[KL75] R.M. Karp and S.Y.R. Li. Two Special Cases of the Assignment Problem. *Discrete Mathematics*, 13: 129–142, 1975.

[Kuh55] H.W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1): 83–97, 1955.

[Lei91] C.E. Leiserson, 1991. Second quiz in the Introduction to Algorithms course, problem Q-3.

[MC80] C. Mead and L. Conway. *Introduction to VLSI systems*. Addison-Wesley, Reading, MA, 1980.

[MS91] O. Marcotte and S. Suri. Fast matching algorithms for points on a polygon. *SIAM Journal on Computing* 20: 405–422, 1991.

[Vai89] P.M. Vaidya. Geometry helps in matching. *SIAM Journal on Computing* 18: 1201–1225, 1989.

[Won91] C.K. Wong, 1991. Personal Communication.

[WPMK86] M. Werman, S. Peleg, R. Melter, and T.Y. Kong. Bipartite graph matching for points on a line or a circle. *Journal of Algorithms*, 7: 277–284, 1986.

[WPR84] M. Werman, S. Peleg, and A. Rosenfeld. A distance metric for multidimensional histograms. Technical Report CAR-TR-90, Center for Automation Research, Univ. of Maryland, August 1984.

it takes to encode this matching is simply the number of *distinct* demand rows and supply columns in this diagonal. The main computational step in the divide and conquer scheme is the algorithm to compute the *separating* row. Suppose that we are computing the separating row for a matrix B (this is the "blown" up matrix) of size $(b - t + 1) \times (r - \ell + 1)$. If all the rows in B are identical (they correspond to the same demand row in the original cost matrix A), then the min cost matching is a single diagonal, and hence the separating row will either be $t - 1$ or b (all the rows will match either to a diagonal $i \leq k$, or a diagonal $i > k$, where $k = \left\lfloor \frac{r+\ell}{2} \right\rfloor$). Recall that the diagonals form a bitonic sequence. Thus in time proportional to the number of distinct columns in B, we can figure out which side of k the cheapest diagonal is. When the rows in B are not identical, we only need to compute the value of s_x for a row that is different from its previous row. Hence the algorithm can be made to run in time proportional to the number of distinct rows and columns in B.

Acknowledgments: We would like to thank James Park for telling us about the skiers problem, and for informing us of the paper by Karp and Li [KL75]. The authors also thank Bill Pulleyblank for several useful discussions and for providing useful references.

References

[AHU83] A.V. Aho, J.E. Hopcroft and J.D. Ullman. *Data Structures and Algorithms.* Addison Wesley, 1983.

[AM86] A. Aggarwal and R.C. Melville. Fast computation of the modality of polygons. *Journal of Algorithms*, 7: 369–381, 1986.

[AMO89] R.K. Ahuja, T.L. Magnanti, and J. B. Orlin. Network flows. In *Optimization, Handbooks in Operations Research and Management Science*, pages 211–370. North-Holland Publishing, 1989.

[CLR90] T.H. Cormen, C.E. Leiserson and R.L. Rivest. *Introduction to Algorithms.* McGraw Hill, 1990.

[GTT89] A.V. Goldberg, É. Tardos, and R.E. Tarjan. Network flow algorithms. *Paths, Flows and VLSI-Layout*, pages 101–164. Springer Verlag, New York, 1990.

[Hof63] A.J. Hoffman. On simple linear programming problems. In V. Klee, editor, *Convexity: Proceedings of the Seventh Symposium in Pure Mathematics of the AMS*, volume 7, pages 317–327. American Mathematical Society, Providence, RI, 1963.

[Ja92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

optimal "rightmost" matching for B_{i+1} that contains M'; that is, M'' is the matching that contains M', and in each row $s_i < x \leq i+1$ the diagonal index is maximal among all optimal matchings for B_{i+1}. Consider the index of the match of row $s_i + 1$ in M''. If this index is at least $k+1$, then s_{i+1} is a separating row, and we are done.

Suppose that this is not the case, and that the match of row $s_i + 1$ lies to the left of diagonal $\mathcal{D}_{k+1}$. Let z be the diagonal index of the match of row $s_i + 1$. Let $s_i + 1 \leq q \leq i$ be the maximum index such that the match of row q in M'' lies in diagonal $\mathcal{D}_z$. Note that the minimum element in row q must be to the left or on $\mathcal{D}_z$. Otherwise, $\mathcal{D}_z(q)$ can be substituted by $\mathcal{D}_{z+1}(q)$ to obtain a valid matching that is no worse than M'', a contradiction to our assumption that M'' is the "rightmost" matching. On the other hand, because $V(s_i+1, s_i+1) \geq W(s_i+1, s_i+1)$, and because the rows are bitonic, it follows that the minimum element in row $s_i + 1$ must be to the right or on $\mathcal{D}_{k+1}$.

Consider the sub-array given by $\{\mathcal{D}_z(s_i+1, q), \ldots, \mathcal{D}_{k+1}(s_i+1, q)\}$. This sub-array conforms with the conditions of Lemma 17, implying that the sequence $\sum_{x=s_i+1}^{q} \mathcal{D}_z(x), \ldots, \sum_{x=s_i+1}^{q} \mathcal{D}_{k+1}(x)$ is bitonic. Note that $V(s_i+1, q) \geq W(s_i+1, q)$. From Lemma 17 we conclude that $\sum_{x=s_i+1}^{q} \mathcal{D}_z(x) \geq \sum_{x=s_i+1}^{q} \mathcal{D}_{z+1}(x)$. However, in this case the matching given by substituting $\mathcal{D}_z(s_i+1), \ldots, \mathcal{D}_z(q)$ of M'' by the corresponding elements that lie on $\mathcal{D}_{z+1}$ form a valid matching that is no worse from M'', a contradiction to our assumption that M'' is the "rightmost" matching.

■

The validity proof of the base recursion is trivial. The correctness of the algorithm follows from substituting $i = n$.

Theorem 19 *The algorithm described in Section 4.2 finds a minimum cost matching in a bitonic Monge array.*

4.4 The transportation problem

Suppose that we are given a transportation problem with integral supplies and demands, and a cost array A that is bitonic Monge. A simple way to transform this problem into a matching problem is by "blowing" the cost array into a $(\sum_{i=1}^{n} d_i) \times (\sum_{j=1}^{m} s_j)$ array, where the (i,j)th entry of the original array is replicated $d_i \times s_j$ times. Applying our algorithm for this array gives an $O((\sum_{i=1}^{n} d_i) \log(\sum_{j=1}^{m} s_j))$ time algorithm. However, the algorithm can be modified such that the amount of work per diagonal is $O(m+n)$, resulting in an $O(m \log(\sum_{j=1}^{m} s_j))$ time algorithm.

We outline the main modifications that are needed to the algorithm. The base case, when we have a single diagonal is quite easy: the matching is a single diagonal. The amount of time

Case 1.2: $V(i+1,i+1) \geq W(i+1,i+1)$, i.e., $\mathcal{D}_k(i+1) \geq \mathcal{D}_{k+1}(i+1)$. In this case the algorithm sets $s_{i+1} = i$. From the bitonicity of row $i+1$ it follows that the minimum entry in row $i+1$ lies in one of the diagonals $\mathcal{D}_{k+1}, \ldots, \mathcal{D}_{m-n+1}$. Hence, this minimum element can be added to M_i to form a matching for B_{i+1}, that costs less than any other matching containing M_i. Since it can be shown that there exists an optimal matching for B_{i+1} that contains M_i, $s_{i+1} = i$ is a separating row for B_{i+1}.

Case 2: $s_i < i$, that is, there exists an optimal matching for B_i the first s_i entries of which lie in diagonals $\mathcal{D}_1, \ldots, \mathcal{D}_k$, and the last $i - s_i$ entries of which lie in diagonals $\mathcal{D}_{k+1}, \ldots, \mathcal{D}_{m-n+1}$.

Case 2.1: $V(s_i+1,i+1) < W(s_i+1,i+1)$. In this case the algorithm sets $s_{i+1} = i+1$. Define M' to be the optimal matching for B_{i+1} for which the diagonal index of each row is minimal among all optimal matchings for B_{i+1}. We claim that such an optimal matching always exists and call it the "leftmost" matching. Consider the index of the match of row $i+1$ in M'. If this index is not greater than k, then $s_{i+1} = i+1$ is indeed a separating row.

Suppose this is not the case, and that the match of row $i+1$ lies in diagonal $\mathcal{D}_z$, for $z \geq k+1$. Let $q \leq i+1$ be the minimum index such that the match of row q in M' lies in diagonal $\mathcal{D}_z$. Note that the minimum element in row q must be to the right or on $\mathcal{D}_z$. Otherwise, $\mathcal{D}_z(q)$ can be substituted by $\mathcal{D}_{z-1}(q)$ to obtain a valid matching that is no worse than M', a contradiction to our assumption that M' is the "leftmost" matching. Since $s_i < i$, we have $V(s_i+1,i) \geq W(s_i+1,i)$. By the assumption $V(s_i+1,i+1) < W(s_i+1,i+1)$. Consequently, the element $\mathcal{D}_k(i+1)$ must be less than the element $\mathcal{D}_{k+1}(i+1)$. This and the bitonicity of the rows imply that the minimum element in row $i+1$ must be to the left or on $\mathcal{D}_k$.

