Shortest Path in Complete Bipartite Digraph Problem and its Applications

Xin He* Zhi-Zhong Chen †

Abstract

We introduce the *shortest path in complete bipartite digraph* (SPCB) problem: Given a weighted complete bipartite digraph $G = (X, Y, E)$ with $X = \{x_0, \ldots, x_n\}$ and $Y = \{y_0, \ldots, y_m\}$, find a shortest path from x_0 to x_n in G. For arbitrary weights, the problem needs at least $\Omega(nm)$ time to solve. We show if the weight matrices are *concave*, the problem can be solved in $O(n + m \log n)$ time.

As applications, we discuss the *traveling salesman* problem for points on a convex polygon and the *minimum latency tour* problem for points on a straight line. The known algorithms for both problems require $\Theta(n^2)$ time. Using our SPCB algorithm, we show they can be solved in $O(n \log n)$ time. These results solve two open questions posed by Marcotte and Suri [10]; and by Afrati *et. al.* [1].

1 Introduction

Let $G = (X, Y, E)$ be a complete bipartite digraph with $X = \{x_0, \ldots, x_n\}$ and $Y = \{y_0, \ldots, y_m\}$. Each edge $e \in E$ has a real-valued weight $w(e)$. We use $x_i \rightarrow y_j$ and $y_j \rightarrow x_i$ to denote the edges. Let $A[0..n, 0..m]$ be the matrix with $A[i, j] = w(x_i \rightarrow y_j)$ and $B[0..m, 0..n]$ be the matrix with $B[i, j] = w(y_i \rightarrow x_j)$. (In applications, A and B are not explicitly stored. Rather, an entry is computed in $O(1)$ time when it is needed). The weight of a path P in G is defined to be $w(P) = \sum_{e \in P} w(e)$. The *shortest path in complete bipartite digraph* (SPCB) problem is: given such a digraph G, find a path P in G from x_0 to x_n such that $w(P)$ is minimized. (We require that G contains no negative cycles, since otherwise the shortest path of G is not well-defined.) For arbitrary weight matrices, we need at least $\Omega(nm)$ time to solve the problem since all edges of G must be examined. A matrix $M[0..n, 0..m]$ is called *concave* if the following hold:

$$(1.1)\ M[i_1, j_1] + M[i_2, j_2] \leq M[i_2, j_1] + M[i_1, j_2]$$

$$\text{for } 0 \leq i_1 \leq i_2 \leq n \text{ and } 0 \leq j_1 \leq j_2 \leq m$$

Concave matrices were first discussed in [12] and have been very successfully used in solving various problems (see [2, 3, 4, 6, 7, 8, 9, 11, 12, 13] and the references cited within). In this paper, we show that if both A and B are concave, the SPCB problem can be solved in $O(n + m \log n)$ time. (Even for this special case, no algorithm with $o(nm)$ running time is previously known). In designing our algorithm, we extend the algorithms by Wilber for solving the *least weight subsequence* problem [11] and the algorithm in [3] for solving the *column minima searching* problem in monotone matrices. The concavity of matrices plays a crucial role in our algorithm.

The setting of the SPCB problem is quite general and it may be used to solve other problems. In particular, we discuss two of its applications which solve two open questions. The first one is the *Traveling Salesman* problem (TSP) for points on an n-vertex convex polygon Q. Given two points x and y on Q, we want to find a Hamiltonian path P containing all points of Q from x to y such that the total weight of P is minimized, where the Euclidean distance is used. This problem can be solved in $\Theta(n^2)$ time by dynamic programming [10]. It was posed in [10] as an open question whether there exists an $o(n^2)$ algorithm for solving the problem. We show the problem can be reduced to the SPCB problem and solved in $O(n \log n)$ time.

The second application is the *Minimum Latency Tour* (MLT) problem [5]. Given a set S of n points, a symmetric distance matrix, and a tour T which visits the points of S, the *latency* of a point p is the length of the tour from the staring point to p. The *total latency* $w(T)$ is the sum of the latencies of all points. We wish to find a tour T such that $w(T)$ is minimized. This problem is also known as *delivery-man* or *traveling repairman* problem in the literature [5]. The MLT problem is very different from the TSP problem in nature [5]. For general case, it is NP-complete [5]. Even for points on a tree or on a convex polygon, it is not known whether the MLT problem is in P or NP-complete [5]. The case where points are on a straight line was considered in [1, 5]. This case is interesting since it is exactly the following *disk head scheduling problem*: A disk head

*Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14260. Research supported in part by NSF grant CCR-9205982. e-mail:xinhe@cs.buffalo.edu

†Department of Mathematical Sciences, Tokyo Denki University, Hatoyama, Saitama 350-03, Japan.

moves along a straight line L. The head must visit a set of n points on L in order to satisfy disk access requests. The time needed to travel is proportional to the distance being traveled. Once the head reaches a point, the disk access time can be ignored. We want to find a tour of the head such that the average delay (or equivalently, the total delay) of all requests is minimized. The MLT problem for this special case can be solved in $\Theta(n^2)$ time by dynamic programming [1, 5]. We show the problem can be reduced to the SPCB problem and solved in $O(n \log n)$ time.

The present paper is organized as follows. In section 2, we introduce definitions and background. Our algorithm for solving the SPCB problem and its analysis are given in sections 3 and 4. In sections 5 and 6, we present algorithms for the TSP problem for convex polygon and the MLT problem for straight line, respectively.

2 Definitions and Background

Given two matrices $A[0..n, 0..m]$ and $B[0..m, 0..n]$, the product $C[0..n, 0..n] = A \times B$ is defined by:

$$C[i,j] = \min_{0 \le k \le m} (A[i,k] + B[k,j]) \tag{2.2}$$

For $0 \le i \le n$ and $0 \le j \le n$, let $I(i,j)$ denote the smallest index k that realizes the minimum value in (2.2) (i.e. $C[i,j] = A[i, I(i,j)] + B[I(i,j), j]$). The following lemmas were proved in [12].

LEMMA 2.1. *If both A and B are concave, so is C.*

LEMMA 2.2. *For any i, j $(0 \le i < n,\ 0 \le j < n)$, we have: $I(i,j) \le I(i,j+1) \le I(i+1,j+1)$.*

Remark: The definitions of concavity and the matrix product in [12] are slightly different from the definitions used here. In [12], a concave matrix is an upper triangular matrix such that the condition (1.1) is true for $i_1 \le i_2 \le j_1 \le j_2$. In the matrix product definition (2.2), the minimum is taken over $i \le k \le j$. Under these definitions, Yao proved Lemmas 2.1 and 2.2. Under our definitions, Lemmas 2.1 and 2.2 can be proved by using similar method.

