Consecutive Interval Query and Dynamic Programming on Intervals

Alok Aggarwal[1] and Takeshi Tokuyama[2]

[1] IBM Research Division, T. J. Watson Research Center, P.O.Box 218, Yorktown Heights, NY 10598

[2] IBM Research Division, Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamato-shi, Kanagawa, 242 Japan

Abstract. Given a set of n points on a line and a set of m weighted intervals defined on these points, we consider a particular dynamic programming problem on these intervals. If the weights are all nonnegative or all nonpositive, we solve this dynamic programming problem efficiently by using matrix searching in Monge arrays, together with a new query data structure which we call the *consecutive interval query* structure. We invoke our algorithm to obtain fast algorithms for the sequential partition of a graph and for the partial clique covering of an interval graph.

1 Introduction

Let U be a set of n integers $\{1, 2, .., n\}$ that can be regarded as a set of points on a line, and let Z be a set of m intervals which have their endpoints in U. We associate a weight $w(I)$ with each interval. Let $W(i,j) = \sum_{I \subset (i,j]} w(I)$. Assume that there exists a non-decreasing sequence $f(i); i = 1, 2, .., n-1$ such that $i \leq f(i) \leq n$. We define $\tilde{W}(i,j) = W(i,j)$ if $i \leq j \leq f(i)$ and ∞, otherwise. The problem is to compute two functions $D(i)$ and $E(i)$ on U, where it is assumed that $D(i)$ can be computed in $O(1)$ time from $E(i)$, $E(1) = 0$, and $E(i)$ is defined by the following recurrence:

$$E(i) = \min_{j>i} \{D(j) + \tilde{W}(i,j)\}. \tag{1}$$

In [1], Asano showed several problems in computational geometry and graph algorithms can be solved by using the recurrence given above and he also gave an $O(m \log n)$ time algorithm for solving this recurrence. (Note that his algorithm is worse than the simple dynamic programming algorithm that takes $O(n^2)$ time, when $m > n^2/\log n$.) His algorithm needs a solution of the *interval query* problem that can be defined as follows: "Given a set of m weighted intervals, preprocess the set so that the total weight of the intervals contained in a query interval I can be computed efficiently". A well-known data structure called the *interval tree* answers the query in $O(\log n)$ time after using $O(m \log n)$ time in preprocessing. Thus, the preprocessing time dominates the total time complexity of Asano's dynamic programming algorithm.

In this paper, we solve the above dynamic programming problem by using a different approach. Our algorithm runs in $O(m + n \log\log n)$ time for the concave version of this problem and in $O(m + n \log n)$ time for the convex version; the problem is called concave (convex) if all weights are non-negative (resp. non-positive). Our algorithm uses a data structure for solving a different query problem (*the consecutive query problem*) in conjuction with searching in Monge arrays [2, 3, 10]; this data structure is sufficient for solving our dynamic programming problem but not for solving the general interval query problem. Finally, we use our algorithms to obtain efficient algorithms for the following problems:

Sequential partition of graph (Figure 1). Let G be an undirected weighted graph on n nodes and m edges, where m may be much larger than n. Let the nodes be numbered according to a specified order. The problem is to partition these nodes into subsets such that each subset consists of contiguous nodes, each subset contains at most K nodes, and the total weight of edges connecting nodes in different subsets is minimized.

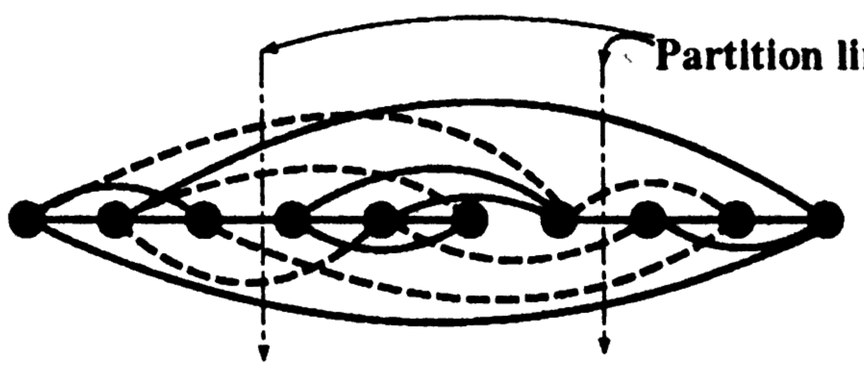

Fig. 1. Sequential partition of graph, n=10, K=4. Solid edges have weight 2, and broken edges have weight 1.