Consider the sub-array given by $\{\mathcal{D}_k(q,i+1), \ldots, \mathcal{D}_z(q,i+1)\}$. This sub-array conforms with the conditions of Lemma 17, implying that the sequence $\sum_{x=q}^{i+1} \mathcal{D}_k(x), \ldots, \sum_{x=q}^{i+1} \mathcal{D}_z(x)$ is bitonic. Since M' is the "leftmost" matching, q must be greater than s_i. Since q has not been chosen as a separating line, $V(s_i+1,q) \geq W(s_i+1,q)$. It follows that $V(q,i+1) < W(q,i+1)$. The bitonicity of the diagonals (Lemma 17) implies that $\sum_{x=q}^{i+1} \mathcal{D}_{z-1}(x) \leq \sum_{x=q}^{i+1} \mathcal{D}_z(x)$. However, in this case the matching given by substituting $\mathcal{D}_z(q), \ldots, \mathcal{D}_z(i+1)$ of M' by the corresponding elements that lie on $\mathcal{D}_{z-1}$ form a valid matching that is no worse than M', a contradiction to our assumption that M' is the "leftmost" matching.

Case 2.2: $V(s_i+1,i+1) \geq W(s_i+1,i+1)$. In this case the algorithm sets $s_{i+1} = s_i$. Let M' be an optimal matching for B_{s_i} that lies in the first k diagonals. Let M'' be the

in row b is to the left or on $\mathcal{D}_\ell$ (i.e., the column index of the minimum element in row b is at most $\ell + b - 1$). Then, the sequence $U_\ell, \ldots, U_r$ is bitonic.

Proof: Consider three consecutive diagonals and denote their elements by $x_1, \ldots, x_c$, $y_1, \ldots, y_c$, and $z_1, \ldots, z_c$, respectively, where $c = b - t + 1$. Let $X = \sum_{i=1}^{c} x_i$, $Y = \sum_{i=1}^{c} y_i$, and $Z = \sum_{i=1}^{c} z_i$. We claim that $Y - X \leq Z - Y$. Consequently, the sequence of the differences between the diagonals is a monotonic non-decreasing sequences and, therefore, the sequence of diagonals is bitonic.

It remains to prove the above claim. For all i, $1 \leq i \leq c - 1$, the Monge property gives: $y_i + y_{i+1} \leq z_i + x_{i+1}$, or, $y_{i+1} - x_{i+1} \leq z_i - y_i$. We can write $Y - X$ and $Z - Y$ as follows:

$$Y - X = (y_1 - x_1) + (y_2 - x_2) + (y_3 - x_3) + \cdots + (y_c - x_c)$$

$$Z - Y = (z_1 - y_1) + (z_2 - y_2) + \cdots + (z_{c-1} - y_{c-1}) + (z_c - y_c)$$

Therefore, $(Y - X) - (Z - Y) \leq (y_1 - x_1) - (z_c - y_c)$. Because of the bitonicity of the rows and since the minimum in row t is to the right or on $\mathcal{D}_r$, $y_1 - x_1$ is negative. Similarly, $z_c - y_c$ is positive. Hence, $(Y - X) - (Z - Y) \leq 0$. ■

Now, we prove the correctness of the recursive step. To simplify the notation, we consider the top level of the recursion. Let B_i be the sub-array containing the first i rows of B.

Lemma 18 *For all $1 \leq i \leq n$, row s_i of B_i is a separating row for the sub-array B_i.*

Proof: The proof is by induction on i. For the base case $i = 1$, and s_1 is either 1 or 0, depending on the values of $V(1,1)$ and $W(1,1)$. (By comparing the two values we know where the minimum of row 1 is.) Now, assume that row s_i of B_i is a separating row for B_i, and we show that row s_{i+1} of B_{i+1} is a separating row for B_{i+1}.

Case 1: $s_i = i$; that is, there exists an optimal matching for B_i that lies in the first $k = \lfloor \frac{m+n}{2} \rfloor$ diagonals. Denote this matching by M_i.

Case 1.1: $V(s_i + 1, i + 1) = V(i + 1, i + 1) < W(s_i + 1, i + 1) = W(i + 1, i + 1)$, i.e., the element $\mathcal{D}_k(i + 1)$ is less than the element $\mathcal{D}_{k+1}(i + 1)$. In this case, the algorithm sets $s_{i+1} = i + 1$. To obtain a contradiction suppose that there is no optimal matching for B_{i+1} in diagonals $\mathcal{D}_1, \ldots, \mathcal{D}_k$. Consider an optimal matching for B_{i+1}. By our assumption it must be that the match in row $i+1$ lies in diagonal $\mathcal{D}_z$ for some $z > k$. It can be shown that there exists such an optimal matching M' that contains the matching M_i. However, the matching M_i can be augmented by $\mathcal{D}_k(i + 1) = a[i + 1, i + k]$, since column $i + k$ does not intersect the first i rows of diagonals $\mathcal{D}_1, \ldots, \mathcal{D}_k$, since row $i + 1$ is bitonic and since $\mathcal{D}_k(i + 1) < \mathcal{D}_{k+1}(i + 1)$ and $\mathcal{D}_k(i + 1) < \mathcal{D}_z(i + 1)$. Thus, the matching given by adding $\mathcal{D}_k(i + 1)$ to M_i is smaller than M'. A contradiction.

and let $W(i_1, i_2) = \sum_{i=i_1}^{i_2} \mathcal{D}_{k+1}(i)$.

$$s_x \leftarrow \begin{cases} t-1+x & \text{if} V(s_{x-1}+1, t-1+x) < W(s_{x-1}+1, t-1+x) \\ s_{x-1} & \text{otherwise} \end{cases}$$

The justification for the definition of s_x is provided in Lemma 18.

We claim that each stage takes constant time. To see this, notice that if $s_{x-1} = t-2+x$, then

$$\begin{aligned} V(s_{x-1}+1, t-1+x) &= \mathcal{D}_k(t-1+x) \\ W(s_{x-1}+1, t-1+x) &= \mathcal{D}_{k+1}(t-1+x) \end{aligned}$$

Otherwise,

$$\begin{aligned} V(s_{x-1}+1, t-1+x) &= V(s_{x-1}+1, t-2+x) + \mathcal{D}_k(t-1+x) \\ W(s_{x-1}+1, t-1+x) &= W(s_{x-1}+1, t-2+x) + \mathcal{D}_{k+1}(t-1+x). \end{aligned}$$

Therefore, the separating row can be found in $O(b-t)$ time, since it takes constant time to compute each $V(s_{x-1}+1, t-1+x)$ and $W(s_{x-1}+1, t-1+x)$.

Starting with an array B consisting of diagonals $\mathcal{D}_1, \ldots, \mathcal{D}_{m-n+1}$, we get the following recurrence relation for the running time of our algorithm. Let $T(a, b)$ denote the running time of the algorithm on a matrix with a rows and b diagonals.

$$T(n, m-n+1) = T\left(s, \left\lfloor \frac{m-n}{2} \right\rfloor + 1\right) + T\left(n-s, \left\lceil \frac{m-n}{2} \right\rceil\right) + O(n).$$

It is not hard to see that $T(n, 1) = O(n)$ since in this case the minimum cost matching is simply the given diagonal. The solution of the recurrence implies the following theorem.

Theorem 16 (Complexity) *The time complexity of the algorithm is $O(n \log m)$.*

The algorithm can be parallelized [Ja92]. The parallel time complexity is $O(\log^2 m)$ time with $n/\log m$ processors on an EREW PRAM.

4.3 Correctness

The correctness proof is based on the following lemma. For $1 \leq \ell < r \leq m-n+1$, consider the sub-array B that consists of the elements in $\{\mathcal{D}_\ell(t, b), \ldots, \mathcal{D}_r(t, b)\}$. Let $U_j = \sum_{i=t}^{b} \mathcal{D}_j(i)$.

Lemma 17 *Suppose that the minimum element in row t is to the right or on $\mathcal{D}_r$ (i.e., the column index of the minimum element in row t is at least r) and that the minimum element*

to be the ith element in this diagonal, i.e., $\mathcal{D}_j(i) = a[i, j+i-1]$. Define $\mathcal{D}_j(t, b)$ to be elements of $\mathcal{D}_j$ between rows t and b, i.e., $\mathcal{D}_j(t, b)$ consists of entries $a[t, j+t-1], a[t+1, j+t], \ldots, a[b, j+b-1]$.

4.2 The algorithm

Let A be a bitonic Monge array. From Lemma 15, it follows that there exists a minimum cost matching in A that only contains elements from the set $\{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_{m-n+1}\}$. Consequently, instead of considering the entire array, we may consider only the set of elements that belong to these diagonals.