Let (i,j) and (i',j') be two pairs of indices. If $i \le i'$ and $j \le j'$, we write $(i,j) \prec (i',j')$. By Lemma 2.2, $(i,j) \prec (i',j')$ implies $I(i,j) \le I(i',j')$. We have the following lemma:

LEMMA 2.3. *Let $(i_1,j_1), (i_2,j_2), \ldots, (i_p,j_p)$ be p pairs of indices such that $(i_l,j_l) \prec (i_{l+1},j_{l+1})$ for all $1 \le l < p$. Then $I(i_1,j_1), I(i_2,j_2), \ldots, I(i_p,j_p)$ can be computed in $O(m \log p)$ time.*

Proof. This can be done in a binary search fashion. In the first stage, we find $I_{p/2} = I(i_{p/2}, j_{p/2})$ in $O(m)$ time. In the second stage, we find $I(i_{p/4}, j_{p/4})$ (by searching k in the range $0 \le k \le I_{p/2}$) and $I(i_{3p/4}, j_{3p/4})$ (by searching k in the range $I_{p/2} \le k \le m$). This totally needs $O(m)$ time. In general, each stage takes $O(m)$ time and there are $\log p$ stages. This proves the lemma. □

Consider a matrix $M[0..n, 0..m]$. For each column index $0 \le j \le m$, let $i(j)$ be the smallest row index such that $M(i(j), j)$ equals the minimum value in the jth column of M. The *column minima searching* problem for M is to find the $i(j)$'s for all $0 \le j \le m$. M is called *monotone* if $i(j_1) \le i(j_2)$ for all $0 \le j_1 < j_2 \le m$. M is *totally monotone* if every 2×2 submatrix of M is monotone [3]. If M is concave, it is easy to check that M is totally monotone. For a totally monotone matrix M, the column minima searching problem for M can be solved in $O(n+m)$ time, provided that each entry of M can be evaluated in $O(1)$ time [3]. Following [8], we refer to the algorithm in [3] as SMAWK algorithm.

The following *least weight subsequence* (LWS) problem was introduced in [8]. Given a sequence $\{x_0, x_1, \ldots, x_n\}$ and a real-valued weight function $w(x_i, x_j)$ defined for indices $0 \le i < j \le n$, find an integer $k \ge 1$ and a sequence $S = \{0 = i_0 < i_1 < \cdots < i_{k-1} < i_k = n\}$ such that the total weight $w(S) = \sum_{l=1}^{k} w(x_{i_{l-1}}, x_{i_l})$ is minimized. The weight function w is *concave* if the following hold:

$$\begin{aligned} &w(x_{i_1}, x_{j_1}) + w(x_{i_2}, x_{j_2}) \le \\ &\qquad w(x_{i_2}, x_{j_1}) + w(x_{i_1}, x_{j_2}) \\ &\text{for } 0 \le i_1 \le i_2 \le j_1 \le j_2 \le n \end{aligned} \tag{2.3}$$

If w is concave, Hirschberg and Larmore showed that the LWS problem can be solved in $O(n \log n)$ time [8]. Similar algorithms were also developed in [6, 7]. Wilber discovered an elegant linear time algorithm for solving this problem [11]. All these algorithms assume each entry $w(i,j)$ can be computed in constant time. From now on we only consider the concave LWS problem and the phrase "LWS problem" always means the concave LWS problem. We will show that an instance of the SPCB problem defined by concave matrices A and B can be reduced to an instance of an *enhanced* version of the LWS problem. However, in the reduced problem, the weight matrix w is the product matrix $A \times B$ (with operators min and +). Thus an entry $w(x_i, x_j)$ cannot be evaluated in $O(1)$ time. So when solving our problem, Wilber's algorithm and its analysis must be modified.

An instance of the *enhanced* LWS problem is a sequence $\{x_0, x_1, \ldots, x_n\}$ and a real-valued concave weight function (satisfying inequality (1.1)) $w(x_i, x_j)$ defined on all $0 \leq i, j \leq n$ such that $w(x_i, x_i) \geq 0$ for all $0 \leq i \leq n$. We want to find a sequence $S = \{0 = i_0, i_1, \ldots, i_k = n\}$, $(i_0, \ldots, i_k$ are not necessarily in increasing order), such that $w(S) = \sum_{l=1}^{k} w(x_{i_{l-1}}, x_{i_l})$ is minimized. In terms of graph formulation, we are given a complete digraph G with vertex set $\{x_0, x_1, \ldots, x_n\}$ and a weight function w, we wish to find a shortest x_0 to x_n path in G. An edge $x_i \rightarrow x_j$ is called a *forward (backward)* edge if $i < j$ $(i > j)$.

LEMMA 2.4. *For any instance of the enhanced LWS problem, there exists a shortest x_0 to x_n path consisting of only forward edges.*

Proof. Let P be a shortest path from x_0 to x_n in G such that the number of edges in P is minimum. Since $w(x_i, x_i) \geq 0$ for all i, P contains no self loops. Toward a contradiction, suppose P contains a backward edge. Let $x_{i_l} \rightarrow x_{i_{l+1}}$ be the first backward edge of P. Thus $i_l > i_{l+1}$ and $i_l > i_{l-1}$. By the concavity of w and the assumption $w(x_i, x_i) \geq 0$ for all x_i, we have: $w(x_{i_{l-1}}, x_{i_{l+1}}) \leq w(x_{i_{l-1}}, x_{i_{l+1}}) + w(x_{i_l}, x_{i_l}) \leq w(x_{i_{l-1}}, x_{i_l}) + w(x_{i_l}, x_{i_{l+1}})$. Thus, if the two edges $x_{i_{l-1}} \rightarrow x_{i_l} \rightarrow x_{i_{l+1}}$ in P are replace by a single edge $x_{i_{l-1}} \rightarrow x_{i_{l+1}}$, we get a path P' such that $w(P') \leq w(P)$ and the number of edges in P' is one less than that in P. This contradicts the choice of P. □

Lemma 2.4 implies that there are no negative cycles in any instance of the enhanced LWS problem. It also implies we can ignore all backward edges and self-loops when solving the enhanced LWS problem.

Consider a SPCB instance defined by a complete bipartite digraph $G = (X, Y, E)$ and concave weight matrices A and B. Let G' be the complete digraph on X with concave weight matrix $w = A \times B$. If $w(x_i, x_i) \geq 0$ for all $0 \leq i \leq n$, then G', w define an instance of the enhanced LWS problem.