Partial clique covering of interval graph (Figure 2). Given a set of n points on a line and a set of m weighted intervals on these points, let G be the associated *interval graph*. Given a number K, compute K subgraphs of G so that the total weight of the intervals associated with the nodes in the union of the subgraphs is maximized.

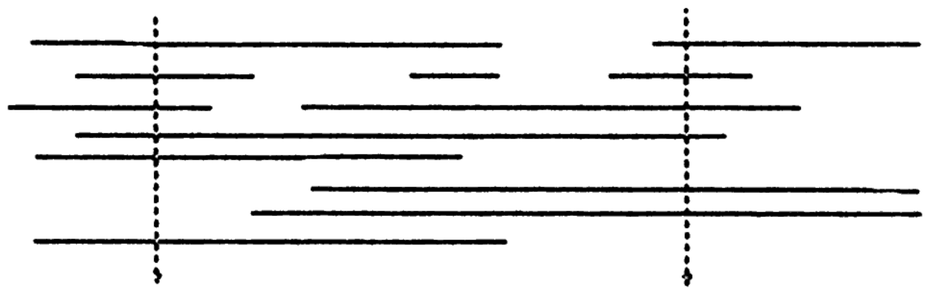

Fig. 2. Partial clique cover of interval graph, K=2

2 Consecutive Interval query

The interval query problem is defined as follows:

Given two integers i and j such that $1 \leq i < j \leq n$, compute the total weight $W(i,j)$ of intervals of Z that are covered by the closed interval $[i,j]$.

Many $O(\log n)$ query-time and $O(m \log n)$ preprocessing-time data structures for the interval query problem are known in the literature, e.g., the *segment tree* and the *orthogonal range tree* [12]. However, for these data structures, the $O(m \log n)$ preprocessing time dominates the time complexity when $m \gg n$. We can, indeed, improve the preprocessing time by providing a data structure that answers the query in $O(n^{\epsilon})$ query time and $O(m)$ preprocessing time. However, such a data structure will improve the time bound of Asano's algorithm only when $m \geq n^{1+\epsilon}$. Similarly, we can provide a data structure that answers a query in $O(\log^2 n / \log\log n)$ query time and $O(m \log n / \log\log n)$ preprocessing time but this also improves Asano's algorithm for only certain values of m and n. Keeping this in view, we restrict our attention to the dynamic programming problem at hand, and show in section 3 that the sequence of queries that we need, have some nice properties. In particular, this sequence of queries can be answered by solving the consecutive query problem given below. Consequently, in this section, we describe a data structure that can be constructed using $O(m)$ preprocessing time and that answers the *consecutive query* problem efficiently; more precisely, we answer the query $W(i+i_0, j+j_0)$ in $O(\frac{i_0+j_0}{\log^c n} + c \log n)$ time (where c is a non-negative constant, and can be 0) as long as we have already queried $W(i,j)$ before. This data structure is constructed using *fractional cascading* [5].

The consecutive query problem: Query $W(i+i_0, j+j_0)$ efficiently after we have queried $W(i,j)$.

We first describe a data structure for consecutive query in $O(i_0+j_0)$ time with $O(m)$ preprocessing time. We map an interval $I = [(i,j)]$ to a point $p(I) = (i,j)$ in the $n \times n$ planar grid G. Then, we obtain a set $S(Z)$ of m points in the grid G. An interval $J = [a,b]$ is contained the interval $I = [i,j]$ if and only if $a \geq i$ and $b \leq j$; in other words, $p(J)$ is located in the South-East of $p(I)$. Thus, the problem is reduced to a South-East rectilinear range search problem in a grid (usually, the problem is referred as the North-West search, after a reflection transformation). The value $W(i+i_0, j) - W(i,j)$ is simply the total weight of the points located in the rectangle whose corners are (i,n), (i,j), $(i+i_0,n)$, and $(i+i_0,j)$. Let R_k be the set of points located on the k-th row of the grid. We equip the sorted list of R_k with respect to the ordinate for each $k = 1,2,..,n$. This can be obtained in $O(n+m)$ time as follows:

