Minimum Back-Walk-Free Latency Problem (Extended Abstract)

Yaw-Ling Lin*

Department of Comput. Sci. and Info Management, Providence University,
200 Chung Chi Road, Shalu, Taichung County, Taiwan 433, R.O.C.
e-mail: yllin@pu.edu.tw

Abstract. Consider a graph with n nodes $V = \{1, 2, \ldots, n\}$ and let $d(i, j)$ denote the distance between nodes i, j. Given a permutation π on $\{1, 2, \ldots, n\}$ such that $\pi(1) = 1$, the back-walk-free latency from node 1 to node j is defined by $\ell_\pi(j) = \ell_\pi(j-1) + \min\{d(\pi(k), \pi(j)) \mid 1 \leq k \leq j-1\}$. Note that $\ell_\pi(1) = d(1, 1) = 0$. Each vertex i is associated with a nonnegative weight $w(i)$. The (weighted) *minimum back-walk-free latency problem* (MBLP) is to find a permutation π such that the total back-walk-free latency $\sum_{i=1}^{n} w(i)\ell_\pi(i)$ is minimized.

In this paper, we show an $O(n \log n)$ time algorithm when the given graph is a tree. For a k-path trees, we derive an $O(n \log k)$ time algorithm; the algorithm is shown to be optimal in term of time complexity on any comparison based computational model. Further, we show that the optimal tour on weighted paths can be found in $O(n)$ time.

No previous hardness results were known for MBLP on general graphs. Here we settle the problem by showing that MBLP is NP-complete even when the given graph is a direct acyclic graph whose vertex weights are either 0 or 1.

1 Introduction

The *minimum latency problem* (MLP) [3], also known as the traveling repairman problem [2], or the deliveryman problem [10], is to find a minimum latency tour on a graph. Usually, the starting node of the tour is given and the goal is to minimize the sum of the arrival times at the other nodes. The arrival time is the distance traversed before reaching that node. Consider a distance matrix of a graph with n nodes, where $d(i, j)$ indicates the distance between nodes i, j. The minimum latency tour is a permutation π of vertices such that $\pi(1) = 1$, and the term

$$\sum_{i=2}^{n} \sum_{j=1}^{i-1} d(\pi(j), \pi(j+1))$$

is minimized. Note that $\ell_\pi(i) = \sum_{j=1}^{i-1} d(\pi(j), \pi(j+1)) = \ell_\pi(i-1) + d(\pi(i-1), \pi(i))$ is the traveling latency from node 1 to node i. In other words, the minimum latency tour is the permutation π that minimizes the total latency

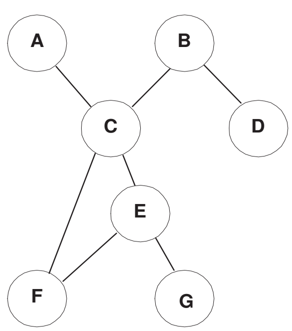
* The work is supported in part by the National Science Council, Taiwan, R.O.C, grant NSC-89-2218-E-126-004.

$\sum_{i=2}^{n} \ell_\pi(i)$. The problem arises in a number of applications in the operations research literatures, on-line search problems, and disk-head scheduling [2, 10, 12].

Despite its apparent similarity with the traveling salesperson problem (TSP), the MLP possesses a variety of aspects very different from the TSP. Small changes in the structure of a metric space can cause highly non-local changes in the structure of MLP. In fact, the problem is conjectured by D. West [8] to be NP-hard for edge-weighted trees, and even caterpillars, i.e., paths with edges sticking out. By contrast, the TSP can be optimally solved on a tree. However, both problems are known to be NP-hard when the points lie in a general metric space or even the (Euclidean) plane. Polynomial time algorithms for the optimal solutions are only known for the cases when the graph is a path [2, 13], an edge-unweighted tree [10], a tree with diameter 3 [3], or a tree with a constant number of leaves [8].