LEMMA 2.5. *Let A and B be two concave matrices such that all main diagonal entries of the matrix $w = A \times B$ are non-negative. If the enhanced LWS problem defined by w can be solved in $T(n, m)$ time, then the SPCB problem defined by A and B can be solved in $O(T(n, m) + m \log n)$ time.*

Proof. In order to solve the SPCB problem defined by matrices A and B, we first solve the enhanced LWS problem defined by the matrix $w = A \times B$. Let $P' = \{0 = i_0 < i_1 < \ldots i_k = n\}$ be the solution path found. We compute $j_1, j_2, \ldots, j_k$, where $j_l = I(i_{l-1}, i_l)$. Since $(i_0, i_1) \prec (i_1, i_2) \prec \ldots \prec (i_{k-1}, i_k)$, this can be done in $O(m \log n)$ time by Lemma 2.3. It can be shown the path $P = \{x_0 = x_{i_0} \rightarrow y_{j_1} \rightarrow x_{i_1} \rightarrow \ldots \rightarrow y_{j_k} \rightarrow x_{i_k} = x_n\}$ is a solution for the SPCB problem. □

We would like to use Wilber's algorithm in [11] to solve our enhanced LWS problem. However, Wilber's algorithm is for the (ordinary) LWS problem defined by a triangular matrix while our problem is defined by a full matrix. Also, Wilber's algorithm assumes $w(i, j)$ can be evaluated in $O(1)$ time while an entry in $w = A \times B$ needs $\Theta(m)$ time to evaluate. We address these issues in the following sections.

3 Wilber's Algorithm

In this subsection, we briefly describe Wilber's algorithm for solving the LWS problem. Then we show how to use Wilber's algorithm to solve the enhanced LWS problem.

Consider an instance of the LWS problem with the sequence $\{x_0, x_1, \ldots, x_n\}$ and the weight matrix w. Let $f(0) = 0$ and, for $1 \leq j \leq n$, let $f(j)$ be the weight of the lowest weight subsequence between x_0 and x_j. For $0 \leq i < j \leq n$, let $g(i, j)$ be the weight of the lowest weight subsequence between x_0 and x_j whose next to the last element is x_i. (That is, the lowest weight subsequence of the form $0 = l_0 < l_1 < \ldots < l_{k-1} = i < l_k = j$). Then we have:

$$(3.4) \quad \begin{cases} f(j) = \min_{0 \leq i < j} g(i, j) & (1 \leq j \leq n) \\ g(i, j) = f(i) + w(x_i, x_j) & (0 \leq i < j \leq n) \end{cases}$$

Adding $f(i_1) + f(i_2)$ to both sides of inequality (2.3) and apply definition (3.4), we get:

$$(3.5) \quad g(i_1, j_1) + g(i_2, j_2) \leq g(i_1, j_2) + g(i_2, j_1) \quad \text{for } 0 \leq i_1 \leq i_2 \leq j_1 \leq j_2 \leq n$$

We extend g to a full $(n+1) \times (n+1)$ matrix by setting $g(i, j) = +\infty$ for $0 \leq j \leq i \leq n$. It is easy to verify that the extended matrix g is totally monotone. Our goal is to determine the row index of the minimum value in each column of g. So we would like to simply apply SMAWK algorithm. But we cannot, because for $i < j$, the value of $g(i, j)$ depends on $f(i)$ which depends on all values of $g(l, i)$ for $0 \leq l < i$. So we cannot compute the value of g in $O(1)$ time as required by SMAWK algorithm.

Wilber's algorithm starts in the upper left corner of g and work rightwards and downwards, at each iteration learning enough new values for f to be able to compute

enough new values of g. Actually, during one step of each iteration, the algorithm might "pretend" to know values of f that it really does not have. At the end of the iteration, the assumed value of f is checked for validity.

We use $f(j)$ and $g(i,j)$ to refer to the correct value of f and g. The currently computed value for $f(j)$ is denoted by $F(j)$, and will sometimes be incorrect. The currently computed value of $g(i,j)$ is denoted by $G[i,j]$, and is always computed as $F[i] + w(i,j)$. So $G[i,j] = g(i,j)$ iff $F(i) = f(i)$. The algorithm does not explicitly store the matrices w, g, G. Rather, their entries are calculated when needed. Let $G[i_1, i_2; j_1, j_2]$ denote the submatrix of G consisting of the intersection of rows i_1 through i_2 and columns j_1 through j_2. $G[i_1, i_2; j]$ denotes the intersection of rows i_1 through i_2 with column j. The rows of G are indexed from 0 and the columns are indexed from 1. Wilber's algorithm is as follows.

Wilber's Algorithm:

$F[0] \leftarrow c \leftarrow r \leftarrow 0$.

while $(c < n)$ do:

1. $p \leftarrow \min\{2c - r + 1, n\}$.
2. Apply SMAWK algorithm to find the minimum in each column of submatrix $S = G[r, c; c+1, p]$. For $j \in [c+1, p]$, let $F[j]$ = the minimum value found in $G[r, c; j]$.
3. Apply SMAWK algorithm to find the minimum in each column of the submatrix $T = G[c+1, p-1; c+2, p]$. For $j \in [c+2, p]$, let $H[j]$ = the minimum value found in $G[c+1, p-1; j]$.
4. If there is an integer $j \in [c+2, p]$ such that $H[j] < F[j]$, then set j_0 to the smallest such integer. Otherwise, set $j_0 \leftarrow p+1$.
5. **if** $(j_0 = p+1)$ **then** $c \leftarrow p$; **else** $F[j_0] \leftarrow H[j_0]$; $r \leftarrow c+1$; $c \leftarrow j_0$.

Fig 1 shows the submatrices S and T during a typical iteration. Each time we are at the beginning of the loop, the following invariants hold:

(1) $r \geq 0$ and $c \geq r$;

(2) $F[j] = f(j)$ for $j \in [0, c]$;

(3) All minima in columns $c+1$ through n of g are in rows $\geq r$;

Thees invariants are clearly satisfied at the start when $r = c = 0$.

Invariant (2) implies $G[i,j] = g(i,j)$ for all j and $i \in [0, c]$. So the entries of S are the same as the corresponding entries of g. Thus S is totally monotone

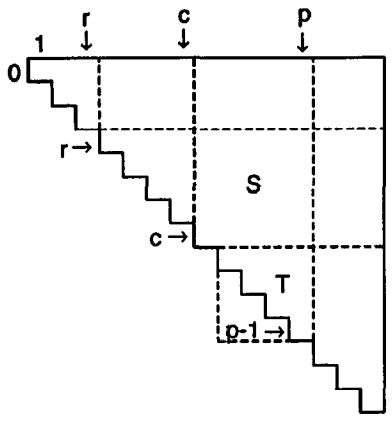

Figure 1: A typical iteration of Wilber's algorithm

and for each $j \in [c+1, p]$, step 2 sets $F[j]$ to the minimum value of $g(r, c; j)$. Since S contains all finite-valued cells in column $c+1$ of g that are in rows $\geq r$, $F[c+1] = f(c+1)$ at the end of step 2. On the other hand, we do not necessarily have $F[j] = f(j)$ for any $j \in [c+2, p]$, since g has finite-valued cells in those columns that are in rows $\geq r$ and not in S.

In step 3, we proceed as if $F[j] = f(j)$ for all $j \in [c+1, p-1]$. Since this may be false, some of the values in T may be bogus. However, T is always totally monotone for if we add $F[i_1] + F[i_2]$ to both sides of (2.3), we get $G[i_1, j_1] + G[i_2, j_2] \leq G[i_1, j_2] + G[i_2, j_1]$. Thus SMAWK algorithm works correctly and $H[j]$ is set to the minimum value of the subcolumn $G[c+1, p-1; j]$.