We compute the sorted list of m points with respect to the ordinate; this can be done in $O(m+n)$ time by using bucket sorting. Then, we distribute the list into R_1,.., R_k, to obtain the sorted list of each subset. For each element x of a list, we store the sum of the weight of its elements that are greater than x with respect to the sorted order. Now, it is easy to compute $W(i+i_0, j) - W(i,j)$ in $i_0 \log n$ time by locating j in each of $R_{i+1}, .., R_{i+i_0}$. To avoid consuming $\log n$

time for searching in each list, we adapt the *fractional cascading* method of [5], which attains $O(\log n + i_0)$ query-time with $O(m+n)$ preprocessing-time.

Theorem 1. *[5] Let G be an undirected graph with maximum node degree d. Suppose a sorted list of elements in an totally ordered set A is associated with each node of G. Let s be the total size of these lists. Then, with $O(s)$ preprocessing time we can construct a data structure such that we can locate any given element a of A in all lists on a path p of length l of G in $O(l \log d + \log s)$ time.*

In our case, the underlying graph G is the path $v_1, v_2, .., v_n$, and the list R_k is the one associated with v_k. Next, an augmented list $\tilde{R}_k$ is constructed for each list R_k, and the location of j in R_k can be computed from that in $\tilde{R}_k$ in $O(1)$ time. Further, the location of j in $\tilde{R}_{k+1}$ is known from that in $\tilde{R}_k$ in $O(1)$ time by using the *bridge* between $\tilde{R}_k$ and $\tilde{R}_{k+1}$. Since the method is well-known, we omit the details and refer the readers to Cole [6] and Chazelle-Guibas [5]. We call above structure the *row-structure*. Note that in such a data structure, when we compute $W(i+i_0, j) - W(i,j)$, we spend $O(\log n)$ time for locating j in the initial list $\tilde{R}_{i+1}$, and the remaining computation is done in $O(i_0)$ time. We give a similar structure (called *column-structure*) for computing $W(i+i_0, j+j_0) - W(i+i_0, j)$. The sorted list of the points in the k-th column is denoted by C_k, and the associated augmented list is denoted by $\tilde{C}_k$. Using the above arguments, it is clear that we can compute $W(i+i_0, j+j_0) - W(i,j)$, and hence $W(i+i_0, j+j_0)$, in $O(i_0 + j_0 + \log n)$ time. We show below that we can remove the $\log n$ term for the consecutive search problem.

When we query $W(i,j)$, we remember not only the value $W(i,j)$ but also both the position of j in $\tilde{R}_i$ and the position of i in $\tilde{C}_j$. When we compute $W(i+i_0, j+i_0)$, we can find the location of j in $\tilde{R}_{i+1}$ from that in $\tilde{R}_i$ in $O(1)$ time. Thus, it takes $O(i_0)$ time for computing $W(i+i_0, j) - W(i,j)$. Besides, we can find the location of $i+i_0$ in $\tilde{C}_j$ in $O(i_0)$ time since it suffices to move up the list $\tilde{C}_j$ from the location of i, at most $O(i_0)$ steps. Therefore, we can compute $W(i+i_0, j+j_0) - W(i+i_0, j)$ in $O(i_0 + j_0)$ time. Furthermore, we can simultaneously compute the location of $i+i_0$ in $\tilde{C}_{j+j_0}$ and that of $j+j_0$ in $\tilde{R}_{i+i_0}$. Hence, we obtain the following:

Theorem 2. *A data structure can be built that answers the consecutive query $W(i+i_0, j+j_0)$ after querying $W(i,j)$ in $O(i_0 + j_0)$ time; this structure can be constructed in $O(m)$ preprocessing time. If we only know the value $W(i,j)$, then this consecutive query can be answered in $O(i_0 + j_0 + \log n)$ time by using the same data structure.*