The algorithm uses the "divide and conquer" paradigm. The input to each step of the recursion is an array B of size $(b - t + 1) \times (r - \ell + 1)$ that contains elements in the set $\{\mathcal{D}_\ell(t, b), \ldots, \mathcal{D}_r(t, b)\}$. Define k to be the index of the middle diagonal of B, $k = \left\lfloor \frac{r+\ell}{2} \right\rfloor$. The output is a *separating* row s, $t \leq s \leq b$, that together with the middle diagonal of B, splits B into four quadrants such that there exists a minimum cost matching for B that is inside the top left quadrant and the bottom right quadrant (quadrants II and IV in Figure 4.2). In other words, there exists a minimum cost matching for B in which rows $t, \ldots, s$ are matched with entries in $B' = \{\mathcal{D}_\ell(t, s), \ldots, \mathcal{D}_k(t, s)\}$, and rows $s + 1, \ldots, b$ are matched with entries in $B'' = \{\mathcal{D}_{k+1}(s + 1, b), \ldots, \mathcal{D}_r(s + 1, b)\}$. Since quadrants II and IV do not share any columns of B it follows that the minimum cost matching for B can be found by two recursive calls: one to an array B' of size $(s - t + 1) \times (k - \ell + 1)$ and one to an array B'' of size $(b - s) \times (r - k)$.

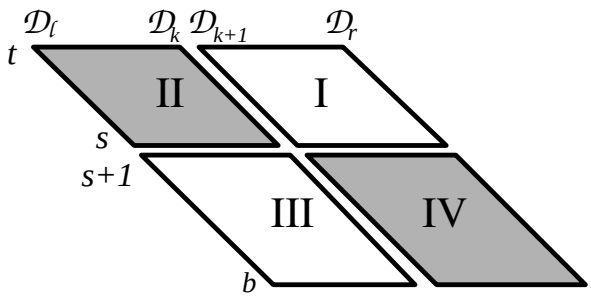

Figure 5: Quadrant *II* includes row s and diagonal k;
quadrant IV includes row $s + 1$ and diagonal $k + 1$.

The algorithm for finding the separating row in B has $(b - t + 1)$ stages. In stage $x \geq 1$, we process row $(t - 1 + x)$ to determine the separating row, s_x, for the subarray given by rows $t, \ldots, t-1+x$. Initially, $s_0 = t-1$ and finally $s = s_{b-t+1}$. In stage $x \geq 1$, the separating row s_x is set either to $t-1+x$ or to s_{x-1}, according to the following criterion. Let $V(i_1, i_2) = \sum_{i=i_1}^{i_2} \mathcal{D}_k(i)$,

Theorem 14 *The transportation problem on a line can be solved in $O((m+n)\log(m+n)$ time.*

Proof: All the procedures described in this section involve operations on "entities" of length bounded by $O(m+n)$. These operations require constant time except for the sorting and the priority queue operations. However, sorting can be done in $O((m+n)\log(m+n)$ time and each priority queue operation can be done in $O(\log(m+n))$ time. The theorem follows since there are constant number of such procedures. ■

4 Minimum cost matching in bitonic Monge arrays

In this section we describe an $O(n \log m)$ time algorithm for finding a minimum cost perfect matching in bitonic Monge arrays. Recall that an $n \times m$ array $A = \{a[i,j]\}$ is *Monge* if for all $1 \leq i_1 < i_2 \leq n$ and $1 \leq j_1 < j_2 \leq m$, $a[i_1,j_1] + a[i_2,j_2] \leq a[i_1,j_2] + a[i_2,j_1]$. An $m \times n$ array $A = \{a[i,j]\}$ is (row) *bitonic* Monge if A is Monge and each row of A is bitonic. That is, if $a[i,j]$ is the minimal entry in row i, then $a[i,1], \ldots, a[i,j]$ is a monotonic non-increasing sequence and $a[i,j], \ldots, a[i,m]$ is a monotonic non-decreasing sequence.

4.1 Preliminaries and notations

A matching in an $n \times m$ array $A = \{a[i,j]\}$ is a sequence of n entries, $a[1,j_1], \ldots, a[n,j_n]$, where $j_p \neq j_q$ for all $1 \leq p < q \leq n$. A minimum cost matching is a matching where $\sum_{i=1}^{n} a[i,j_i]$ is minimized. The following lemma defines the structure of one of the minimum cost matchings in a Monge array.

Lemma 15 *There exists a minimum cost matching in a Monge array A, such that the entries form a generalized diagonal; that is, for all $i_1 < i_2$, if $a[i_1,j_1]$ and $a[i_2,j_2]$ are two entries in the matching, then $j_1 < j_2$.*

Proof: Consider a minimum cost matching M for A, and suppose that M includes two entries $a[i_1,j_2]$ and $a[i_2,j_1]$ such that $i_1 < i_2$ and $j_1 < j_2$. By the Monge property, $a[i_1,j_1]+a[i_2,j_2] \leq a[i_1,j_2]+a[i_2,j_1]$. Thus, the matching M' obtained by replacing the entries $a[i_1,j_2]$ and $a[i_2,j_1]$ of M by $a[i_1,j_1]$ and $a[i_2,j_2]$ costs at most the same as M. If M' costs less than M, we have a contradiction. Otherwise, M' is a different minimum matching. We can continue in this manner until the resulting matching forms a generalized diagonal. ■

Following Lemma 15, it is easy to see that the case $n = m$ is solved by just taking the main diagonal as the solution. However, the problem is not so simple when $n < m$. The best known algorithm when $n < m$ is a dynamic programming algorithm that runs in time $O(mn)$.

Let A be an $n \times m$ bitonic Monge array. For $1 \leq j \leq m-n+1$, define $\mathcal{D}_j$ to be the jth (full) diagonal in A; that is, $\mathcal{D}_j$ consists of the entries $a[1,j], a[2,j+1], \ldots, a[n,j+n-1]$. Define $\mathcal{D}_j(i)$

Step 1.3. Find the leftmost and rightmost bundles in NEG (operations FINDLEFT(NEG) and FINDRIGHT(NEG)). Let $\langle j, q\rangle$ and $\langle j, r\rangle$ be the leftmost and rightmost points of node j that are covered by NEG.

Step 1.4. Find the left partners of points $\langle j, q\rangle, \ldots, \langle j, r\rangle$ using L. For these left partners, delete their right-partner edges from R and then delete the left-partner edges of $\langle j, q\rangle, \ldots, \langle j, r\rangle$ from L.

Implementation of Step 2: This step consists of the following substeps.

Step 2.1. Find the leftmost point in j that has a right partner. This can be done by a binary search in R. Let this point be $\langle j, q\rangle$.

Step 2.2. Check if $\langle j, q\rangle$ is covered by a bundle in $SET(j)$ (operation FINDPOINT($SET(j), q$)). If not, then all the chains in $SET(j)$ end at j, and we move to Step 3. Otherwise, find the rightmost point covered by a bundle in $SET(j)$ (operation FINDRIGHT($SET(j)$). Let this point be $\langle j, r\rangle$, where $r \geq q$.

Steps 2.3 to 2.8 are iterated.

Step 2.3. Find the right partner of $\langle j, q\rangle$. This can be done by a binary search in R. Suppose that this right partner is in node i. Using L find the rightmost point $\langle j, q'\rangle$ the right partner of which is in i.

Step 2.4. Find the successor of $\langle j, q\rangle$. This can be done by a binary search in S. Suppose that this successor is in node j'. Using S find the rightmost point $\langle j, q''\rangle$ the successor of which is in node j'. Let $p = \min\{q', q'', r\}$.

Step 2.5. Find the bundle B that covers $\langle j, p\rangle$ (operation FINDPOINT($SET(j), p$)). If $\langle j, p\rangle$ is not the rightmost point in its bundle replace B by two bundles B and B' where B covers all the points up to $\langle j, p\rangle$ and B' covers the rest of the points in node j originally covered by B. (This is done by INSERT and DELETE operations.)

Step 2.6. Split $SET(j)$ into two at bundle B (operation SPLIT($SET(j), B, SUCC, SET(j)$)).

Step 2.7. Let c be the cost of the edge from $\langle j, q\rangle$ to its right partner minus the cost of the edge from this partner to the successor of $\langle j, q\rangle$. Add c to all bundles in $SUCC$ (operation ADDCOST($SUCC, c$)) and add $SUCC$ to $SET(j')$. Note that all the bundles in $SUCC$ are to the right of the bundles currently in $SET(j')$, hence, this can be done using the operation UNION($SET(j'), SUCC, SET(j')$).

Step 2.8. If $p < r$, then let $q = p + 1$ and repeat the iteration starting at Step 2.3.

Implementation of Step 3: The implementation is very similar to Substeps 2.3 to 2.7 where $r = d_j$ and $\langle j, q\rangle$ is initialized to be the leftmost point of node j which does not have a left partner but has a right partner, if such exists.

We maintain each set $SET(j)$ in a priority queue. The ordering among the prefix-bundles in $SET(j)$ is determined by the left-to-right ordering of these prefix-bundles' roots. We know that the ordering on the roots induces the same left-to-right ordering on the points the prefix-bundles cover in node j (from Lemma 12) and a decreasing ordering on the prefix-bundles' shifting costs (from Lemma 13). The priority queue supports the following operations:

- INSERT(PQ, B): insert bundle B into PQ.
- DELETE(PQ, B): delete bundle B from PQ.
- FINDLEFT(PQ): find the leftmost bundle in PQ.
- FINDRIGHT(PQ): find the rightmost bundle in PQ.
- FINDPOINT(PQ, q): find the bundle in PQ that covers point $\langle j, q\rangle$. (We assume that priority queue PQ consists of bundles of prefixes of j.)
- FINDCOST(PQ, c): find the rightmost bundle in PQ whose shifting cost is greater or equal c.
- ADDCOST(PQ, c): add c to the shifting costs of all the bundles in PQ.
- SPLIT(PQ, B, PQ_1, PQ_2): split the priority queue PQ into two priority queues: PQ_1 that stores all the bundles that are to the left of bundle B, and PQ_2 that stores the rest.
- UNION(PQ_1, PQ_2, PQ): Union the priority queue PQ_1 with the priority queue PQ_2 and store the union at PQ. It is assumed that both PQ_1 and PQ_2 consist of bundles of prefixes of j, and that all chains in PQ_1 are to the left of all chains in PQ_2.