In step 4, we verify that $F[j] = f(j)$ for $j \in [c+2, p]$ (this is the case if $H[j] \geq F[j]$ for all $j \in [c+2, p]$); or find the smallest j where this condition fails (this is the case if $H[j] < F[j]$ for some $j \in [c+2, p]$). In either case, c and r are set accordingly at step 5 so that the loop invariants hold.

Next we discuss how to use Wilber's algorithm to solve an instance of the enhanced LWS problem defined by weight matrix w. Let L denote the portion of w consisting of the entries on and below the main diagonal of w. Let w' be the matrix obtained from w by replacing all entries in L by $+\infty$. Then w' defines an instance of the (ordinary) LWS problem. By Lemma 2.4, the solution for the problem defined by w' is identical to the solution for the problem defined by w. If each entry of w can be computed in $O(1)$ time, we can use Wilber's algorithm on w' to solve the problem. However, if the enhanced LWS problem is derived from an instance of the SPCB problem, the entries of the matrix $w = A \times B$ cannot be computed in $O(1)$ time. In this case, we cannot afford to change w to w' since doing so will distroy some properties of w that are crucial for obtaining a fast algorithm. Fortunately, we have:

LEMMA 3.1. *Wilber's algorithm solves the enhanced LWS problem without changing the weight matrix w.*

Proof. It is enough to show that the entries in L have no effects on the computation of Wilber's algorithm, regardless of whether they are changed to $+\infty$ or not. The only place where Wilber's algorithm needs the entries in L is step 3, where SMAWK algorithm is applied to the submatrix T. For each $j \in [c+2, p]$, let $F[j]$ and $H[j]$ be the minimum value of column j in S and T, respectively.

There are three cases:

(a) $F[j] \leq H[j]$;

(b) $F[j] > H[j]$ and $H[j]$ is not in L (i.e. $H[j] = G[i,j]$ for some $i < j$);

(c) $F[j] > H[j]$ and $H[j]$ is in L (i.e. $H[j] = G[i,j]$ for some $i \geq j$).

In cases (a) and (b), the values in L does not affect the computation. In the following we show case (c) cannot occur. Toward a contradiction, assume there exist indices $j \in [c+2, p]$ and i such that $i \geq j$ and $H[j] = G[i,j] < F[j]$.

Case 1: $i = j$. Then $H[j] = G[j,j] = F[j] + w(j,j) \geq F[j]$. This is impossible.

Case 2: $i > j$. In this case, $H[j] = G[i,j] = F[i] + w(i,j)$. Recall that $F[i]$ is the minimum value of the subcolumn $G[r, c; i]$. Suppose $F[i] = G[t,i] = F[t] + w(t,i)$ for some $r \leq t \leq c$. Note that $t \leq c < i$ and $j < i$. By the concavity of w, we have: $w(t,j) + w(i,i) \leq w(t,i) + w(i,j)$. Since $w(i,i) \geq 0$ for all i, we have: $H[j] = F[i] + w(i,j) = F[t] + w(t,i) + w(i,j) \geq F[t] + w(t,j) + w(i,i) \geq F[t] + w(t,j) = G[t,j] \geq F[j]$. This contradicts the assumption that $H[j] < F[j]$. □

4 Implementation and Time Analysis

In this section, we discuss how to use Wilber's algorithm to solve an instance of the enhanced LWS problem derived from an instance of the SPCB problem. Namely, the enhanced LWS problem is defined by the matrix $C = A \times B$ where $C[i,i] \geq 0$ for all i. During each stage of Wilber's algorithm (steps 2 and 3), we need to find column minima of submatrices S and T. Both S and T have the form $C'[r, c; q, p]$ where $C'[i,j] = F[i] + C[i,j]$ for some known value $F[i]$. Since $C'[i,j]$ cannot be computed in $O(1)$ time, we cannot use SMAWK algorithm. Instead, we use the algorithm given in the following lemma. (Similar methods was used in [2]).

LEMMA 4.1. *The column minima searching problem for the submatrix $C'[r, c; q, p]$ with $r \leq q$ and $c \leq p$ can be solved in $O((c-r) + (p-q) + (k_2 - k_1))$ time, where $k_1 = I(r,r)$ and $k_2 = I(p,p)$.*

Proof. By Lemma 2.2, for each $i \in [r, c]$ and $j \in [q, p]$, $C[i,j] = \min_{0 \leq k \leq m}(A[i,k] + B[k,j])$ can be computed by searching k in the range $k \in [k_1, k_2]$. For $j \in [q, p]$, let $d(j)$ denote the column minimum of $C'[r, c; j]$. Then:

$$\begin{aligned} d(j) &= \min_{r \leq i \leq c}\{F[i] + C[i,j]\} \\ &= \min_{r \leq i \leq c}\{F[i] + \min_{k_1 \leq k \leq k_2}(A[i,k] + B[k,j])\} \\ &= \min_{k_1 \leq k \leq k_2}\{B[k,j] + \min_{r \leq i \leq c}(F[i] + A[i,k])\} \end{aligned}$$

For $i \in [r, c]$ and $k \in [k_1, k_2]$, let $A'[i,k] = F[i] + A[i,k]$. Then A' is totally monotone. For each $k \in [k_1, k_2]$, let $J[k]$ be the minimum of the subcolumn $A'[r, c; k]$.

For $k \in [k_1, k_2]$ and $j \in [q, p]$, let $B'[k,j] = B[k,j] + J[k]$. Then B' is totally monotone. Clearly, $d(j)$ is the minimum of the subcolumn $B'[k_1, k_2; j]$. Thus the column minima $d(j)$'s of $C'[r, c; q, p]$ can be found by two applications of SMAWK algorithm, once on A' and once on B'. So the total time is $O((c-r) + (k_2 - k_1)) + O((k_2 - k_1) + (p-q)) = O((c-r) + (p-q) + (k_2 - k_1))$. □

Each iteration of Wilber's algorithm is completely specified by three parameters: r, c, p. Let r_i, c_i, p_i be the values of these parameters at the beginning of the ith iteration. $r_{i+1}, c_{i+1}, p_{i+1}$ are calculated in step 5 as follows:

Case 1: "then" part of step 5 is executed. In this case, $r_{i+1} = r_i$; $c_{i+1} = p_i$; and (1a) $p_{i+1} = 2c_{i+1} - r_{i+1} + 1$, if it is $\leq n$; or (1b) $p_{i+1} = n$, otherwise.

Case 2: "else" part is executed. In this case, $r_{i+1} = c_i + 1$, $c_{i+1} = j_0$ ($c_i + 2 \leq j_0 \leq p_i$); and (2a) $p_{i+1} = 2c_{i+1} - r_{i+1} + 1$, if it is $\leq n$; or (2b) $p_{i+1} = n$, otherwise.