The minimum latency problem can be generalized to the vertex-weighted case. The goal is to minimize the sum of latencies of each vertex multiplied by its weight. In other words, when each node i is given a real weight $w(i)$, the goal is to find a permutation π such that the term $\sum_{i=2}^{n} \ell_\pi(i) w(\pi(i))$ is minimized. To find an optimal tour that minimizes the vertex-weighted latency on a path, an $O(n^2)$ time algorithm can be easily deduced from the results in [2, 3]. Wu [13] shows that finding the best starting vertex so as the minimum vertex-weighted latencies is minimized can also be done in $O(n^2)$ time.

In this paper, we discuss a variant of the vertex-weighted MLP problem that ignores the back-walking costs. Given a starting node 1 and a permutation π, the *back-walk-free travelling latency* from node 1 to node j is now defined by $\ell_\pi(j) = \ell_\pi(j-1) + \min\{d(\pi(k), \pi(j)) \mid 1 \le k \le j-1\}$. Given the distance matrix of a graph with n nodes, where $d(i,j)$ indicates the distance between nodes i, j, the (weighted) *minimum back-walk-free latency problem* (MBLP) is to find a permutation π that minimizes the total back-walk-free latency $\sum_{i=1}^{n} w(i)\ell_\pi(i)$. Figure 1 gives an instance of MBLP and its minimum latency tour.



1. A is the starting vertex. Weights of vertices:

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

2. Distances of charted edges are 1's.

3. Distances of uncharted edges are 100's.

Optimal tour: <A,C,E,G,F,B,D> with latency:
0*30+1*45+2*42+3*101+4*65+5*0+6*50
= 992

Fig. 1. The minimum back-walk-free latency tour of a graph.

Several variations of this problem focused on index and data allocation in a single broadcast channel using conventional techniques [6, 7]. The issue of multiple broadcast channels using prune strategies and heuristic answers from single broadcast channel is addressed in [11, 9].

When the given graph is a tree, Adolphson and Hu [1] gave an $O(n \log n)$ time algorithm; however, the technique used is difficult to be used for special cases of trees, where more efficient algorithms are possible. Thus, in this paper, we propose another $O(n \log n)$ time algorithm for the tree case. And, by exploiting the techniques we used on the algorithm, we derive an $O(n \log k)$ time algorithm for k-path trees. Further, we show that the optimal tour on weighted paths can be found in $O(n)$ time.

No previous hardness results were known for MBLP on general graphs. Here we settle the problem by showing that MBLP is NP-complete even when the given graph is a direct acyclic graph whose vertex weights are either 0 or 1, while each dag edge is associated with an unit distance, and each non-dag edge with a large distance.

2 Basic Notations and Properties

In this paper, we consider a rooted tree $T = (V, E)$ such that the vertex set $V = \{1, 2, \ldots, n\}$ and the root is node 1. Associate each vertex i with a nonnegative weight $w(i)$ and a nonnegative distance $d(i)$. Intuitively, $d(i, j) = d(j)$ if i is the parent node of j; otherwise, $d(i, j) = \infty$.

A permutation of nodes on subset of V is called a *subtour* of T. That is, a sequence $A = \langle a_1, a_2, \ldots, a_k \rangle$ is a subtour of T if $\{a_1, a_2, \ldots, a_k\} \subset V$ and $a_i \neq a_j$ whenever $i \neq j$. Denote $w(A) = \sum_{i=1}^{k} w(a_i)$; $d(A) = \sum_{i=1}^{k} d(a_i)$. The *cost ratio* of A is defined by $\rho(A) = w(A)/d(A)$.

Definition 1. *The* weighted latency *of a subtour $A = \langle a_1, a_2, \ldots, a_k \rangle$ is defined by $L(A) = \sum_{i=1}^{k} (w(a_i) \cdot \sum_{j=1}^{i} d(a_j))$.*

A subtour spanning every vertex of V, which defines a permutation on V, is called a *tour* on T. A tour π is *feasible* if the corresponding (weighted) latency $\ell(\pi) = \sum_{i=1}^{n} w(i)\ell_\pi(i) < \infty$. By assigning zero to $d(1)$, we have the following observation.

Proposition 1. *Let $d(1) = 0$. Given a feasible tour A with the corresponding permutation π, it follows that $\ell(\pi) = \sum_{i=1}^{n} w(i)\ell_\pi(i) = L(A)$.*

Let $A = \langle a_1, a_2, \ldots, a_i \rangle$ and $B = \langle b_1, b_2, \ldots, b_j \rangle$ be two disjoint subtour of T. The *concatenation* of A and B, denoted by $AB = \langle a_1, a_2, \ldots, a_i, b_1, b_2, \ldots, b_j \rangle$ is also a subtour of T.