Several data structures can be used to implement this priority queue so that each operation can be done in logarithmic time in the number of bundles, that is $O(\log(m+n))$. In particular, the *augmented red-black tree* described in [CLR90] (cf. Chapters 14–15) is suitable.

Initially, priority queues $SET(j)$ for all source nodes j are set to be the empty set. We now turn to the implementation of the *Unbalanced-chains-matching algorithm.*

Implementation of Step 1: This step consists of the following substeps.
Step 1.1. Find the rightmost bundle B in $SET(j)$ with non-negative shifting cost (operation FINDCOST$(SET(j), 0)$).
Step 1.2. Remove all bundles with negative shifting cost from $SET(j)$ (operation SPLIT$(SET(j), B, SET(j), NEG)$).

As with R and L, we can show that the size of S is $O(m+n)$ and that it can be computed in linear time using R and L.

Before giving the implementation details of the **Unbalanced-chains-matching algorithm**, we need a few technical lemmas. As with chain-bundles, the leftmost node of each chain is called the *root* of the chain. Given two chains C_1 and C_2 we say that C_1 is *to the left of* C_2 (and C_2 is *to the right of* C_1) if the root of C_1 is to the left of the root of C_2.

Lemma 12 *Let C_1 and C_2 be two chains that visit source node j at points $\langle j,q\rangle$ and $\langle j,p\rangle$, respectively. If C_1 is to the left of C_2 then $\langle j,q\rangle$ is to the left of $\langle j,p\rangle$, i.e., $q<p$. (See Figure 4.)*

Proof: Consider a planar embedding of chains C_1 and C_2 as shown in Figure 4. The chain C_1 partitions the line into intervals corresponding to each of its edges. There are two types of intervals: source–sink intervals that correspond to edges from a left source node to a right sink node, and sink–source intervals. We claim that the root of chain C_2 is in a source–sink interval of C_1. In other words, the closest point of chain C_1 to the right of the root of chain C_2 must be a sink point. Otherwise, the root of chain C_2 (which is a source point) must have a left partner, yielding a contradiction.

It follows from the definition of partners that the embedded chains never cross each other. Hence, all points of chain C_2 are in source–sink intervals of C_1. In particular, $\langle j,p\rangle$ is in such an interval and thus must be to the right of $\langle j,q\rangle$. ■

Lemma 13 *Let C_1 and C_2 be two chains, where C_1 is to the left of C_2. Suppose that (i) a source point $\langle j_1,q\rangle \in C_1$ is to the left of a source point $\langle j_2,r\rangle \in C_2$, (ii) the left partner of $\langle j_2,r\rangle$ (if such exists) is to the left of $\langle j_1,q\rangle$, and (iii) the shifting costs of* `Prefix`(C_1,j_1') *for all source nodes j_1' to the left of j_1 (including j_1 itself) are non-negative. Then, the shifting cost of* `Prefix`(C_2,j_2) *is less than or equal to the shifting cost of* `Prefix`(C_1,j_1).

Proof: Let $\langle j_1',q'\rangle$ be the closest point of C_1 to the left of the root of C_2. (This point is a source point.) We claim that the shifting cost of the sub-chain of C_1 from j_1' to j_1 is greater than or equal to the shifting cost of `Prefix`(C_2,j_2). Note that by assumption (iii), the shifting cost of `Prefix`(C_1,j_1') is non-negative, and thus, the lemma follows from the claim. The shifting cost of a chain is given by the total length of its source–sink intervals minus the total length of the its sink–source intervals. Recall that all points of chain C_2 are in source–sink intervals of C_1. Also, all points of chain C_1 are in sink–source intervals of C_2. This implies that: (1) the total length of source–sink intervals of `Prefix`(C_2,j_2) is less than or equal to the total length of source–sink intervals of the sub-chain of C_1 from j_1' to j_1; and (2) the total length of sink–source intervals of `Prefix`(C_2,j_2) is greater than or equal to the total length of sink–source intervals of the sub-chain of C_1 from j_1' to j_1. The claim follows. ■

Step 1: If $SET(j)$ is not empty, remove all the prefix-bundles with negative shifting costs from $SET(j)$. By Lemma 11 the leftmost unmatched points of all the chains in these prefix-bundles are in j or to the right of j. Thus, all the sink points to the left of j in these prefix-bundles can be matched to the left. Consequently, for each such prefix-bundle B that covers points $\langle j, q\rangle$ through $\langle j, r\rangle$ we delete the edges connecting $\langle j, q\rangle$ through $\langle j, r\rangle$ to their left partners. Once these edges are removed from B, what remains of B to the left of (but not including) node j is simply a balanced chain-bundle (ending at the sink node which contains the left partners of $\langle j, q\rangle$ through $\langle j, r\rangle$). The removal of edges also "frees up" points q through r of node j which will become the starting points of the new unbalanced chains. (These points are processed in Step 3.)

Step 2: If $SET(j)$ is not empty, then the remaining prefix-bundles in the set all have non-negative shifting costs. We "extend" each of these prefix-bundles to the next source node (in case such exists) as follows. Consider a prefix-bundle B in $SET(j)$. We break B into sub-prefix-bundles according to the right partners and successors of the source points in j covered by B; that is, all the chains in B that visit the same sink node and then the same source node after visiting node j form a sub-bundle. Suppose that the successors of the source points in j covered by B are in nodes $j_1, \ldots, j_k$. We insert all the sub-bundles that end at j_ℓ, into $SET(j_\ell)$, for $1 \leq \ell \leq k$, after updating their shifting costs. Specifically, for such a sub-bundle B_ℓ, which visits i after j and then visits j_ℓ after i, we update the shifting cost by adding the cost of the edge (j, i) and subtracting the cost of edge (i, j_ℓ).

Step 3: We create new bundles corresponding to the points in j that have no left partners but have right partners (these include points made free in Step 1). We treat all these points as one bundle and basically repeat Step 2 for this bundle.

Note that Steps 2 and 3 guarantee that when we reach node j the set $SET(j)$ indeed consists of all the bundles of the prefixes of j. After the scan is completed we are left with some balanced chains whose rightmost point is sink, and hence are to be matched to their left. The rest of the chains have to be matched to their right. We can compute this matching in two more scans, one in each direction (right-to-left and then left-to-right) similar to the way it was done for the balanced chains.

Implementation details: In addition to R and L, we pre-compute a list of successors for all source points. We explicitly maintain a list S which contains all those source points $\langle j, q\rangle$ and their successors for which either (1) $q = 1$, i.e., it is the leftmost point in its node, or (2) $q > 1$ and the successor of $\langle j, q-1\rangle$ is in a different source node. We keep S sorted lexicographically.

- f: the total cost of the edges from sink nodes to the right of j to their right partners.

The definition of node j implies that $b \leq a$ and $b + d > a + c$. Consequently, $d > c$. If $\langle k, t \rangle$ is the unmatched point, the cost of matching the sink points of chain C is $a + d + f$ and it is $a + c + f$ if $\langle j, q \rangle$ is the unmatched point. Since $d > c$ it follows that $\langle j, q \rangle$ is a better unmatched point; a contradiction. ■

Using this lemma we can find the leftmost unmatched point of a chain C as follows. Initially, we make the leftmost point of C a candidate for the unmatched point. We scan C from left to right and compute the shifting cost of $\texttt{Prefix}(C, j)$, for each source node j in C. Whenever we encounter a node j such that the shifting cost of $\texttt{Prefix}(C, j)$ is negative, we set $\langle j, q \rangle$ to be the new candidate (where $\langle j, q \rangle$ is the point in j that is in C), and "cut" $\texttt{Prefix}(C, j)$ from C, i.e., consider C as if it starts at $\langle j, q \rangle$. The candidate at the end of the scan is the leftmost unmatched point of C. The naive implementation of this scan is not polynomial; the **Unbalanced-chains-matching algorithm** gives an efficient implementation.

Define *prefixes* of a source node j to be the set of all prefixes $\texttt{Prefix}(C, j)$ such that the chain C visits j. Define a *prefix-bundle* of j to be a maximal collection of the prefixes of j that start at the same node and visit the same nodes up to j. Note that a prefix-bundle of j may consist of several chain-bundles, in the case when all these chain-bundles visit the same nodes up to j. Define $SET(j)$ to be the set of prefix-bundles of j. We now give a high level description of the algorithm for finding the leftmost unmatched points of all chain-bundles; the implementation details follow.