If the case 1a (or 1b, 2a, 2b, resp.) applies to the ith iteration, we call it a type 1a (or 1b, 2a, 2b, resp.) iteration. We call $[r_i, p_i]$ the ith *span*; r_i and p_i the left and the right end of the ith span, resp. Note that after a type 1a or 1b iteration, the left end of the $(i+1)$st span is not changed, the right end of the $(i+1)$st span increases. After a type 2a or 2b iteration, the left end of the $(i+1)$st span increases, the right end of $(i+1)$st span may increase or decrease. For an interval $[t, t+1]$ ($0 \leq t < n$), we say a span $[r_i, p_i]$ *covers* $[t, t+1]$, if $r_i \leq t$ and $t+1 \leq p_i$. Since the left end of spans never decreases, the spans "move" from left to right. Once the left end of a span is $\geq t+1$, $[t, t+1]$ will never be covered by subsequent spans. We make the following observations.

(1) If a type 1a or 1b iteration follows a type 1b or 2b iteration, the algorithm terminates immediately.

(2) If the ith iteration is of type 1a, then: $p_{i+1} - r_{i+1} = (2c_{i+1} - r_{i+1} + 1) - r_{i+1} = 2(p_i - r_i) + 1$. Namely,

the length of the $(i+1)$st span is 1 + twice the length of the ith span.

(3) Suppose the ith iteration is of type 2a or 2b. Since $p_i \leq 2c_i - r_i + 1$, we have $c_i \geq (p_i + r_i - 1)/2$. Hence: $r_{i+1} = c_i + 1 \geq (p_i + r_i - 1)/2 + 1$.

(4) Suppose an interval $[t, t+1]$ is covered by the ith span $[r_i, p_i]$. If the ith iteration is of type 1a or 1b, and the $(i+1)$st iteration is of type 2a or 2b, then $r_{i+2} = c_{i+1} + 1 = p_i + 1 > t + 1$. Hence $[t, t+1]$ is not covered by $[r_{i+2}, p_{i+2}]$ and subsequent spans.

LEMMA 4.2. *Any interval $[t, t+1]$ $(0 \leq t < n)$ is covered by at most $2 \log n + 2$ spans.*

Proof. Let $[r_{i_1}, p_{i_1}], [r_{i_2}, p_{i_2}], \ldots, [r_{i_k}, p_{i_k}]$ be all spans covering $[t, t+1]$, where $i_1 < i_2 < \ldots < i_k$. Thus, $r_{i_l} \leq t$ and $t + 1 \leq p_{i_l}$ for all $1 \leq l \leq k$. Let l be the first index such that the i_lth iteration is of type 1a or type 1b. (If no such l exists, let $l = k$). We first show $k - l \leq \log n + 2$.

Case 1: The i_lth iteration is of type 1b. If the (i_l+1)st iteration is of type 1a or 1b, then the algorithm terminates by observation (1). If the $(i_l + 1)$st iteration is of type 2a or 2b, then by observation (4), $[t, t+1]$ is not coverted by $[r_{i_l+2}, p_{i_l+2}]$ and all subsequent spans.

Case 2: The i_lth iteration is of type 1a. Let s be the largest integer such that the iterations $i_l, i_l+1, \ldots, i_l+s$ are all of type 1a. Clearly, $[t, t+1]$ is covered by all spans $[r_{i_l+1}, p_{i_l+1}], \ldots, [r_{i_l+s}, p_{i_l+s}]$. By observation (2), each type 1a iteration doubles the length of the span. Since the length of any span is at most n, we have $s \leq \log n$. The $(i_l + s + 1)$st iteration is of either type 1b or 2a or 2b. If it is of type 2a or 2b, then by the observation (4), $[t, t+1]$ is not covered by the $(i_l + s + 2)$nd and all subsequent spans. If the $(i_l + s + 1)$st iteration is of type 1b, then similar to case 1, the algorithm either terminates at the $(i_l + s + 2)$nd iteration; or $[t, t+1]$ is not covered by the $(i_l + s + 2)$nd and all subsequent spans.

In either case, the number of spans following the i_lth iteration that cover $[t, t+1]$ is at most $\log n + 2$. So $k - l \leq \log n + 2$. Next we show $l \leq \log n$ and this will complete the proof of the lemma.

For each $1 \leq h < l$, the i_hth iteration is of type 2a or 2b. Fix an index h. For each $j \geq i_h$, let $L_j = (t+1) - r_j$. By the fact that $t+1 \leq p_{i_h}$ and observation (3), we have:

$L_{i_h+1} = (t+1) - r_{i_h+1} \leq (t+1) - ((p_{i_h} + r_{i_h} - 1)/2 + 1) = (2t - p_{i_h} - r_{i_h} + 1)/2 \leq (t - r_{i_h})/2 < L_{i_h}/2$.

Since the left end of the spans never decreases, this implies that $L_{i_{h+1}} \leq L_{i_h+1} < L_{i_h}/2$. This is true for all $1 \leq h < l$. Hence $L_{i_l} < L_{i_1}/2^l$. If $l > \log n$, then L_{i_l} becomes 0 and the interval $[t, t+1]$ is not covered by $[r_{i_l}, p_{i_l}]$ and subsequent spans. So we must have $l \leq \log n$. This proves the lemma. □

THEOREM 4.1. *Given two concave matrices A, B such that the main diagonal entries of the matrix $C = A \times B$ are non-negative, the SPCB problem defined by A and B can be solved in $O(n + m \log n)$ time.*

Proof. Given an instance of the SPCB problem defined by matrices A and B, we first compute $I(0,0), I(1,1), \ldots, I(n,n)$. This takes $O(m \log n)$ time by Lemma 2.3. Then we use Wilber's algorithm to solve the enhanced LWS problem defined by the matrix $C = A \times B$. But instead of using SMAWK algorithm, we use the subroutine in Lemma 4.1 for finding column minima in S and T. If we can show the time needed by these subroutine calls is $O(n + m \log n)$, the theorem will follow from Lemma 2.5.

