end{aligned}$$

By equation (1), it follows that $L(ab\alpha) < L(a\alpha b)$, contradicting the assumption.

Similarly, for the other case that $c(\alpha) \geq c(ab)$, we get that $L(\alpha ab) < L(a\alpha b)$, which also contradicts the assumption. □

So, if $c(a) \leq c(b)$ then we can shrink a and b into inseparable subtour ab. Based on Lemma 3.1, we claim that if an inseparable subtour has the maximum ratio cost among all others. It can glue with its parent.

Lemma 3.2 *If T'_k is inseparable subtour with maximum ratio cost among all others, It can glue with its parent or if T'_k is the root edge, then T'_k can be output directly.*

Proof. First from the Lemma 3.1, we can say that the inseparable subtour has maximum ratio cost must be the maximum child of its parent and can always be shrink with its parent. Second, if $T'k$ is a root edge and inseparable with maximum ratio cost. According to Lemma 3.1, no nonempty subsequence α can separate T'_k. Further T'_k is the root edge, so we can easily claim that all other nonempty subsequence will be traversed after T'_k is completed. □

ALGORITHM **TreeMLT**(T)

Input: A tree T.

Output: Minimum latency tour.

Data Structure: Consist each edge $e(u,v)$ and vertex v pair into one data node. Each node $E=(e,v)$ have the following data:

- d: The latency time of e and v.
- w: The weight of v.
- cr: The cost ratio of inseparable subtour form E.
- *parent*: The parent edges.
- *next*: The next node of inseparable subtour.

Initial: Build each edge and vertex pair. Let each $E_i.cr \leftarrow \frac{w(e)}{d(e)}$, and set all other variable by it definition.

Step 1: Build a Fibonacci heap F for all E_i by the key $E_i.cr$.

Step 2: If F is empty then stop the job else let e is the node returned from the extract-max from F.

Step 3: If e is the root edge than output the inseparable subtour starting from e following the $e.next$ link to get all member for output. Continue the Step 2.

Step 4: If e is not root edge, then we modify the parent's edge $e'.w \leftarrow e'.w + e.w$, $e'.d \leftarrow e'.d + e.d$ and re-compute the $e'.cr \leftarrow \frac{e'.w}{e'.d}$. Continue the step 2.

END OF **TreeMLT**

Figure 2: Algorithm for finding minimum latency tour in tree.

Theorem 3.3 *The minimum latency tour can be found in $O(n \log n)$ time for tree case.*

Proof. From the Lemma 3.1 and Lemma 3.2, we can propose an algorithm with $O(n \log n)$ time. The algorithm is shown in Figure 2. Note that Step 1 of the algorithm takes $O(n)$ time to build a Fibonacci heap structure. Clearly Step 2 will be execute $O(n)$ times. Because every time the Step 2 execute, one node been deleted. Each time Step 2 execute will perform an Extract-Max action. This action need $O(\log n)$ time. Further, Step 3 can be done in $O(n)$ time, too. The reason is every node will be output only once. Finally, Step 4 perform an increase key of Fibonacci heap. The key increasing action only cost $O(1)$. Base on the analysis above, we can prove that the minimum latency tour of tree case can be found in $O(n \log n)$ time.

4 Algorithm for Full k-ary Trees

We say that a tree T is a *full k-ary tree* if each internal node of T has exactly k children. Within the given tree T, we distinguish two kinds of vertices: let $I=\{i_1,i_2,\ldots,i_{|I|}\}$ be the set of internal nodes and $D=\{d_1,d_2,\ldots,d_{|D|}\}$ be the set of leaf nodes. In this section, we provide a linear time algorithm for full k-ary tree T where each internal edge has a constant weight and latency $w(I), d(I)$ and each leaf edge has $w(D), d(D)$.

First we consider the case that $c(I) \geq c(D)$. This case is trivial because we can easily traverse all internal node by any order and visit every leaf nodes sequentially. Clearly, it only need $O(n)$. If $c(I) < c(D)$, than we must to show two things, one is every subtree of full-ary tree is inseparable so we can propose a top-down algorithm. Another is if a subtree have more internal node, it ratio cost must small than subtree have less internal node. Form this two result, we can easily provide a linear time algorithm and prove its correctness.

Let $w_I = x, w_D = y, d(I) = u, d(D) = v$.