We scan the source nodes from left to right. When we are scanning a source node j, we know $SET(j)$. For each prefix-bundle in $SET(j)$ we maintain the following information:

1. The leftmost node of the prefixes in the prefix-bundle. (All of these prefixes start at the same node.)
2. The number of prefixes in the prefix-bundle.
3. The points in j covered by the prefixes in the prefix-bundle.
4. The shifting cost of the prefixes in the prefix-bundle. (All of these prefixes have the same shifting cost.)

Before describing the scan procedure, we need one more definition. Define the *successor* of a source point $\langle j, q \rangle$ to be the source point that is the right partner of the right partner of $\langle j, q \rangle$ (if they exist).

Unbalanced-chains-matching algorithm:

by B, where $p < d_i$, Consider the alternating chain C that starts at $\langle j, q-1\rangle$. Its next point must be $\langle i, p+1\rangle$. That is, the first two nodes visited by C are the same as the first two nodes of B. Consider the first node k visited by C that is not in B. Such a node must exist since chain C is not in the chain-bundle B. We have two cases:

Case 1: k is a sink node. Let $\langle k, t\rangle$ be the point of k that is in C, and let $\langle x, r\rangle$ be the left partner of $\langle k, t\rangle$. Since up to node k chain-bundle B and chain C visited the same nodes, $\langle x, r+1\rangle$ is covered by B. Let $\langle \ell, s\rangle$ be the right partner of $\langle x, r+1\rangle$. Since $\langle \ell, s+1\rangle$ is not the right partner of $\langle x, r\rangle$, we must have that s is the rightmost point of ℓ. We charge B to this node.

Case 2: k is a source node. Let $\langle k, t\rangle$ be the point of k that is in C, and let $\langle x, r\rangle$ be the left partner of $\langle k, t\rangle$. Similar to Case 1 we have that $\langle x, r-1\rangle$ is covered by B. Since $\langle k, t+1\rangle$ is not the right partner of $\langle x, r-1\rangle$, we must have that t is the rightmost point of k. We charge B to this node.

Case 2 is the only one in which B is charged to an extreme point of a node that it does not cover. This extreme point is a rightmost point of a source node. Assume that chain-bundle B' covers this point. Since a chain-bundle is never charged to a rightmost point of a source node that it covers, it follows that B and B' are charged to different extreme points. No other chain-bundle is charged to this rightmost point by our charging process. Therefore, each extreme point is charged at most once. ■

Let C be an alternating chain that visits source node j. Denote the prefix of C that ends at j by $\texttt{Prefix}(C, j)$. The *shifting cost* of $\texttt{Prefix}(C, j)$ is defined to be the total cost of the edges from sink nodes in $\texttt{Prefix}(C, j)$ to their left partners minus the total cost of the edges from these sink nodes to their right partners.

Let j be the leftmost node of C such that the shifting cost of $\texttt{Prefix}(C, j)$ is negative.

Lemma 11 *The leftmost unmatched point of C is either in j or at a node to the right of j.*

Proof: Let $\langle j, q\rangle$ be the point in j that is in C. To obtain a contradiction assume that the leftmost unmatched point of C is a point $\langle k, t\rangle$, where $k < j$. Define the following quantities for the chain C:

- a: the total cost of the edges from sink nodes to the left of k to their left partners.
- b: the total cost of the edges from sink nodes to the left of k to their right partners.
- c: the total cost of the edges from sink nodes between k and j to their left partners.
- d: the total cost of the edges from sink nodes between k and j to their right partners.
- e: the total cost of the edges from sink nodes to the right of j to their left partners.

3.3 The unbalanced chains

We now consider the unbalanced alternating chains. Recall that in any solution to the transportation problem each unbalanced chain consists of exactly one source point that is unmatched. All the sink points to the left of this point (if any) are matched to their left partners and all the sink points to its right (if any) are matched to their right partners. The computation of the unmatched point can be done independently for each chain. Note that more than one unmatched point may exist for a given chain; each such point defines a different solution, each of which has the same value. Our solution finds the leftmost unmatched point of every chain.

We define a *chain-bundle* B to be a maximal collection of chains that visit the same nodes (and also start and end at the same nodes). The leftmost node of each chain-bundle is called the *root* of the chain-bundle. Note that if node i is in B, then the points of i "covered" by B, i.e., the points of i that are in chains that belong to B, are consecutive.

Example revisited: in Figure 4 we have four chain-bundles of unbalanced chains (from left to right):

Chain-bundle 1: $\langle 1, 2 \cdots 3\rangle \longrightarrow \langle 1, 9 \cdots 10\rangle \longrightarrow \langle 3, 1 \cdots 2\rangle$

Chain-bundle 2: $\langle 1, 4 \cdots 6\rangle \longrightarrow \langle 1, 6 \cdots 8\rangle \longrightarrow \langle 3, 3 \cdots 5\rangle \longrightarrow \langle 2, 1 \cdots 3\rangle \longrightarrow \langle 4, 1 \cdots 3\rangle$

Chain-bundle 3: $\langle 1, 7 \cdots 8\rangle \longrightarrow \langle 1, 4 \cdots 5\rangle \longrightarrow \langle 4, 4 \cdots 5\rangle$

Chain-bundle 4: $\langle 2, 1 \cdots 2\rangle \longrightarrow \langle 1, 2 \cdots 3\rangle \longrightarrow \langle 4, 6 \cdots 7\rangle$

Consider two chains C_1 and C_2 that belong to the same chain-bundle B. Since the edge costs of both chains are the same, the leftmost unmatched point of both chains is in the same node. Thus, when determining the unmatched points we may consider all the chains in one chain-bundle together. This is helpful since although the number of alternating chains is proportional to $\sum_{i=1}^{n} d_i$ the number of chain-bundles is $O(m+n)$, as shown below.

Lemma 10 *The number of chain-bundles is at most* $2(m+n)$.

Proof: To prove the lemma we "charge" every chain-bundle to an extreme point of a node (either a leftmost or a rightmost point) so that no extreme point is charged more than once. Consider a chain-bundle B. If B covers the leftmost point of its root node $\langle j, 1\rangle$, then we charge B to this leftmost point. Else, let node i be the sink node following the root node in B. If B covers $\langle i, d_i\rangle$, the rightmost point of node i, then we charge B to this rightmost point. Suppose that B does not cover either of these extreme points. Let $\langle j, q\rangle$ be the leftmost point of node j that is covered by B, where $q > 1$, and let $\langle i, p\rangle$ be the rightmost point of node i that is covered

chains whose leftmost point is sink. For the balanced chains whose rightmost point is sink the computation is similar.

We scan the sink nodes from left to right. When we reach a sink node i with demand d_i, we find the sequence X of (consecutive) points of node i that do *not* have left partners. To find X, we find element $\langle i, 1\rangle$ in L and then perform a linear scan of L until we find the element $\langle i, p\rangle$ whose left partner is nil. If no such $\langle i, p\rangle$ exists, then all the points of i have a left partner; if such an element is found, then the sequence of points $X = \langle i, p\rangle, \ldots, \langle i, d_i\rangle$ have no left partners and they must be matched to their right partners. In the latter case, we output sink points $\langle i, p\rangle$ and $\langle i, d_i\rangle$ with their right partners (located by binary-search in R) along with any other sink points $\langle i, p'\rangle$ (and their right partners) that appear in R between points $\langle i, p\rangle$ and $\langle i, d_i\rangle$. Denote by Y the set of right partners of points in X. Once points of X are matched to points of Y, we need to update R and L to reflect the fact that points of Y are no longer "available" as left partners. In other words, in future processing, no sink point whose left partner is a source point in Y can match to this point. This updating is performed as follows. For every source point q of Y, we first delete from R the edge from q to its right partner, denoted by point r, and then delete from L the edge from r to its left partner (i.e. to q). We cannot afford to do this deletion for *every* point in Y. Instead, we keep Y implicitly in the same format as R and L, and we perform deletions of sequences of points in a manner similar to the above procedure for finding the right matches of X. This process matches all the sink points that are in alternating chains whose leftmost point is sink.

Lemma 9 *Matching all the sink points in balanced chains can be done in $O(m + n)$ time.*

Proof: The time required for the scan is linear in the number of nodes and the size of R, which is $O(m + n)$. ■

Example revisited: In the example considered earlier (Figure 4), there are no alternating chains whose leftmost point is sink, there are four alternating chains whose rightmost point is sink (from right to left):

1. $\langle 3, 4\rangle, \langle 4, 8\rangle, \langle 1, 1\rangle, \langle 2, 3\rangle$,
2. $\langle 3, 3\rangle, \langle 4, 9\rangle$,
3. $\langle 3, 2\rangle, \langle 4, 10\rangle$,
4. $\langle 3, 1\rangle, \langle 4, 11\rangle$.

When we scan the nodes from right to left we will first match all the points in node 3 to their left and then point $\langle 1, 1\rangle$ to $\langle 2, 3\rangle$.