Lemma 1 (Recursion formula of $L(\cdot)$). *Let A, B be two disjoint subtours of T. We have: $L(AB) = L(A) + L(B) + d(A)w(B) = L(A) + L(B) + d(A)d(B)\rho(B)$.*

Proof. The proof appears in the complete paper. □

We immediately have the following observation:

Proposition 2. *Let A, B be two disjoint subtours of T. It follows that $L(AB) \leq L(BA)$ if and only if $\rho(A) \geq \rho(B)$.*

Intuitively, it suggests us to place the heaviest vertex at the front of visited tour as early as possible.

Definition 2. *We say a subtour A of T is* atomic *if there exists a minimum (back-walk-free) latency tour P on T containing A contiguously. That is, $P = BAC$, such that B, C are two (possibly empty) subtours.*

Note that each vertex i defines an atomic subtour $\langle i \rangle$ with cost ratio $\rho(i) = w(i)/d(i)$.

3 Algorithm for General Trees

In this section we discuss properties of MBLP on general trees. Using these properties, an $O(n \log n)$ time algorithm for tree can be naturally deduced.

The idea here is to repeatedly identify the atomic parent-child pairs in the tree T, and *glue* them together until the final minimum latency tour is found. Initially, each node is, by definition, an atomic subtour of T; however, during the process, the original tree node is generalized to a super node, which contains atomic subtour of the T. Note that the topologically transformed tree structure reflects several interesting properties of T. For example, the subtour resides within each super node induces a (connected) subtree of T.

Consider two atomic subtours of T, namely A and B, and let T_A (T_B) denote the subtree of T induced by A (B). If there exists a node a of A and a node b of B such that a is a parent of b, then A is thus called a *parent* of B; conversely, B is a *child* of A.

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

Lemma 2. *Let A be an atomic subtour of T. The descendants of A consists of k mutually disjoint atomic subtours of T, namely, $A_1, A_2, \ldots, A_k$. Let $\rho(A_c) = \max\{\rho(A_j) \mid 1 \leq i \leq k\}$. If A_c is a child of A, and $\rho(A_c) \geq \rho(A)$, then AA_c is an atomic subtour of T.*

Proof. The proof appears in the complete paper. □

Corollary 1. *If A is an atomic subtour of T with largest ratio cost comparing to every other atomic subtours of T. Either it can be glued with its parent; otherwise, A can be appended to the output queue if A is a root.*

Theorem 1. *The minimum back-walk-free latency tour can be found in $O(n \log n)$ time for vertex-weighted trees.*

Proof. We propose an $O(n \log n)$ time algorithm, MBLP-TREE(T), shown in Figure 2. The algorithm essentially keeps clustering two atomic subtours into one. Each time the number of vertices is reduced by one. The correctness of the algorithm is easily verified by Lemma 2 and Corollary 1.

MBLP-Tree(T)
Input: A rooted tree T with weighted vertices $V = \{1, 2, \ldots, n\}$; node 1 is root.
Each vertex i has a nonnegative weight $w(i)$ and a nonnegative distance $d(i)$.
Output: A tour with minimum weighted back-walk-free latency.
1 **for** $i \leftarrow 2$ **to** n **do** $\quad\triangleright$ Initializes the Fibonacci Queue Q
2 **new** v; $w[v] \leftarrow w(i); d[v] \leftarrow d(i); key[v] = w[v]/d[v]$; $list[v] \leftarrow \langle i \rangle$
3 EnQueue($Q, key[v]$) ; Make-Set(v)
4 **for** each $uv \in E(T)$ **do** $p[v] \leftarrow u$ $\quad\triangleright$ u is a parent of v
5 **while** $Q \neq \emptyset$ **do**
6 $key[v] \leftarrow$ Extract-Max(Q); $\quad\triangleright$ Pick the maximal cost ratio.
7 $u \leftarrow p[$Find-Set$(v)]; w[u] \leftarrow w[u] + w[v]; d[u] \leftarrow d[u] + d[v]$
8 Union$(u, v); p[v] \leftarrow p[u]$ $\quad\triangleright$ Efficient clustering
9 $key[u] = w[u]/d[u]; list[u] = list[u] \circ list[v]$; Increase-Key($Q, key[u]$)
10 **return** the output list of the root contains the minimum latency tour.