We construct an $O(\frac{i_0+j_0}{(\log n)^c} + \log n)$ query time data structure for any given constant c. Let $L = \lfloor \log n \rfloor$, and $h(s) = \lfloor \frac{n}{L^s} \rfloor$ for $s = 1, 2, .., c$. For each $k = 1, 2, .., h(s)$, we consider the set $R(s)_k$ of the points of $S(Z)$ whose abscissa is in the interval $[(k-1)L^s, kL^s)$ if $s \neq 0$, where $R(0)_k = R_k$. For each s and k, we sort $R(s)_k$ with respect to the ordinate values, and store the sorted list. If more than two points has the same ordinate value, we sort

them with respect to the x-coordinate value; it is easy to see the sorting can be done in $O(m+n)$ time. We construct a fractional cascading structure for the set of lists $\{R(s)_0, .., R(s)_{h(s)} | s = 0, 1, .., c\}$. The underlying graph has nodes $\{v(s)_1, v(s)_2, .., v(s)_{h(s)} | s = 0, 1, .., c\}$. The edges are given between $(v(s)_i, v(s)_{i+1}))$ for $i = 1, 2, .., h(s) - 1$ and $s = 0, 1, .., c$, and $(v(s)_k, v(s-1)_{kL})$ for $k = 1, 2, .., h(s)-1$ and $s = 1, 2, ..c$, where the list $R(s)_j$ is associated with $v(s)_j$. We denote $\tilde{R}(s)_i$ for the augmented list of $R(s)_j$.

Lemma 3. *By using the above data structure, $W(i+i_0, j) - W(i, j)$ can be computed in $O(\frac{i_0}{(\log n)^c} + \log n)$ time.*

Proof. Since there is a path of length at most $\frac{ci_0}{(\log n)^c} + c \log n$ from $v(i)$ to $v(i+i_0)$ in the underlying graph, the above lemma follows from Theorem 1. □

By constructing a similar structure for the columns, we can similarly compute $W(i+i_0, j+j_0) - W(i+i_0, j)$. Hence, we have the following theorem:

Theorem 4. *A data structure can be constructed such that it answers the query $W(i+i_0, j+j_0)$ in $O(\frac{i_0+j_0}{(\log n)^c} + c \log n)$ time as long as the query $W(i, j)$ has been answered before; this data structure can be constructed in $O(m+n)$ time; where the constant c is non-negative but can be zero.*

3 Dynamic programming on intervals

3.1 Dynamic programming and Matrix searching

In this section, we show how to solve the dynamic programming on intervals by using above data structure. Let us give a brief summary of dynamic programming on concave (or convex) functions.

A matrix $A = (A(i,j))_{i,j=1,2,..,n}$ is called a convex Monge (concave Monge) matrix if $A(i,j) + A(i+1, j+1) \geq A(i, j+1) + A(i+1, j)$ (resp. $A(i,j) + A(i+1, j+1) \leq A(i, j+1) + A(i+1, j)$) for $1 \leq i, j \leq n$. A matrix A is called a staircase matrix if $A(i,j)$ is ∞ unless $i < j < f(i)$ for some non-decreasing sequence $f(i)$. A staircase matrix is called a convex staircase Monge matrix (or concave staircase Monge matrix) if the Monge relation holds within the staircase. Note that if a matrix is concave staircase, then it is a concave Monge matrix, although the same statement is not true for a convex staircase matrix. Let $U = \{1, 2, .., n\}$. A function $F(i,j)$ on $U \times U$ is called *convex* (resp.*concave*) if the associated matrix is convex staircase Monge (resp. concave staircase Monge). The convex (resp. concave) dynamic programming problem is to compute a function $E(i)$ by using the inductive formula $E(i) = \min_{j<i}\{D(j) + F(i,j)\}$, where $D(j)$ can be computed in $O(1)$ time from $E(j)$, and $F(i,j)$ is convex (or concave).