We now describe how to compute R using a stack. Each item on the stack is a pair $[i, x]$, where $1 \leq i \leq n$, and $1 \leq x \leq d_i$. Item $[i, x]$ corresponds to the cluster of x sink points starting at $\langle i, 1\rangle$. We scan the sorted list of nodes from left to right. When we encounter a sink node i with demand d_i, we push the item $[i, d_i]$ onto the stack. When we encounter a source node j with supply s_j, we pop from the stack items that correspond to s_j sink points (if they exist) and match them to the source points of node j. This is done as follows. Let q denote the current supply of node j. Initially, $q = s_j$. The current q source points of node j are the rightmost ones; that is, points $s_j - q + 1$ through s_j. Iteratively, we pop an item $[i, x]$ from the stack. If $x \leq q$ then we match the x sink points starting at $\langle i, 1\rangle$ to the x current leftmost points of source node j, that is, to points $s_j - q + x, \ldots, s_j - q + 1$ of node j. Consequently, we add the sink point $\langle i, 1\rangle$ and its right partner $\langle j, s_j - q + x\rangle$ to the list, and decrease q by x. If $x > q$ then we match the q rightmost sink points of node i; that is points $\langle i, x - q + 1\rangle, \ldots, \langle i, x\rangle$ to points $s_j, \ldots, (s_j - q + 1)$ of source node j. Consequently, we add the sink point $\langle i, x - q + 1\rangle$ and its right partner $\langle j, s_j\rangle$ to the list, push $[i, x - q]$ back on to the stack, and set q to zero. If at any point q becomes zero or the stack becomes empty, we consider the next node on the sorted list of nodes. To finish the computation of R, we sort it lexicographically. A similar process computes L.

Lemma 7 *We can compute R in $O((m + n)\log(m + n))$ time.*

Proof: Since the number of push and pop operations is proportional to the size of R, it follows that the computation of the unsorted list requires $O(m + n)$ time. The sorting is done in $O((m + n)\log(m + n))$ time. ■

All the procedures and proofs described in this section would hold if we augment R and L to contain similarly selected source points and their partners. Thus, in the following sections we will assume that R and L contain right/left partners of both sink and source points.

3.2 The balanced chains

A chain C, as before, is an *alternating chain of points.* Recall that the transportation problem is equivalent to the minimum cost matching problem on the points. The following lemma is implied by Lemmas 1 and 2.

Lemma 8 *There exists a nested solution to the transportation problem in which the demand of every sink point is supplied by either its left partner or its right partner.*

If a chain's leftmost (resp. rightmost) point is a sink point, then all the red points on that chain are supplied by their right (resp. left) partners. Below, we consider only alternating

We will explicitly maintain a list R that contains all those sink points $\langle i, p\rangle$ and their right partners for which either (1) $p = 1$, i.e., it is the leftmost point in its node, or (2) $p > 1$ and the right partner of $\langle i, p-1\rangle$ is in a different source node. We keep R sorted lexicographically. The analogous left-partner list is L.

Lemma 6 *The size of R is bounded by $m + n$.*

Proof: The number of sink points in R for which $p = 1$ is n. For each sink point in R with $p > 1$, its right partner must be the rightmost point in a source node. There are only m rightmost points in source nodes. ■

Figure 4 gives an example of the list.

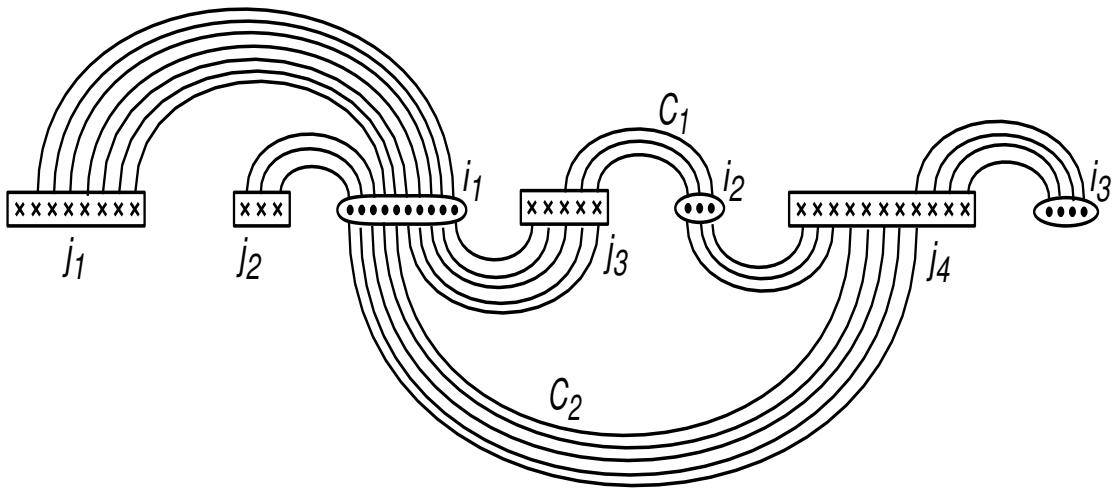

Figure 4: i_k is a sink node with the demand indicated by the number of dots. j_ℓ is a source node with the supply indicated by the number of x's. The edges above the nodes connect sink nodes to their left partners; the edges below the nodes connect sink nodes to their right partners.

Sink Point	$\langle 1,1\rangle$	$\langle 1,6\rangle$	$\langle 2,1\rangle$	$\langle 3,1\rangle$
Right Partner	$\langle 4,8\rangle$	$\langle 3,5\rangle$	$\langle 4,3\rangle$	nil

Given R, to compute the right partner of a sink point $\langle i, p'\rangle$ we search for the rightmost sink point $\langle i, p\rangle$ in R such that $p \le p'$. Let $\langle j, q\rangle$ be the right partner of $\langle i, p\rangle$. It follows that the right partner of $\langle i, p'\rangle$ is $\langle j, q - (p' - p)\rangle$. Since R is sorted, the search for the right partner of a point takes time logarithmic in the size of R; that is, $O(\log(m + n))$. An analogous result holds for searching out the left partner of a point.

3 Transportation problem on a line

We now consider the transportation problem, when the sources and sinks are on a straight line with integer supplies and demands. The algorithm for this problem runs in $O((m+n)\log(m+n))$ time where the number of source nodes is m, and the number of sink nodes is n. We assume that d_i is the integer demand at the i-th sink node and s_j is the integer supply at the j-th source node[1] and that $\sum_{i=1}^{n} d_i \leq \sum_{j=1}^{m} s_j$. We describe the algorithm for the line, since it is simpler. The algorithm for the circle is similar.

View each sink node i (resp. each source node j) as a "cluster" of d_i sink (resp. s_j source) points placed infinitesimally close to each other, and ordered from left to right. From now on, we use the terminology *node* and *point* in accordance with this distinction. We denote the p-th sink point in the i-th node by the pair $\langle i, p\rangle$, for $1 \leq i \leq m$ and $1 \leq p \leq d_i$. source points are denoted similarly. Note that the lexicographic order of the pairs corresponds to the left to right ordering of the points.

It is easy to see that our transportation problem is equivalent to the problem of finding a minimum cost matching of the sink points. Thus, our previous algorithm can be applied to obtain a solution for the transportation problem. However, this algorithm is not polynomial since its running time is proportional to $\sum_{j=1}^{m} s_j$. Below, we show a polynomial algorithm for the problem.

The algorithm consists of three stages. In the first stage we compute left and right partners for the sink points. In the second stage we define alternating chains similar to the matching algorithm. There are two kinds of alternating chains: balanced chains that have the same number of sink and source points, and unbalanced chains that have an extra source point. In the second stage we solve the problem for all the balanced chains. In the third (and most involved) stage we deal with the unbalanced chains. Note that the number of partners and alternating chains is proportional to $\sum_{i=1}^{n} d_i$, and thus extra care has to be taken in order to make all these stages polynomial.

3.1 Computing the partners

We define left partners and right partners of the sink points exactly as before. We use the term partner for the source points as well; for example, if sink point $\langle i, p\rangle$ has source point $\langle j, q\rangle$ as its right partner, then the left partner of source point $\langle j, q\rangle$ is the sink point $\langle i, p\rangle$. Note that each source point has at most *one* left partner and *one* right partner. In the first stage of the algorithm we compute the partners. To keep the running time polynomial, we compute and maintain partners implicitly, in a way that allows efficient retrieval.

[1] In this section, i will always be associated with sink nodes and j with source nodes.

Note that the theorem holds when the chain is a cycle, i.e., $v_0 = v_k$. In this case, the unmatched source point in the chain is either v_0 or v_k (which are the same).

Mincost-matching-on-a-circle algorithm:

Step 1: Sort all the source and sink points (together) in some direction around the circle, say clockwise.

Step 2: Compute ℓ_p and r_p for each sink point p. We show how to compute the r_p's using a stack (finding the ℓ_p's is analogous). Starting with any sink point, we proceed clockwise around the circle. Any time we encounter a sink point, we push it onto the stack. Any time we encounter a source point q, we pop the last sink point p from the stack and set $r_p = q$. If at any point the stack becomes empty, we push the clockwise-next sink point onto it, ignoring any source points that this action causes us to skip.

Step 3: Compute all the alternating chains. This can be done by keeping links between the sink points and their partners. Starting with any sink point that is not already contained in a chain, we simply trace the left and right links until we encounter the source endpoints of the chain.

Step 4: For each alternating chain $C = \langle v_0, u_1, v_1, u_2, v_2, \ldots, u_k, v_k \rangle$, decide which source point is unmatched as follows. We compute M_i^C, which is the cost of the matching with the source point v_i being unmatched. We first compute M_0^C, which is simply $\sum_{i=1}^{k} d(u_i, v_i)$. Given M_i^C, we can compute M_{i+1}^C by subtracting the cost of edge (u_{i+1}, v_{i+1}) from M_i^C and adding the cost of (u_{i+1}, v_i).