Fig. 2. Algorithm for finding minimum back-walk-free latency tour in tree.

Note that Step 1 to Step 4 of the algorithm takes $O(n)$ time to build a Fibonacci heap Q. Note that Q is a maximum-finding queue instead of the regular minimum-finding queue. Step 5 to Step 9 of MBLP-Tree(T) is to pick the maximum cost ratio node and glue it to its parent. The while loop executes exactly $n - 2$ times. The Extract-Max operation of Step 6 takes amortized $O(\log n)$ time. Thus totally $O(n \log n)$ time will be spent on Step 6. On the other hand, the Increase-Key operations performed in Step 9 takes amortized $O(1)$ time; thus totally $O(n)$ time is spent on Step 9.

What is interesting in the algorithm is that we use the disjoint set data structure [4] in clustering the nodes contained within each subtour. A naive approach will need to locate all the pointers of the children of v, and let their parent pointer changed to the new parent u in Step 8 of the algorithm. Totally there are potentially $O(n^2)$ changes. Now with the help of the disjoint set data structure, Step 7 and Step 8 of the algorithm totally spends $n\alpha(n)$ time, where $\alpha(n)$ is the inverse of Ackermann's function. We thus conclude that the algorithm totally spends $O(n \log n)$ time. □

4 Algorithm for k-Path Trees

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

A tree T is called a *k-path tree* if there is one vertex of T with degree k and all other vertices have degrees either 1 or 2. Note that a simple path is a 2-path. A *rooted k-path*, is the k-path rooted at the degree k vertex.

Let $P = \langle a_1, a_2 \ldots, a_m \rangle$ be a path of T, where vertices are listed from root to the leaf. It follows that any feasible subtour on P consists of consecutive vertices on P. Let $A_1, A_2, \ldots, A_k$ be disjoint subtours of P such that $\cup_{i=1}^{k} A_k = P$;

further, $P = A_1 A_2 \cdots A_k$. These A_i's is called a *path partition* of P; A_i is the ith path partition of P. Recall the definition in Section 2, the cost ratio of $A_i$is defined by $\rho(A_i) = w(A_i)/d(A_i)$.

Definition 3. *We call a subtour $A = \langle a_1, a_2 \ldots, a_m \rangle$* right-skew *if and only if the cost ratio of any prefix $\langle a_1, a_2 \ldots, a_j \rangle$ is always smaller than or equal to the cost ratio of the remaining subpath $\langle a_{j+1}, a_2 \ldots, a_m \rangle$. A path partition $P = A_1 A_2 \cdots A_k$ is* decreasingly right-skew *if each A_i is right-skew and $\rho(A_i) > \rho(A_j)$ whenever $i < j$.*

Proposition 3. *Let A, B be two subtour of T with $\rho(A) \geq \rho(B)$. It follows that $\rho(A) \geq \rho(AB) \geq \rho(B)$.*

Lemma 3. *Let A, B be two right-skew subtour of T with $\rho(A) \leq \rho(B)$. It follows that AB is also right-skew.*

Proof. The proof appears in the complete paper. □

Here we first show that all single path P can be partition into decreasingly right-skew.

Lemma 4. *Any path can be uniquely partitioned into a decreasingly right-skew partition.*

Proof. Let $P = \langle v_1, v_2 \ldots v_n \rangle$. The argument obviously follows if $n = 1$. By induction, assume that a path $Q = A_1 A_2 \cdots A_k$, $|Q| = n$ is decreasingly right-skew partitioned. Now consider a path $P = \langle Q, a \rangle$.