Lemma 4.1 *Given a full k-ary tree T with ratio cost c_I and c_D and $c_I < c_D$. Any sub-tree T_i' of full k-ary tree T is inseparable.*

Proof. Since T is a full k-ary tree ,every subtrees are defined recursively. If T_i has i internal nodes T_j has j internal nodes, then T_i has $ki-i+1$ leaf and T_j has $kj-j+1$ respectively. We want to prove that for any sub-tree T_i' is inseparable, then

$$\begin{aligned} c(T_i') &\geq \lim_{i\to\infty} \frac{(ki-i+1)y+ix}{(ki-i+1)v+iu} \\ &= \frac{ky+x-y}{kv+u-v} \end{aligned} \tag{2}$$

Since $c(I) < \frac{ky+x-y}{kv+u-v} < c(D)$, we can conclude that $c(I) < c(T_i')$ for all i. Let T_n' be a subtree of T, and the root of T_n' be α. Let T_n' have m inseparable unit $T_{a_1}', T_{a_2}' \ldots T_{a_m}'$, where T_{a_i}' denote a subtree with a_i internal nodes. Assume that if $i<j$ $c(T_{a_i}) > c(T_{a_j})$.

We prove the induction basis by shrinking the α with an inseparable unit T_{a_1}' with maximum ratio cost within $\{T_{a_1}', T_{a_2}', \ldots, T_{a_m}'\}$. After the shrinking process, we have

$$c(\alpha T_{a_1}') = \frac{(ka_1-a_1+1)y+(a_1+1)x}{(ka_1-a_1+1)v+(a_1+1)u} \leq \frac{ky+x-y}{kv+u-v} \tag{3}$$

Since $k \geq 2$, and $c(I) < c(D)$, we can easily prove that Equation(3) holds. Again, now we have $c(\alpha T_{a_1}') \leq \frac{ky+x-y}{kv+u-v}$ and $c(T_{a_j}') \geq \frac{ky+x-y}{kv+u-v} \forall 2 \leq j \leq m$; we can iterative shrinking the next subtrees. Let $A_m = a_1 + a_2 + \ldots + a_m$.Assume that the following equation is hold.

$$\begin{aligned} &c(\alpha T_{a_1}' \ldots T_{a_{m-2}}') \\ &= \frac{(kA_{m-2}-A_{m-2}+m-2)y+(A_{m-2}+1)x}{(kA_{m-2}-A_{m-2}+m-2)v+(A_{m-2}+1)u} \end{aligned}$$

$$\leq \frac{ky+x-y}{kv+u-v} \quad (4)$$

where $T'_{a_1}...T'_{a_{m-2}}$ has $m-2$ subtrees of T'_n.

By induction, we must show that after shrink $T'_{a_{m-1}}$ and $c(\alpha T'_{a_1}...T'_{a_{m-2}})$. The $c(\alpha T'_{a_1}...T'_{a_{m-1}}) \leq \frac{ky+x-y}{kv+u-v}$

$$\begin{aligned} & c(\alpha T'_{a_1}...T'_{a_{m-2}}T'_{a_{m-1}}) \\ &= \frac{(kA_{m-1} - A_{m-1} + m - 1)y + (A_{m-1}+1)x}{(kA_{m-1} - A_{m-1} + m - 1)v + (A_{m-1}+1)u} \\ &\leq \frac{ky+x-y}{kv+u-v} \end{aligned}$$

Because $c(T_{a_{m-1}}) > \frac{ky+x-y}{kv+u-v}$ and $k \geq 2$, we can prove that a root of any subtree can iterative shrink all subtree. Further, we can say any subtree of full k-ary tree is inseparable. □

Lemma 4.2 *Given a full k-ary tree T with ratio cost c_I and c_D and $c_I < c_D$. Denote the sub-tree of T with i internal nodes by T_i. It follows that $c(T_i) > c(T_j)$ if $i < j$.*

Proof. According to Lemma 4.1,the $c(T_i)$ can be evaluated reasonable with full subtree node. If $i < j$ then

$$\begin{aligned} & c(T_i) - c(T_j) \\ &= \frac{((ki-i+1)y + ix}{(ki-i+1)v + iu} - \frac{(kj-j+1)y + jx}{(ki-j+1)v + ju} \\ &= \frac{(yu-xv)(j-i)}{((ki-i+1)v+iu)((ki-j+1)v+ju)} \\ &> 0 \end{aligned}$$

Since $c_I < c_D$ and $i < j$, then $c(T_i) > c(T_j)$. Thus, if $i < j$ and $c_I < c_D$ then $c(T_i) > c(T_j)$. □

Theorem 4.3 *The minimum latency tour can be found in linear time, if G is full k-ary tree with constant ratio cost.*

Proof. According lemma 4.1, we can propose an algorithm with linear time. The algorithm is shown in Figure 3. Note that Step 1 of the algorithm takes $O(n)$ time to compute internal node number of each subtree. Clearly Step 2 and Step 3 can be done in $O(n)$ time. Step 4 is a depth first search that traverse each node and output it in $O(n)$ time [3].