It is known that such kind of dynamic programming problems can be solved efficiently by using the matrix searching technique [10, 8, 7]. In particular, if it takes $O(q)$ time to compute $F(i,j)$ for an arbitrary (i,j), it is known the concave problem can be solved in $O(nq)$ time [10], and convex problem can be solved in in $O(nq\alpha(n))$ time [11], where $\alpha(n)$ is the inverse Ackermann function.

3.2 Dynamic programming for concave problems

Let S be a set of m intervals whose endpoints are among $\{1, 2, .., n\}$. Each interval e has a nonnegative weight $w(e)$. We define $W(i,j)$, $\tilde{W}(i,j)$, $D(i)$, and $E(i)$ as in the introduction, where the function W is the weight function. The following lemma is a key observation, which is easy to verify:

Lemma 5. *If $w(e)$ are nonnegative, $\tilde{W}(i,j)$ is a concave function.*

We define the matrix A by $A(i,j) = D(i) + \tilde{W}(i,j)$. By definition, $E(j)$ is the minimum element in the j-th column. The following lemma is easy to show:

Lemma 6. *A is concave staircase Monge.*

If it takes $O(q)$ time to query the value of the weight function W for an arbitrary (i,j), the problem is solved in $O(nq)$ time [10]. Hence, if we use segment tree as the query structure, the overall time complexity becomes $O(m \log n)$, which matches that of Asano [1]; we improve it to $O(m + n \log\log n)$ by using the consecutive interval query data structure. We investigate the ordering of querying W in the algorithm of Klawe [10], and show that consecutive query structure is an efficient amortized query time structure.

The algorithm of Klawe [10] consists of a series of off-line matrix searching in rectangular submatrices; here, off-line means that $D(i)$ is known for each row index i of the matrix even before the algorithm begins. Each rectangular submatrix has contiguous column indices and row indices. The total sum of the column size and row size of the submatrices is $O(n)$. Therefore, we first consider the problem of searching all column minima of a (rectangular) Monge matrix M of size $n \times l$ where the value of $D(i)$ is known for the i-th row of M. We use the consecutive query structure, so that $M(i + i_0, j + j_0)$ can be computed in $O(i_0 + j_0)$ time from $M(i,j)$.

Lemma 7. *All column minima of M can be computed in $O((n+l)\log l)$ time.*

Proof. Let us denote the row index of the minimum of i-th column by $r(i)$. We first compute $r(l/2)$ (we can assume that l is even). This takes $O(n+l)$ time since we can find all entries $M(i, l/2)$ in $O(n+l)$ time by using the consecutive search. Because of the concave Monge property, the row indices of the column minima is a non-increasing sequence. Thus, $r(j) \geq r(l/2)$ (resp. $r(j) \leq r(l/2)$) if $j > l/2$ (resp. $j < l/2$). So, it suffices to compute in the shaded regions in Figure 3. Lemma 7 then easily follows from the corresponding recursion. □

We improve the time complexity of Lemma 7 by using the data structure for computing $W(i + i_0, j + j_0)$ from $W(i,j)$ in $O(\frac{i_0+j_0}{\log^2 n} + \log n)$ time. Let A be our $n \times n$ staircase Monge matrix, and let M be its submatrix corresponding to contiguous k rows and l columns of A.

Lemma 8. *All column minima of M can be computed in $O((k+l)\log\log n)$ time.*

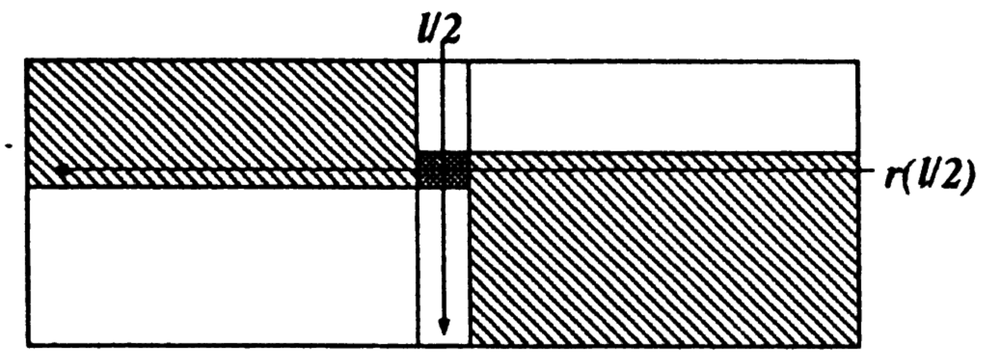

Fig. 3. Possible location of minima

Proof. If both l is smaller than $\log^3 n$, then the lemma follows from Lemma 7, since $\log l = O(\log\log n)$. Consequently, assume that l is larger than $\log^3 n$.