The Lemma is proven if $\rho(a) < \rho(A_k)$. Otherwise, we find the largest i such that $\rho(A_i A_{i+1} \cdots A_k a) < \rho(A_{i-1})$; let $i = 1$ if such i can not be found. It suffices to show that $\langle A_i A_{i+1} \cdots A_k a \rangle$ is right-skew, and this can be done by observing that the single segment $\langle a \rangle$ is right-skew and by applying Lemma 3 repeatedly to the segments $A_i, A_{i+1}, \ldots, A_k, \langle a \rangle$ from right to left. That is, $\langle A_k a \rangle$ is right-skew because $\rho(A_k) \leq \rho(a)$, $\langle A_{k-1} A_k a \rangle$ is right-skew because $\rho(A_{k-1}) \leq \rho(A_k a)$, *etc.* Clearly, the partition is unique because other choices of i would not result in a decreasingly right-skew partition of A. □

Lemma 5. *Partitioning a path into a decreasingly right-skew partition can be done in $O(n)$.*

Proof. Consider the algorithm PATH-PARTI that produces a list of right-skew sequences in Figure 3. Note that i is the current working pointer scanning the vertices of P from right to left. Further, the pair $(i, p[i])$ represents a subsequence of P that is always right-skew throughout the entire algorithm. The observation is justified by Lemma 3, that two increasingly right-skew subtours can be grouped into one. Note that the condition checking and grouping is done by Step 3 to Step 6 of the algorithm.

The correctness of the algorithm follows from the fact that, after the execution of Step 1 to Step 6, each subsequence is right-skew, and the partition is increasing. Further, by Lemma 4, the partition must be unique.

PATH-PARTI(P)
Input: A path $P = \langle a_1, a_2, \ldots, a_n \rangle$
Output: A decreasingly right-skew partition of P.
1 **for** $i \leftarrow n$ **downto** 1 **do**
2 $\quad p[i] \leftarrow i; w[i] \leftarrow w(a_i); d[i] \leftarrow d(a_i);$ $\quad \triangleright$ Each a_i alone is right-skew.
3 $\quad$ **while** $(p[i] < n)$ and $(w[i]/d[i] \leq w[p[i]+1]/d[p[i]+1])$ **do**
4 $\qquad p[i] \leftarrow p[p[i]+1]$
5 $\qquad w[i] \leftarrow w[i] + w[p[i]+1]$
6 $\qquad d[i] \leftarrow d[i] + d[p[i]+1]$
7 $i \leftarrow 1$
8 **while** $i \leq n$ **do** $\quad \triangleright$ Reports (i, i) as a right-skew subtour $\langle a_i, \ldots, a_j \rangle$.
9 $\quad$ OUTPUT $(i, p[i]); i \leftarrow p[i] + 1$

Fig. 3. Partitioning a path into decreasingly right-skew in $O(n)$ time.

The $O(n)$ time complexity is justified by the amortized analysis. Note that total operations of the algorithm is clearly bounded by $O(n)$ except for the while loop body of Step 3 to Step 6. We define the potential of P to be k_i, the number of right-skew subsequences in P after the ith iteration of the for loop (Step 1 to Step 6.). Let us compute the amortized cost of operations done by Step 3 to Step 6. Let the actual cost of the operation is $c_i + 1$ in advancing c_i times of the $p[i]$ pointer. It follows that the potential difference is

$$\Phi(P_i) - \Phi(P_{i-1}) = (1 + k_{i-1} - c_i) - k_{i-1};$$

The amortized cost is therefore $\hat{c}_i = c_i + \Phi(P_i) - \Phi(P_{i-1}) = 1$ □

Let T be a rooted k-path. After deleting the root, T is decomposed into k paths: $P_1, \ldots, P_n$. For each P_i, we perform the PATH-PARTI(P_i) resulting $P_i = \langle A_1 \cdots A_j \rangle$ such that the partition $\langle A_1 \cdots A_j \rangle$ is decreasingly right-skew. It follows that each A_i is an atomic subtour in T.

Lemma 6. *A feasible right-skew subtour of a k-path, T, containing only vertices of one of its descendant path, is atomic in T.*

Proof. The proof appears in the complete paper. □

Theorem 2. *Finding the minimum back-walk-free latency tour in k-path can be done in $O(n \log k)$ time.*

Proof. We propose an $O(n \log k)$ time algorithm, MBLP-k-TREE(T), shown in Figure 4. The correctness of the algorithm follows from the fact that, The right-skew subsequence of each k-path is atomic in T as show by Lemma 6. Further, the algorithm reflects the spirit of MBLP-TREE(T) in extracting the maximal cost ratio and output.