5 Algorithm for k-Path Graphs

In this section, we will discuss some properties of the k-path graph and propose an $O(n \log k)$ time algorithm. If k is a constant, the problem is linear time solvable.

A graph $G(V, E)$ is called the k-path if G is a star graph which central vertex has degree k. To solve this kind of graph, we have some observations in k-path. First, after the starting point been visited, G split into k paths. Next, because we need not to compute the walk back cost, we can always choose a best subpath to visit and then choose another subpath without any extra cost. Those observations construct the skeleton of k-Path algorithm. To simplify the discussion below, we give some definitions about the k-path graph G. Let P be a path of G and $\rho = \langle \alpha_1, \alpha_2 \ldots, \alpha_d \rangle$ be a path partition of P. Where the α_i is the i-th path partition of ρ. By the definition in Section 2, each α has a ratio cost called $c(\alpha)$. We call a subpath $\alpha = \alpha_a \alpha_b$ is *Right-Skew* if and only if the ratio cost for the prefix α_a is always smaller than the ratio cost of the remaining subpath α_b. We also call a ρ is *Decreasing Right-Skew* if each α_i is right-skew and $c(\alpha_i) > c(\alpha_j)$ for all i, j and $i < j$.

ALGORITHM **kTreeMLT**(T)

Input: A full k-ary tree with constant internal node weight x, latency time u and leaf node weight y, latency time v where $\frac{x}{u} < \frac{y}{v}$.

Output: Minimum latency tour.

Data Structure: Each node of Tree have a value $V_i.internal$ indicate the internal node number of subtrees rooted by V_i, and a priority queue $V_i.queue$.

Initial: $V_i.internal$=0 and $V_i.queue$ is empty.

Step 1: Use a recursive function to compute internal nodes number of each sub-trees. And keep it in $V_i.internal$.

Step 2: Use a radix sort function to sort all node by $V_i.internal$.

Step 3: By the sorting result R, scan R form minimum to maximum and add itself into the priority queue of its parent.

Step 4: Use a depth-first search and use priority queue $V_i.queue$ of each node to choose next child to be visited. Output the chose node.

End of kTreeMLT

Figure 3: Algorithm for finding minimum latency tour in full k-ary tree.

Observation 5.1 *Let a, b, c be the real numbers and $c > \frac{a+b}{2}$. If $a > b$ then $\frac{c+a}{2} > b$*

Here we first show that all single path P can be partition into Decreasing Right-Skew.

Lemma 5.2 *Let P be a path, there exist a partition ρ is Decreasing Right-Skew.*

Proof. Let $P = v_1, v_2 \ldots v_n$. The valid partition can be found by the following step. First, for all i from 1 to n, compute the $c(v_1 \ldots v_i)$. Assume $c(v_1 \ldots v_j)$ be the maximum ratio cost among all i. Let $\alpha_1 = v_1 \ldots v_j$. Next, eliminate the prefix $v_1 \ldots v_j$ from P and

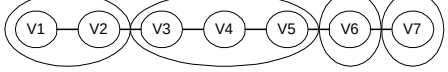


Figure 4: The example of the path partition

ALGORITHM **pathPartition**(T)

Input: A path $P = v_1, v_2 \ldots, v_n$.

Output: A path partition.

Initial: $i \leftarrow n-1; j \leftarrow n; V \leftarrow v_n$.

Step 1: Set $v_j \leftarrow V$ and set v_i as the parent of v_j.

Step 2: if v_j has no parent and $c(v_j) \leq c(v_{j+1})$ then shrink v_{j+1} to v_j.

Step 3: if v_j has no parent then stop the algorithm and return the partition.

Step 4: If $c(v_i) \leq c(v_j)$ then shrink v_j to v_i and set $V \leftarrow v_i v_j$.

Step 5: If $c(v_i) > c(v_j)$ and $c(v_j) \leq c(v_{j+1})$ shrink v_{j+1} to v_j and set $V \leftarrow v_j v_{j+1}$.