Consider the ith iteration. We need to find the column minima of $S_i = G[r_i, c_i; c_i + 1, p_i]$ and $T_i = G[c_i+1, p_i-1; c_i+2, p_i]$. Let $k_1 = I(r_i, r_i)$, $k_2 = I(p_i, p_i)$ and $k_3 = I(c_i + 1, c_i + 1)$. Since $r_i < c_i + 1 \leq p_i$, we have: $k_1 \leq k_3 \leq k_2$ by Lemma 2.2. By Lemma 4.1, the searching of S_i needs $O((c_i - r_i) + (p_i - c_i - 1) + (k_2 - k_1)) = O((p_i - r_i) + (k_2 - k_1))$ time. The searching time of T_i is $O((p_i - 1 - c_i - 1) + (p_i - c_i - 2) + (k_2 - k_3)) = O((p_i - r_i) + (k_2 - k_1))$. Thus the total time needed to search S_i and T_i in all iterations is $\sum_{i=1}^{K} O((p_i - r_i) + (I(p_i, p_i) - I(r_i, r_i)))$, where K is the total number of iterations. Since Wilber's algorithm takes $O(n)$ time, the term $\sum_{i=1}^{K} O(p_i - r_i)$ is bounded by $O(n)$. On the other hand,

$$\sum_{i=1}^{K} (I(p_i, p_i) - I(r_i, r_i)) =$$

$$\sum_{i=1}^{K} \sum_{t=r_i}^{p_i - 1} (I(t+1, t+1) - I(t,t)) =$$

$$\sum_{t,i \text{ where } [t,t+1] \in [r_i, p_i]} (I(t+1, t+1) - I(t,t))$$

By Lemma 4.2, each interval $[t, t+1]$ is covered by at most $2 \log n + 2$ spans. Thus the above sum is bounded by: $O(\log n \sum_{t=0}^{n-1} (I(t+1, t+1) - I(t,t))) = O(m \log n)$ as to be shown. □

5 TSP Problems for Points on a Convex Polygon

Let Q be a convex polygon. For any two points $x, y \in Q$, let $d(x, y)$ be the Euclidean distance between x and y. Let P be a path connecting points on Q. Given two points x, y of Q, we wish to find a Hamiltonian path P of Q from x to y such that the weight $w(P) = \sum_{e \in P} d(e)$ is minimized. By a geometric argument, one can show the optimal path P is simple (i.e. no two edges of P cross each other).

Let $Q_X = \{x = x_0, x_1, \ldots, x_n = y\}$ be the points of Q from x to y in clockwise order. Let $Q_Y = \{x = y_0, y_1, \ldots, y_m = y\}$ be the points of Q from x to y in counterclockwise order. Let $x_i \xrightarrow{X} x_j$ denote the portion of Q_X from x_i to x_j and $y_i \xrightarrow{Y} y_j$ denote the portion of Q_Y from y_i to y_j.

Let P be an optimal Hamiltonian path from $x_0 = y_0$ to $x_n = y_m$. We assume both the first and the last edge of P are in Q_Y. Then P must be of the following form. (See Fig 2. For clarity, the points in Q_X and Q_Y are drawn on two vertical lines).

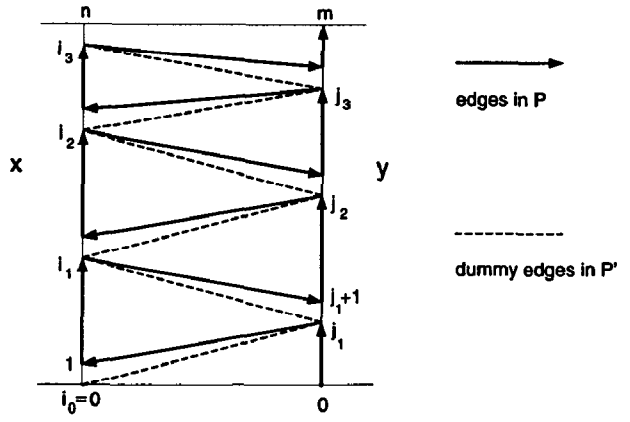

Figure 2: Optimal path in a convex polygon

$$x_{i_0} = x_0 = y_0 \xrightarrow{Y} y_{j_1} \to x_1 \xrightarrow{X} x_{i_1} \to y_{j_1+1} \xrightarrow{Y} y_{j_2} \to x_{i_1+1} \xrightarrow{X} \ldots \xrightarrow{X} x_{i_t} = x_{n-1} \to y_{j_t+1} \xrightarrow{Y} y_m = x_n$$

for some $0 < j_1 < \ldots < j_t < m-1$ and $0 = i_0 < i_1 < \ldots < i_t = n-1$. We use the following *dummy path* $P' = \{0 = i_0 \to j_1 \to i_1 \to j_2 \to i_2 \to \ldots \to j_{t-1} \to i_{t-1} \to j_t \to i_t = n-1\}$ to represent P. Each edge $i_{l-1} \to j_l$ and $j_l \to i_l$ in P' is called a *dummy edge*. P is completely specified by P'.

For each dummy edge $i_{l-1} \to j_l$ in P', the edge $x_{i_{l-1}} \to x_{i_{l-1}+1}$ is not in P, while the edge $y_{j_l} \to x_{i_{l-1}+1}$ is in P. For each dummy edge $j_l \to i_l$ in P', the edge $y_{j_l} \to y_{j_l+1}$ is not in P, while the edge $x_{i_l} \to y_{j_l+1}$ is in P. This motivates the following definition of the weighs of dummy edges:

$$A[x_i, y_j] = w(x_i \to y_j) = d(x_{i+1}, y_j) - d(x_i, x_{i+1});$$
$$B[y_j, x_i] = w(y_j \to x_i) = d(y_{j+1}, x_i) - d(y_j, y_{j+1}).$$

Note that $A[x_0, y_0] = B[y_0, x_0] = 0$. Let $S_X = \sum_{i=1}^{n-1} d(x_{i-1}, x_i)$ and $S_Y = \sum_{j=1}^{m} d(y_{j-1}, y_j)$. One can verify that the total weight of P is:

$$(5.6)\quad w(P) = S_X + S_Y + \sum_{l=1}^{t} A[x_{i_{l-1}}, y_{j_l}] + \sum_{l=1}^{t} B[y_{j_l}, x_{i_l}]$$

Although the above discussion is carried out by assuming the first and the last edges of P are in Q_Y, it also applies to other cases. (If the first edge of P is in Q_X, let $j_1 = 0$. If the last edge of P is in Q_X, let $j_t = m-1$). It is easy to verify equation (5.6) is valid for these cases too. Since the term $S_X + S_Y$ in (5.6) is fixed, in order to minimize $w(P)$, we only need to minimize the *reduced weight*:

$w(P') = \sum_{l=1}^{t} A[x_{i_{l-1}}, y_{j_l}] + \sum_{l=1}^{t} B[y_{j_l}, x_{i_l}]$.