The time complexity of the algorithm is easily shown to be $O(n \log k)$ by the fact that the size of the priority queue Q is at most k while there are totally $O(n)$ operations performed on Q. □

```
MBLP-k-PATH(T)
Input: A rooted k-path T.
Output: A tour with minimum weighted back-walk-free latency.
Decompose T into k paths: P_1, ..., P_k.
for i ← 1 to k do
    list[i] ← PATH-PARTI(P_i)        ▷ Right-skew partition
    ENQUEUE(Q, EXTRACT(list[i]))        ▷ Put the first element into priority queue Q.
while Q ≠ ∅ do
     seq[i] ← EXTRACT-MAX(Q);        ▷ Pick the maximal cost ratio.
     ENQUEUE(Q, EXTRACT(list[i]))
     Append seq[i] to the output list.
return the output list.
```

Fig. 4. Algorithm for finding minimum back-walk-free latency tour in k-path.

Note that the output of the MBLP-TREE(T) can be viewed as a length $O(n)$ (multiple) permutation of the set $\{1, 2, \ldots, k\}$. Each output number represents the currently top vertex of one of the k paths being visited. Thus the number of all different possible output orderings can be as large as k^n. It follows that the solution space of the algorithm is therefore of the order $O(k^n)$. Since a comparison-based algorithm essentially searches the solution space in a binary (or ternary) manner, the underlying decision tree must have a depth of $\Omega(n \log k)$ by using simple counting arguments. It follows that

Proposition 4. *Any comparison-based algorithm for solving the MBLP on k-path spends $\Omega(n \log k)$ time in average.*

Corollary 2. *The algorithm* MBLP-k-PATH *spends $\Theta(n \log k)$ time, in average, in finding the minimum back-walk-free latency tour.*

Corollary 3. *Finding the minimum back-walk-free latency tour in a path (with any arbitrary starting vertex) can be done in $O(n)$ time.*

5 MBLP is NP-complete

Despite its apparent simplicity comparing to the original MLP, here we show that MBLP is NP-complete even for a *directed acyclic graph* (usually briefed as dag) by reducing the 3-SAT problem to MBLP on a dag.

Theorem 3. *The decision version of MBLP for directed acyclic graphs is NP-complete.*

Proof. The decision version of MBLP is clearly in NP. Here we show that it is NP-hard by a reduction from 3-SAT [5] to MBLP.

Given an instance of 3-SAT, S, with literals $L = \{x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n\}$ and clauses $C = \{c_1, \ldots, c_k\}$, we will construct a directed acyclic graph $G = (V, E)$

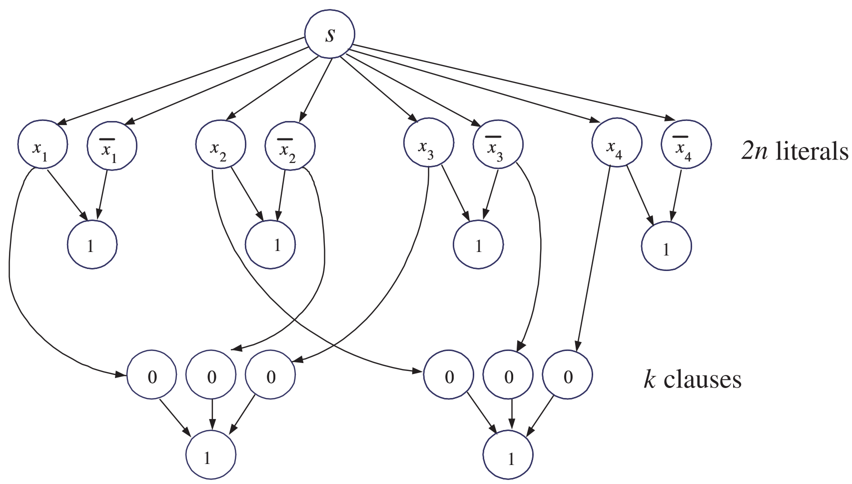


Fig. 5. Reducing the 3-SAT instance $(x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee \overline{x}_3 \vee x_4)$ into a dag.

such that G has a back-walk-free tour with latency $(n+k)(n+k+1)$ if and only if S is satisfiable. The reduction is illustrated at Figure 5.