Step 6: If $c(v_i) > c(v_j)$ and $c(v_j) > c(v_{j+1})$ set $V \leftarrow v_i$.

Step 7: Goto Step 1.

END OF **pathPartition**

Figure 5: Algorithm to partition path

repeat the first step to find the next sub-path α. In this phrase we can easily prove that $c(\alpha_i) > c(\alpha_j)$ because if $c(\alpha_i) \leq c(\alpha_j)$ then $c(\alpha_i\alpha_j) > c(\alpha i)$ (Observation 5.1) contradict the first step.

Further, because $c(\alpha_i) > c(\alpha_j)$ for all i, j, where $i < j$ has been proved. From the Observation 5.1, if α_1 is not the right-skew then there exist a prefix of α_1 has larger ratio cost. Contradict the assumption of first step. The α_1 is right-skew. If α_{i+1} is not the right-skew then α_i may concatenate the prefix of α_{i+1} to enlarge the ratio cost of α_i. Contradict the assumption of first step. The α_{i+1} is right-skew. Thus, we can prove that given a path P, we always can partition it into Decreasing Right-Skew.

Algorithm to prove Lemma 5.2 costs $O(n^2)$ times. In Algorithm 5, we will use a bottom-up manner to reduce the time complexity to linear time. □

Lemma 5.3 *Algorithm 5 is correct and the time complexity is $O(n)$.*

Proof. Consider the Algorithm 5. The V denote the current working pointer initially at the end vertex of P. Every times from Step 1 to Step 6 will cause the V move to v_1 at least one vertex or shrink at least one vertex. After the linear step, the path will be partition into Decreasing right-skew. Because each step only cost constant time. The time complexity of Algorithm 5 is linear. To prove the correctness of Algorithm 5, assume the result path partition is $\rho = \langle \alpha_1, \alpha_2 \ldots, \alpha_d \rangle$ and there exist $c(\alpha_i) < c(\alpha_j), i < j$ or α_i is not right-skew. Base on the shrink condition in Step 2,4,5. Each α_i must be right-skew. If $c(\alpha_i) < c(\alpha_j), i < j$ then $c(\alpha_i) > c(\alpha_j)$ when $V = \alpha_j$. If $c(\alpha_j) \leq c(\alpha_{j+1})$ then α_j will shrink the α_{j+1} into α_j and set the V as the new α_j and repeat the step until the $c(\alpha_j) > c(\alpha_i)$ or $c(\alpha_j) > c(\alpha_{j+1})$. If $c(\alpha_j) > c(\alpha_i)$ contradict the assumption. If $c(\alpha_j) > c(\alpha_{j+1})$ the V will move to α_i and then $c(\alpha_i) > c(\alpha_j)$ also contradict the assumption. □

Lemma 5.4 *Given k paths. We denote the partition as $\rho_i = \langle \alpha_{i1}, \alpha_{i2} \ldots, \alpha_{id} \rangle$. If $c(\alpha_{i1}) > c(\alpha_{j1})$ then in optimal solution α_{i1} must placed in front of α_{j1}.*

Proof. Assume in optimal solution α_{j1} placed in front of α_{i1}. We compare the case which α_{i1} be placed in front of α_{j1}.

$$\begin{aligned} &\ell_j \ell_i c(\alpha_{i1}) - \ell_i \ell_j c(\alpha_{j1}) \\ &= c(\alpha_{i1}) - c(\alpha_{j1}) \\ &> 0 \end{aligned}$$

Since $c(\alpha_{i1}) > c(\alpha_{j1})$, we can change the position between α_{i1} and α_{j1} to get better solution. Contradiction the original assumption.

Theorem 5.5 *If G is a k-path graph, the minimum latency tour can be done in $O(n \log k)$ time.*

Proof. Base on Lemma 5.3, we can partition all the k path into several Decreasing Right-Skews in linear time. We denote as $\rho_i = \langle \alpha_{i1}, \alpha_{i2} \ldots, \alpha_{id} \rangle$. Then according to the Lemma 5.4, we can initial build a heap from $\alpha_{i1}, \ldots, \alpha_{k1}$ node. And iterative extract the maximum right-skew α_{ij} form heap and insert the $\alpha_{i(j+1)}$ into heap. After the iterative we get a sequence of walk tour. From the Lemma 5.4, we prove the output walk tour is a minimum latency tour for G. □

6 Lower Bound

In this section, we prove that $\Omega(n \log n)$ and $\Omega(n)$ are lower bounds for general tree , full k-ary tree with constant weight, and k-path graphs respective.