It takes $O(k)$ time to compute M_0^C (where k is the length of the chain), and $O(k)$ time to compute all the other M_i^Cs. Thus, overall time is O(k). If i^* is the index that gives the smallest value for M_i^C, the points u_i on C with $i \leq i^*$ match to their left partners, and the other sink points match to their right partners.

The correctness of this algorithm follows from Theorem 4. As for time complexity, each step, except for sorting the points, can be done in linear time. We obtain the following theorem.

Theorem 5 (Complexity) *The algorithm runs in linear time if the points are given sorted in some direction along the circle.*

All the steps in our algorithm can be done in parallel in $O(\log m)$ time with m processors on an EREW PRAM. We need to use prefix computations, list ranking, and the computation of left and right partners. All these operations can be done in the stated bounds using techniques described in [Ja92].

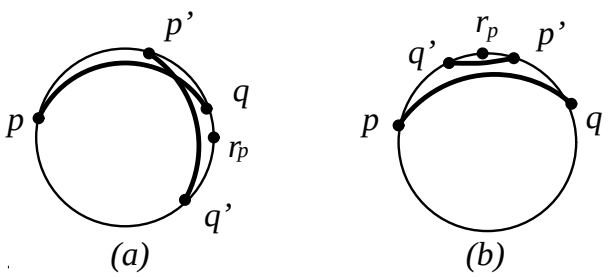

Figure 3: Thicker arcs between points represent the edges of the matching.

1. $\{u_1, u_2, \ldots, u_k\} \subseteq Sinks$ and $\{v_0, v_1, v_2, \ldots, v_k\} \subseteq Sources$;
2. the source point v_0 is the left partner of u_1 and the source point v_k is the right partner of u_k;
3. the source point v_i is the left partner of u_{i+1} and the right partner of u_i, for each $1 \leq i \leq k-1$; and
4. the chain C is maximal.

Note that since C is maximal, either v_0 is not a right partner and v_k is not a left partner of any sink point, or v_0 is the same as v_k. Also note that in the case of points on a line, a chain may not necessarily start at a source point.

Lemma 3 *Each sink point belongs to a unique alternating chain, and any two chains are disjoint.*

Proof: Consider a source point v_i. Suppose that two sink points u_a and u_b have v_i as their left partner and that we have v_i, u_a, u_b in clockwise order. Between u_a and v_i there is an equal number of sink and source points. If v_i is the left partner of u_b, then due to the presence of u_a (a sink point) there must be an excess source point between u_a and u_b. In this case, this source point must be the left partner of u_b, yielding a contradiction. Consequently, only *one* sink point can have v_i as its left partner. The same property is true for right partners. This observation implies the lemma. ■

The following theorem is implied by Lemma 2 and Lemma 3 and is the key for our algorithm.

Theorem 4 *Consider the points of a single chain in a nested minimum cost matching. There is one unmatched source point in the chain. All the sink points to the left of this source point are matched to their left partners, and all the sink points to the right of this source point are matched to their right partners.*

uncrossed (p,q) and (p',q') and, hence, the number of edge crossings in M' must be at least one fewer that in M. The following case analysis proves the above lemma.

Case 2.1: (s,t) crosses only (p',q), i.e. $s,t \in cw(p,q')$. W.l.o.g, assume $s \in cw(p',q)$ and $t \in ccw(p',q)$. If $t \in cw(p,p')$, then (s,t) crosses (p',q'). Otherwise, $t \in cw(q,q')$ and (s,t) crosses (p,q).

Case 2.2: (s,t) crosses only (p,q'), i.e. $s,t \notin cw(p',q)$. W.l.o.g, assume $s \in cw(p,q')$, and $t \in ccw(p,q')$. If $s \in cw(p,p')$, then (s,t) crosses (p,q). Otherwise, $s \in cw(q,q')$ and (s,t) crosses (p',q').

Case 2.3: (s,t) crosses both (p',q) and (p,q'). W.l.o.g, assume $s \in cw(p',q) \cap cw(p,q') = cw(p',q)$ and $t \in ccw(p,q') \cap ccw(p',q) = ccw(p,q')$. In this case (s,t) crosses both (p,q) and (p',q').

■

For $p \in Sinks$, define the *left partner* (respectively, *right partner*) of p, denoted by ℓ_p (respectively, r_p), to be the first source point q counterclockwise (respectively, clockwise) from p, such that going counterclockwise (respectively, clockwise) from p to q on the circle, there are *as many* sink points *as* source points.

Lemma 2 *In a nested minimum cost perfect matching, every sink point is matched to either its left partner or its right partner.*

Proof: (By contradiction.) W.l.o.g., suppose that $(p,q) \in M$, $x(p,q) = cw(p,q)$, and $q \neq r_p$.

Case 1: $r_p \notin x(p,q)$. (See Figure 3(a).) From the definition of the right partner, there must be a sink point $p' \in x(p,q)$ such that $(p',q') \in M$ and $q' \notin x(p,q)$. Then (p,q) crosses (p',q'), a contradiction.

Case 2: $r_p \in x(p,q)$. (See Figure 3(b).) Since there is an equal number of sink and source points in $x(p,r_p)$, one source point q' in $x(p,r_p)$ or r_p itself, is either not matched or matched with some p' not in $x(p,r_p)$. If q' is not in M, then $M - \{(p,q)\} + \{(p,q')\}$ costs less than M, a contradiction. If $(p',q') \in M$, then $p' \in x(r_p,q)$ since M is nested. Therefore, $M - \{(p,q),(p',q')\} + \{(p,q'),(p',q)\}$ gives a cheaper matching than M, a contradiction.

■

Define the chain $C = \langle v_0, u_1, v_1, u_2, v_2, \ldots, u_k, v_k \rangle$ to be an *alternating chain* of points if the following conditions hold:

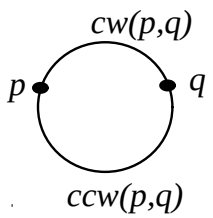

Figure 1: In this figure, $x(p,q) = cw(p,q)$.

crossed edges of M to get another minimum cost perfect matching with fewer edge crossings. Let $(p,q) \in M$ and $(p',q') \in M$ be some crossed pair of edges, where $p, p' \in Sinks$ and $q, q' \in Sources$. Consider a matching $M' = M - \{(p,q),(p',q')\} + \{(p,q'),(p',q)\}$, which is simply the matching M with (p,q) and (p',q') uncrossed. First, we show that the cost of M' is less than or equal to the cost of M.

Case 1: $q' \in x(p,q)$ and $q \in x(p',q')$. (See Figure 2(a).) We obtain a contradiction by showing that M' costs less than M. This is because
$d(p,q') + d(p',q) = [d(p,q) - d(q',q)] + [d(p',q') - d(q',q)] < d(p,q) + d(p',q')$.

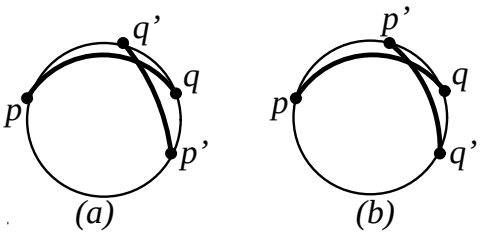

Figure 2: Thicker arcs between points represent the edges of the matching.

Case 2: $p' \in x(p,q)$ and $q \in x(p',q')$. (See Figure 2(b).) Here we show that the cost of M' is at most the cost of M.

$$\begin{aligned} & d(p,q) + d(p',q') \\ & \quad = [d(p,p') + d(p',q)] + [d(p',q) + d(q,q')] \\ & \quad = [d(p,p') + d(p',q) + d(q,q')] + [d(p',q)] \\ & \quad \geq d(p,q') + d(p',q). \end{aligned}$$

We now argue that whenever M' costs the same as M in Case 2, then M' has fewer edge crossings than M. In particular, we show that if some edge $(s,t) \in M$ crosses either (p',q) or (p,q'), it must have crossed either (p,q) or (p',q'). This completes the proof, since we have

skiers so that the sum of the absolute differences of the heights of each skier and his/her skis is minimized. There is a fairly simply $O(nm)$ dynamic programming algorithm for this problem. The algorithm by Karp and Li [KL75], as well as our algorithm, solves this problem in $O(m \log m)$ time.

All our results use the quadrangle inequality in a crucial manner. Because of the many applications, we hope that this paper will generate more interest towards the understanding of minimum cost bipartite matching and transportation for useful special cases. For example, the three important problems that we were not able to solve are: (i) An efficient computation of the minimum cost matching when the cost array is (staircase) Monge; a special case of this is when the sink and source points form a convex polygon. (ii) An efficient computation of the minimum cost matching when the sink and source points form a simple polygon. (iii) The transportation problem when the underlying cost array is Monge and the total supply is greater than the total demand.