The constructed dag consists of one starting vertex s, and other three kinds of vertices: $2n$ *literal vertices* $V_L = \{x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n\}$, $3k$ *clause vertices* $V_C = \cup_{i=1}^{k}\{c_{i1}, c_{i2}, c_{i3}\}$, and $n+k$ *sink vertices* $V_t = \{t_1, \ldots, t_n, t'_1, \ldots, t'_k\}$. The weights of vertices is defined by $w(v) = 1$ if $v \in V_t$; otherwise, $w(v) = 0$. The edge set of G consists of $4n$ *literal edges*, $E_L = \{(s,a) \mid a \in V_L\} \cup \{(x_i, t_i), (\overline{x}_i, t_i) \mid 1 \le i \le n\}$, k *clause edges*, $E_C = \cup_{i=1}^{k}\{(c_{i1}, t'_i), (c_{i2}, t'_i), (c_{i3}, t'_i)\}$, and finally the k *cross edges*, $E_X = \cup_{i=1}^{k}\{(a_{i1}, c_{i1}), (a_{i2}, c_{i2}), (a_{i3}, c_{i3})\}$ where $a_{i1}, a_{i2}, a_{i3} \in V_L$ are 3 vertices of the ith clause c_i. At last, we say that the distance of each (directed) edge is 1, and the distance of each unstated edge is $+\infty$.

The idea is that a low latency tour attempts to reach as many sink vertices as possible in the earliest manner, which constitutes the truth assignment of the given 3-SAT instance if it is satisfiable. The rest of the detailed arguments appears in the complete paper. □

6 Concluding Remarks

In this paper, a variation of the minimum latency problem, so called the minimum back-walk-free latency problem (MBLP) is considered. Several interesting properties of MBLP on trees are discussed, and we propose a simple and efficient $O(n \log n)$ time algorithm for general weighted tree, as well as an $O(n \log k)$ time algorithm for k-path trees. Further, we show that the optimal tour on weighted paths can be found in $O(n)$ time.

No previous hardness results were known for MBLP on general graphs. Here we settle the problem by showing that MBLP is NP-complete even when the given graph is a direct acyclic graph whose vertex weights are either 0 or 1, while each dag edge is associate with the same distance.

An interesting future work concerning about MBLP is to find a *good* starting point. In the original problem setting, the starting point is given. How fast can we find the best origin that minimizes the MBLP tour? What happens if the starting point is online changed? Can we get the new tour in a more efficient way?

References

1. D. Adolphson and T. C. Hu. Optimal linear ordering. *SIAM J. Appl. Math.*, 25:403–423, 1973.
2. F. Afrati, S. Cosmadakis, C. Papadimitriou, G. Papageorgiou, and N. Papakostantino. The complexity of the traveling repairman problem. *Informatique Theorique et Applications (Theoretical Informatics and Applications)*, 20(1):79–87, 1986.
3. A. Blum, P. Chalasani, D. Coppersmith, B. Pulleyblank, P. Raghavan, and M. Sudan. The minimum latency problem. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing*, pages 163–171, Montréal, Québec, Canada, 23–25 May 1994.
4. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.
5. M.R. Garey and D.S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.
6. 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.
7. 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.
8. E. Koutsoupias, C. Papadimitriou, and M. Yannakakis. Searching a fixed graph. In *Proc. 23rd Colloquium on Automata, Languages and Programming*, LNCS 1099, pages 280–289. Springer-Verlag, 1996.
9. 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.
10. E. Minieka. The delivery man problem on a tree network. *Annals of Operations Research*, 18(1–4):261–266, February 1989.
11. N. Shivakumar and S. Venkatasubramanian. Energy-efficient indexing for information dissemination in wireless systems. *ACM, Journal of Wireless and Nomadic Application*, 1996.
12. Tsitsiklis. Special cases of traveling salesman and repairman problems with time windows. *NETWORKS: Networks: An International Journal*, 22, 1992.
13. Bang-Ye Wu. Polynomial time algorithms for some minimum latency problems. *Information Processing Letters*, 75(5):225–229, October 2000.