Theorem 6.1 *Lower bound of the variation minimum latency problem is $\Omega(n \log n)$ even for binary tree.*

Proof. Given a set of nonnegative real values $\{N_0, N_1, \ldots, N_n\}$, we construct a binary tree illustrated by Figure 6. Note that all internal nodes and one leaf node have a very large weight, say ∞. Because $w(D_{n+1}) = \infty$, we first shrink $\{I_0, I_1, \ldots, I_n, D_{n+1}\}$ into one inseparable supper node. After this, a comparison sorting is required for determining the output sequence

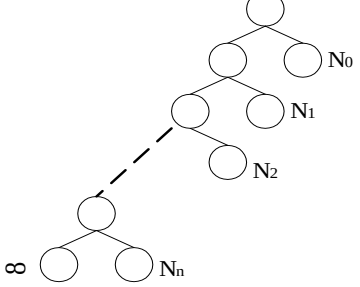


Figure 6: The worst case for general tree graph

of $\{N_0, N_1, \ldots, N_n\}$. Note that $\Omega(n \log n)$ time is the lower bound for comparison sorting. Thus, we can say the lower bound of the minimum access latency problem of general tree is $\Omega(n \log n)$ time. □

Theorem 6.2 *Algorithm kTreeMLT is optimal in terms of time complexity.*

Proof. Obviously the lower bound is $\Omega(n)$ since the size of the output set is exactly the input size n for the full k-ary tree. It follows that algorithm kTreeMLT is optimal for the full k-ary tree. □

7 Concluding Remarks

In this paper, we define a variation of original minimum latency problem. In the new problem, the weight and latency cost of vertices been defined additionally and we ignore the walk back cost to make the problem more realistic in some real problem. This variation is only defined in trees and some simpler graphs, like full k-ary tree. In this paper, we propose three versions of algorithms for different graphs. Let n denote number of vertices in the given graph. We finds an $O(n \log n)$ time algorithm for general trees with weighted vertices and weighted edges, and an $O(n)$ time algorithm for full k-ary tree with constant ratio costs of each vertex. For general case, previous results [1] show that $O(n \log n)$ is optimal, but our algorithm seems more intuitive and easily to implement. Two topics concerning the minimum latency problem will be addressed in the future. One is the problem in dags, we believe that is NP or $NP - Hard$, another is to consider an efficient on line algorithm to immediately reflect the weight change.

For the case of direct acyclic graph, we didn't find an polynomial time algorithm yet. It will be interesting to know whether the time complexity of the algorithm on dags can be solved in polynomial time. Another interesting problem is about the starting point. In our definition, the starting point is given and never change. If the starting point is online changed. Can we get the new minimum latency tour efficient? If the starting point is undefined, where the best starting point is? Those problems still remain open. Of course, the ultimate open problem is whether the variant of minimum latency problem can be efficiently solved in dags.

References

[1] D. Adolphson and T. C. Hu. Optimal linear ordering. *SIAM J. Appl. Math.*, 25:403–423, 1973.

[2] A. Blum, P. Chalasani, D. Coppersmith, B. Pulleyblank, P. Raghavan, and M. Sudan. The minimum latency problem. *Proc. 26th Annu. ACM Sympos. Theory Comput.*, pages 163–171, 1994.

[3] T. Corman, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[4] T. Imielinski, S. Viswanathan, and B.R. Badrinath. Power efficient filtering of data on air. *4th International Conference on Extending Database Technoloy*, pages 245–258, March 1994.

[5] T. Imielinski, S. Viswanathan, and B.R. Badrinath. Data on air: Organization and access. *IEEE Trans. on Knowledge and Data Engineering*, 9(3):353–372, May 1997.

[6] Shou-Chih Lo and Arbee L.P. Chen. Index and data allocation in multiple broadcast channels. *IEEE Interantion Conference on Data Engineering 2000*, pages 293–302, 2000.

[7] N. Shivakumar and S. Venkatasubramanian. Energy-efficient indexing for information dissemination in wireless systems. *ACM, Journal of Wireless and Nomadic Application*, 1996.

[8] M. Chen P. Yu and K. Wu. Indexed sequential data broadcasting in wireless mobile computing. *IEEE Interantional Conference on Distributed Computing Systems*, pages 124–131, 1997.