Let $G = (X, Y, E)$ be the complete bipartite digraph with $X = \{x_0, x_1, \ldots, x_{n-1}\}$, $Y = \{y_0, \ldots, y_{m-1}\}$ and the weight matrices A and B. Then a dummy path P' with minimum reduced weight $w(P')$ is exactly a shortest path in G from x_0 to x_{n-1}. For $0 \le i < i' \le n-1$ and $0 \le j < j' \le m-1$, by the definition of A and the fact that Q is a convex polygon, we have:

$A[i,j] + A[i',j'] - A[i,j'] - A[i',j] = d(x_{i+1}, y_j) + d(x_{i'+1}, y_{j'}) - d(x_{i+1}, y_{j'}) - d(x_{i'+1}, y_j) \le 0$

Thus A is concave. Similarly, we can show B is also concave. Let $C = A \times B$. Then:

$$C[i,i] = \min_{0 \le j \le m-1} [d(x_{i+1}, y_j) - d(x_i, x_{i+1}) + d(y_{j+1}, x_i) - d(y_j, y_{j+1})]$$

By the triangle inequality, each term in min sign is ≥ 0. So $C[i,i] \ge 0$ for all i. By Theorem 4.1, we have:

THEOREM 5.1. *The TSP problem for an N-point convex polygon can be solved in $O(N \log N)$ time.*

6 Minimum Latency Problem for Points on a Straight Line

Consider a set S of $n+1$ points, a symmetric distance matrix $d[0..n, 0..n]$, and a tour T which visits the points of S in the order $p_0, p_1, \ldots, p_n$ starting at p_0. Let $d(p_{i-1}, p_i)$ be the distance traveled along T between p_{i-1} and p_i. Then the *latency* of p_i on T is $w(p_i) = \sum_{j=1}^{i} d(p_{j-1}, p_j)$. The *total latency* $w(T)$ of T is the sum of the latencies of all points: $w(T) = \sum_{i=1}^{n} w(p_i)$. Or, equivalently:

$$(6.7)\quad w(T) = \sum_{k=1}^{n} d(p_{k-1}, p_k)(n-k+1)$$

We wish to find a tour T with minimum $w(T)$. In this section, we show that the MLT problem for points on a straight line can be reduced to the SPCB problem. Let $S = \{x_n, \ldots x_1, x_0 = y_0 = 0, y_1, \ldots, y_m\}$ be a set of points on the real line from left to right. We overload x_i

(and y_j) to denote both a point and the distance from it to the origin. The tour starts at the point 0. Define:

$$w(T_X) = \sum_{k=1}^{n}(x_k - x_{k-1})(n-k+1)$$

$$w(T_Y) = \sum_{k=1}^{m}(y_k - y_{k-1})(m-k+1)$$

$w(T_X)$ is the total latency of the tour T_X that starts at $x_0 = 0$ and travels the points $x_1, x_2, \ldots, x_n$. $w(T_Y)$ is the total latency of the tour T_Y that starts at $y_0 = 0$ and travels the points $y_1, y_2, \ldots, y_m$.

Consider an optimal tour T for S. We assume the first edge is to the right and the last edge is to the left. Then T must be of the following form (see Fig 3):

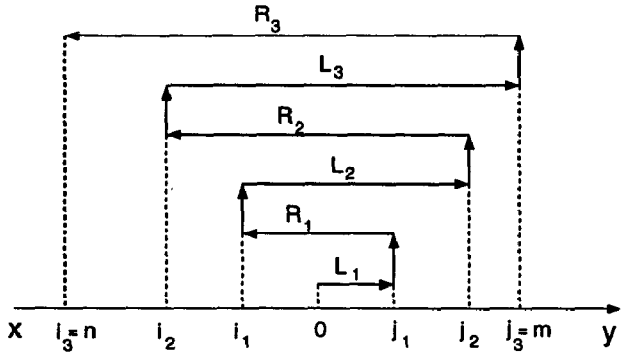

Figure 3: Optimal tour for points on a straight line

$$x_{i_0} = y_{j_0} = x_0 = y_0 \stackrel{\Delta}{\rightarrow} y_{j_1} \stackrel{\Delta}{\rightarrow} x_{i_1} \stackrel{\Delta}{\rightarrow} y_{j_2} \stackrel{\Delta}{\rightarrow} x_{i_2} \cdots \stackrel{\Delta}{\rightarrow} x_{i_{t-1}} \stackrel{\Delta}{\rightarrow} y_{j_t} = y_m \stackrel{\Delta}{\rightarrow} x_{i_t} = x_n$$

for some $0 = j_0 < j_1 < \ldots j_{t-1} < j_t = m$ and $0 = i_0 < i_1 < \ldots i_{t-1} < i_t = n$. ($x_i \stackrel{\Delta}{\rightarrow} y_j$ denotes a path from x_i to y_j consisting of several edges). We use the following *dummy tour* $T' = \{0 = i_0 \rightarrow j_1 \rightarrow i_1 \rightarrow \ldots \rightarrow i_{t-1} \rightarrow j_t = m \rightarrow i_t = n\}$ to represent T. For each $1 \le l \le t$, let L_l denote the subpath $x_{i_{l-1}} \stackrel{\Delta}{\rightarrow} y_{j_l}$. Let R_l denote the subpath $y_{j_l} \stackrel{\Delta}{\rightarrow} x_{i_l}$. The contribution of L_l to $w(T)$ can be calculated as follows.

$$\begin{aligned}
&w(L_l) = (y_{j_{l-1}} + x_{i_{l-1}})(n+m-i_{l-1}-j_{l-1}) + \\
&\textstyle\sum_{k=j_{l-1}+1}^{j_l}(y_k - y_{k-1})(n+m-i_{l-1}-k+1) \\
&= (y_{j_{l-1}} + x_{i_{l-1}})(n+m-i_{l-1}-j_{l-1}) + \\
&\quad \textstyle\sum_{k=j_{l-1}+1}^{j_l}(y_k - y_{k-1})(n-i_{l-1}) + \\
&\quad \textstyle\sum_{k=j_{l-1}+1}^{j_l}(y_k - y_{k-1})(m-k+1) \\
&= (y_{j_{l-1}} + x_{i_{l-1}})(n+m-i_{l-1}-j_{l-1}) + \\
&\quad (y_{j_l} - y_{j_{l-1}})(n-i_{l-1}) + \\
&\quad \textstyle\sum_{k=j_{l-1}+1}^{j_l}(y_k - y_{k-1})(m-k+1)
\end{aligned}$$

Similarly, the contribution of R_l to $w(T)$ is:

$$\begin{aligned}
w(R_l) &= (x_{i_{l-1}} + y_{j_l})(n+m-i_{l-1}-j_l) + \\
&\quad (x_{i_l} - x_{i_{l-1}})(m-j_l) + \\
&\quad \textstyle\sum_{k=i_{l-1}+1}^{i_l}(x_k - x_{k-1})(n-k+1)
\end{aligned}$$