2 Minimum cost matching on a circle

In this section we present our algorithm for the minimum cost perfect matching problem when the sink points and the source points lie on a circle and the cost function is the Euclidean distance along the arcs of the circle. For the sake of simplicity, we assume that all the points are distinct. The algorithm and all the proofs are analogous for points on a line. Furthermore, these results hold for points on any curve homeomorphic to a circle or a line, as long as the cost function is the distance along the curve. The results also hold for the case where all the points are vertices of a convex unimodal polygon and the cost is the Euclidean distance between points.

Consider a circle with sink and source points. Any two points p and q split the circle into two arcs (see Figure 1), one going clockwise from p to q, and one going counterclockwise from p to q. Let $cw(p,q)$ $(respectively, ccw(p,q))$ be the clockwise (respectively, counterclockwise) arc from p to q. Let $x(p,q)$ refer to the shorter of $cw(p,q)$ and $ccw(p,q)$. The distance between p and q, $d(p,q)$, is defined to be the length of $x(p,q)$.

We say that two edges in the matching M, $(p,q) \in M$ and $(p',q') \in M$, *cross* each other if $x(p,q) \cap x(p',q') \neq \emptyset$, $x(p,q) \not\subseteq x(p',q')$, and $x(p',q') \not\subseteq x(p,q)$. We call a matching *nested* if no two edges in the matching cross each other. In other words, a matching is nested if for any two edges in the matching: $(p,q) \in M$ and $(p',q') \in M$, $x(p,q) \cap x(p',q') \neq \emptyset$ implies that either $x(p,q) \subseteq x(p',q')$ or $x(p',q') \subseteq x(p,q)$.

Lemma 1 *There exists a nested minimum cost perfect matching for points on a circle.*

Proof: Let M be any minimum cost perfect matching. We show how to uncross any pair of

is that the total supply is greater than or equal to the total demand.

The remainder of this section lists the main results of this paper and their applications.

First we consider the case in which the sink and source points lie on a curve that is homeomorphic to either a line or a circle and the cost function is the Euclidean distance along the curve. For this case, we give a linear time algorithm for computing the minimum cost matching that is simpler than the one given by Karp and Li [KL75]. Our algorithm can be extended to the corresponding transportation problem, where each source point has integral supply and each sink point has integral demand. For this transportation problem we develop an algorithm with an $O((m+n)\log(m+n))$ running time. This improves on the $O((m+n)^2)$ time algorithm of Karp and Li [KL75].

Our matching algorithm for the points on the circle can also be applied to the case where the sink and the source points form the vertices of a convex *unimodal* polygon [AM86]. A polygon is *unimodal* if for every vertex, the distances from it to all other vertices form a sequence that is first non-decreasing and then non-increasing when the polygon is traversed in clockwise order starting at that vertex. As an application, we note that the case when the points lie on the circle arises in pattern recognition when feature sets of different objects are compared. We refer the reader to [WPR84, WPMK86] for a detailed discussion of this application.

Next we consider the case in which the cost array forms an $n \times m$ bitonic *Monge* array. An array is *bitonic* if the entries in every row form a monotone non-increasing sequence that is followed by a monotone non-decreasing sequence. For the bitonic Monge array, we present an $O(n \log m)$ time algorithm for computing a minimum cost matching. Note that the cost array for the sorted sink and source points on a line forms a bitonic Monge array (observe that the cost array for the points on a circle is *not* bitonic Monge). As a matter of fact, this is true even for a more general case where the sink points lie on one straight line and the source points lie on another straight line (that is not necessarily parallel to the first one). Hence this result can be applied to the problem of connecting power lines (or connecting clock lines) in VLSI river routing [MC80, Won91]. In this problem, n terminals that lie on a straight line need to be connected to any n of the m power (or clock) terminals that lie on a parallel line, and the total amount of wire used should be minimized.

Our results for bitonic Monge arrays can be applied to the transportation problem. We provide a weakly polynomial algorithm for the transportation problem when the associated cost array is a bitonic Monge array. Our algorithm for this problem runs in $O(m \log(\sum_{j=1}^{m} s_j))$ time.

Our interest in the matching problem was partly sparked by the following problem that appeared on an exam for an algorithms course taught by C. E. Leiserson [Lei91]: given m pairs of skis with heights $s_1, \ldots, s_m$ and $n \leq m$ skiers with heights $t_1, \ldots, t_n$, assign skis to

1 Introduction

Given a complete bipartite graph $G = (Sinks \cup Sources, Sinks \times Sources)$, where $|Sinks| = n$, $|Sources| = m$ and $n \leq m$, a *perfect matching* on G is a subset of the edges $M \subseteq E$ such that for all sink nodes p, exactly one edge of M is incident to p and for all source nodes q, at most one edge of M is incident to q. Given a cost function defined on the edges of G, a *minimum cost perfect matching* on G is a perfect matching on G of minimum cost.

The best known algorithm for this problem for arbitrary bipartite graphs (with $m+n$ nodes and mn edges) takes $O(n(mn + m \log m))$ time. This time complexity can be achieved by using the Hungarian method [Kuh55, AMO89, GTT89]. Since this problem has a relatively high time complexity, researchers have investigated special cases. For example, Vaidya [Vai89] showed that the minimum cost perfect matching among $2n$ points in the Euclidean plane can be computed in $O(n^{5/2} \log^4 n)$ time. For the case where $n = m$, the sink and source points lie on a convex polygon (respectively, simple polygon), and the distance between two points is simply the Euclidean distance, Marcotte and Suri [MS91] showed that a minimum cost matching can be computed in $O(n \log n)$ time (respectively, $O(n \log^2 n)$ time). Note that these papers assume that $n = m$, in spite of the fact that the situation $n < m$ arises in many matching problems. Karp and Li [KL75] considered the $n \leq m$ case. They investigated the case where the points lie on either a line or a circle, and the cost function is given by the Euclidean distance along the corresponding curve. For this problem, they obtained a linear time algorithm for minimum weight matching, assuming that the points are given in sorted order. (See also [WPMK86].)

In the transportation problem, each sink point i has demand d_i, and each source point j has supply s_j, where $\sum_{i=1}^{n} d_i \leq \sum_{j=1}^{m} s_j$. A *feasible transportation* is an assignment of supplies from source points to sink points such that all the demands are satisfied. The cost of moving one unit of supply from a source point to a sink point is given by the cost function defined on the edges. A *minimum cost transportation* is a feasible one with minimum cost.

Hoffman [Hof63], following the French mathematician Gaspard Monge (1746–1818), considered a special case of the transportation problem in which the cost array forms an $n \times m$ *Monge* array. (The entry $a[i, j]$ of the cost array is the cost of the edge between sink point i and source point j.) An $n \times m$ array $A = \{a[i, j]\}$ is *Monge* if for all i_1, i_2, j_1, and j_2 satisfying $1 \leq i_1 < i_2 \leq n$ and $1 \leq j_1 < j_2 \leq m$,

$$a[i_1, j_1] + a[i_2, j_2] \ \leq \ a[i_1, j_2] + a[i_2, j_1] \ .$$

Hoffman [Hof63] showed that if the cost array associated with a transportation problem is Monge and the total supply is the *same* as the total demand, then a simple greedy algorithm solves the transportation problem in linear time. However, in reality, a more reasonable assumption

Efficient Minimum Cost Matching and Transportation Using Quadrangle Inequality

Alok Aggarwal * Amotz Bar-Noy * Samir Khuller †
Dina Kravets ‡ Baruch Schieber *

Abstract

We present efficient algorithms for finding a minimum cost perfect matching, and for solving the transportation problem in bipartite graphs, $G = (Sinks \cup Sources, Sinks \times Sources)$, where $|Sinks| = n$, $|Sources| = m$, $n \leq m$, and the cost function obeys the quadrangle inequality. First, we assume that all the sink points and all the source points lie on a curve that is homeomorphic to either a line or a circle and the cost function is given by the Euclidean distance along the curve. We present a linear time algorithm for the matching problem that is simpler than the algorithm of [KL75]. We generalize our method to solve the corresponding transportation problem in $O((m+n)\log(m+n))$ time, improving on the best previously known algorithm of [KL75].

Next, we present an $O(n \log m)$-time algorithm for minimum cost matching when the cost array is a bitonic Monge array. An example of this is when the sink points lie on one straight line and the source points lie on another straight line Finally, we provide a weakly polynomial algorithm for the transportation problem in which the associated cost array is a bitonic Monge array. Our algorithm for this problem runs in $O(m \log(\sum_{j=1}^{m} s_j))$ time, where d_i is the demand at the ith sink, s_j is the supply available at the jth source, and $\sum_{i=1}^{n} d_i \leq \sum_{j=1}^{m} s_j$.

*IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.
Email: {aggarwa,amotz,sbar}@watson.ibm.com.

†Dept. of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. Email: samir@cs.umd.edu. Research supported by NSF Research Initiation Award CCR-9307462. Part of this research was done while this author was visiting the IBM T. J. Watson Research Center and was affiliated with UMIACS and partly supported by NSF grants CCR-8906949, CCR-9111348 and CCR-9103135.

‡Dept. of Computer Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102. Email: dina@cis.njit.edu. Supported by the NSF Research Initiation Award CCR-9308204 and the New Jersey Institute of Technology SBR grant #421220. While at MIT, this research was supported by the Air Force under Contract AFOSR-89-0271, the Defense Advanced Research Projects Agency under Contracts N00014-87-K-825 and N00014-89-J-1988.