Summing up and simplifying, we have:

$$\begin{aligned}
&w(T) = \textstyle\sum_{l=1}^{t} w(L_l) + \sum_{l=1}^{t} w(R_l) \\
&= \textstyle\sum_{l=1}^{t}\sum_{k=j_{l-1}+1}^{j_l}(y_k - y_{k-1})(m-k+1) + \\
&\quad \textstyle\sum_{l=1}^{t}\sum_{k=i_{l-1}+1}^{i_l}(x_k - x_{k-1})(n-k+1) + \\
&\quad \textstyle\sum_{l=1}^{t}\{(y_{j_{l-1}} + x_{i_{l-1}})(n+m-i_{l-1}-j_{l-1}) + \\
&\quad (y_{j_l} - y_{j_{l-1}})(n-i_{l-1})\} + \\
&\quad \textstyle\sum_{l=1}^{t}\{(x_{i_{l-1}} + y_{j_l})(n+m-i_{l-1}-j_l) + \\
&\quad (x_{i_l} - x_{i_{l-1}})(m-j_l)\} \\
&= \textstyle\sum_{k=1}^{j_t(=m)}(y_k - y_{k-1})(m-k+1) + \\
&\quad \textstyle\sum_{k=1}^{i_t(=n)}(x_k - x_{k-1})(n-k+1) + \\
&\quad \textstyle\sum_{l=1}^{t} y_{j_{l-1}}[(n+m-i_{l-1}-j_{l-1}) - (n-i_{l-1})] + \\
&\quad \textstyle\sum_{l=1}^{t} y_{j_l}[(n-i_{l-1}) + (n+m-i_{l-1}-j_l)] + \\
&\quad \textstyle\sum_{l=1}^{t} x_{i_{l-1}}[(n+m-i_{l-1}-j_{l-1}) + \\
&\quad (n+m-i_{l-1}-j_l) - (m-j_l)] + \textstyle\sum_{l=1}^{t} x_{i_l}[m-j_l] \\
&= w(T_X) + w(T_Y) + \\
&\quad \textstyle\sum_{l=0}^{t-1} y_{j_l}[m-j_l] + \sum_{l=1}^{t} y_{j_l}[2n+m-2i_{l-1}-j_l] + \\
&\quad \textstyle\sum_{l=0}^{t-1} x_{i_l}[2n+m-2i_l-j_l] + \sum_{l=1}^{t} x_{i_l}[m-j_l] \\
&= w(T_X) + w(T_Y) + \textstyle\sum_{l=0}^{t} y_{j_l}[2n+2m-2i_{l-1}-2j_l] \\
&\quad + \textstyle\sum_{l=0}^{t} x_{i_l}[2n+2m-2i_l-2j_l]
\end{aligned}$$

(The following facts are used in the last step: $m - j_t = 0$; $y_{j_0} = 0$; $2n + m - 2i_t - j_t = 0$; and $x_{i_0} = 0$. In the first summation of the last equation, the value of i_{-1} is irrelevant and we may define $i_{-1} = -1$). Define the *reduced weight* of T' to be:

$$w(T') = \sum_{l=0}^{t} y_{j_l}[n+m-i_{l-1}-j_l] + \sum_{l=0}^{t} x_{i_l}[n+m-i_l-j_l]$$

Then we have:

$$w(T) = w(T_X) + w(T_Y) + 2w(T') \tag{6.8}$$

Although the above discussion is carried out by assuming the first edge of T is to the right and the last edge is to the left, it also applies to other cases. (If the first edge is to the left, let $j_1 = 0$. If the last edge is to the right, let $i_{t-1} = n$ and delete the subpath R_t from T). It can be verified that (6.8) is valid for those cases

too. Since the term $w(T_X) + w(T_Y)$ is fixed, in order to minimize $w(T)$, we need to minimize $w(T')$.

Let $G = (X, Y, E)$ be the complete bipartite digraph with $X = \{x_0, x_1, \ldots, x_n\}$, $Y = \{y_0, y_1, \ldots, y_m\}$ and the weight matrices $A[0..n, 0..m]$ and $B[0..m, 0..n]$ defined as follows:

$A[i,j] = w(x_i \rightarrow y_j) = y_j(n+m-i-j)$; and

$B[j,i] = w(y_j \rightarrow x_i) = x_i(n+m-i-j)$

Note that $A[0,0] = B[0,0] = 0$. It is easy to check that a dummy tour T' with minimum reduced weight $w(T')$ is exactly a shortest path in G from x_0 to x_n. For $0 \leq i < i' \leq n$ and $0 \leq j < j' \leq m$, we have: $(A[i,j]+A[i',j'])-(A[i,j']+A[i',j]) = (i'-i)(y_j - y_{j'}) < 0$. Thus A is concave. Similarly, we can show B is also concave. Since all entries of A and B are non-negative, all entries of $C = A \times B$ are non-negative. Thus, by Theorem 4.1, we have:

THEOREM 6.1. *The MLT problem for N points on straight line can be solved in $O(N \log N)$ time.*

Open Problems: The setting of the SPCB problem is quite general. It is intersting to find other applications of the SPCB problem. In particular, we tried to use this technique to solve the MLT problem for points on a convex polygon. In the two applications discussed in this paper, the optimal paths are simple (i.e. no two edges of the path cross). Unfortunately, the optimal tour in the MLT problem for points on a convex polygon does not have this crucial property. It will be interesting to find a polynomial time algorithm for solving the MLT problem for this case.

References

[1] F. Afrati, S. Cosmadakis, C. Papadimitriou, G. Papageorgiou, and N. Papkostantinou, The Complexity of the Traveling Repairman Problem, Informatique Theorique et Applications (Theoretical Informatics and Applications) 20(1), 1986, pp. 79-87.

[2] A. Aggarwal and J. Park, Notes on Searching in Multidimensional Monotone Arrays, in Proc. 29th IEEE FOCS, 1988, pp. 497-512.

[3] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor and R. Wilber, Geometric Applications of a Matrix Searching Algorithm, Algorithmica 2, 1987, pp. 195-208.

[4] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, and S-H Teng, Constructing Trees in Parallel, in Proc. ACM SPAA, 1989, pp. 421-431.

[5] A. Blum, P. Chalasani, D. Coppersmith, B. Pulleyblank, P. Raghavan and M. Sudan, The Minimum Latency Problem, in Proc. 26th ACM STOC, 1994, pp. 163-171.

[6] D. Eppstein, Sequence Comparison with Mixed Convex and Concave Costs, J. of Algorithms 11, 1990, pp. 85-101.

[7] Z. Galil and R. Giancarlo, Speeding-up Dynamic Programming with Applications to Molecular Biology, Theoretical Computer Science 64, 1989, pp. 107-118.

[8] D. S. Hirschberg and L. L. Larmore, The Least Weight Subsequence Problem, SIAM J. Comput. 16(4), 1987, pp. 628-638.

[9] M. M. Klawe and D. J. Kleitman, An Almost Linear Time Algorithm for Generalized Matrix Searching, SIAM J. Disc. Math. 3(1), 1990, pp. 81-97.

[10] O. Marcotte and S. Suri, Fast Matching Algorithms for Points on a Polygon, SIAM J. Comput. 20(3), 1991, pp. 405-422.

[11] R. Wilber, The Concave Least-Weight Subsequence Problem Revisited, J. of Algorithms 9, 1988, pp. 418-425.

[12] F. F. Yao, Efficient Dynamic Programming Using Quadrangle Inequalities, in Proc. 12th ACM STOC, 1980, pp. 429-435.

[13] F. F. Yao, Speed-up in Dynamic Programming, SIAM J. Alg. Meth. 3(4), 1982, pp. 532-540.