We consider the columns which has a column index of an integer multiple of $L = \lfloor \log^2 n \rfloor$. Assume that we have already computed the minimum of those columns. Given i, for every j such that $iL \geq j \geq (i+1)L$, $r(iL) \geq r(j) \geq r((i+1)L)$ because of Monge property. Therefore, it suffices to search in submatrices $M_1, .., M_s$, where $s = \lceil l/L \rceil$, such that M_i is a $(L-1) \times k_i$ matrix and $\sum_{i=1}^{s} k_i = k$. These sumbatrices can be computed in $O((k+l)\log\log n)$ time in total, since each submatrix has $\log^2 n$ contiguous columns. Thus, it suffices to compute $r(iL)$ for $i = 1, 2, .., s$. We first use the REDUCE subroutine of the SMAWK algorithm of [2]. Since REDUCE computes the entry in a consecutive manner, we can remove $k - \frac{l}{L}$ rows from the searched matrix in $O(k+l)$ time by using the consecutive query structure with $c = 0$. Thus we have a matrix M' which has $\frac{l}{L}$ rows and $\frac{l}{L}$ columns. However, neither the column indices nor the row indices are contiguous. We process this matrix by using the algorithm implicit in Lemma 7. By using the consecutive query structure, we can find all element on a column (and a row) of M' in $O(\frac{l}{\log^2 n} + \frac{l \log n}{L}) = O(\frac{l}{\log n})$ time. Thus, the algorithm of Lemma 7 computes all column minimum of M' in $O(l)$ time. □

Using the algorithm of Klawe [10] together with the matrix searching algorithm of Lemma 8, we obtain the following:

Theorem 9. *The concave dynamic programming problem on intervals can be solved in $O(m + n\log\log n)$ time.*

3.3 Dynamic programming for convex problems

This section uses the same notation as the concave case except that each interval e has a nonpositive weight.

Lemma 10. *If $w(e)$ are nonpositive for all e, $\tilde{W}(i,j)$ is a concave function.*

The algorithms of Klawe-Kleitman [11] and Aggarwal-Klawe [2] are based on matrix searching in rectangular submatrices; the difference from the convex case is that these submatrices do not have contiguous column indices. We first give a relatively naive $O(m + n\log n\log\log n)$ time algorithm, and then improve it to $O(m + n\log n)$ time.

Proposition 11. *The dynamic programming for convex weights can be solved in* $O(m + n \log n \log\log n)$ *time.*

Proof. The matrix $A(i, j)$ is a convex staircase matrix. As shown in [2], a staircase matrix can be decomposed into upper triangular submatrices and lower triangular submatrices. The total sum of the row size and column size of these submatrices is $O(n)$, and each submatrices has contiguous row indices and column indices. Thus, we can solve our problem as a series of matrix searching problems in triangular matrices. When we compute a lower triangular matrix, the value $D(i)$ for each row index of it has been already computed, thus we can process it by using Lemma 8. Hence, we can assume that $A(i, j)$ is an upper triangular matrix. Let $T(n)$ be the computing time. We apply a simple divide and conquer method. We divide the matrix $A(i, j)$ into four $n/2 \times n/2$ submatrices which are upper-left, upper-right, lower-left, and lower-right with respect to the $n/2$-th row and the $n/2$-th column. All entries in the lower-left submatrix are infinity, hence we need not compute it. We first compute, in $T(n/2)$ time, the column minimums of the upper-left submatrix, which is a triangular matrix of size $n/2 \times n/2$. We next compute the column minima of the right upper submatrix; this is a rectangular Monge matrix, and we know the value of $D(i)$ for each row index i of it. Thus, we can compute in $O(n \log\log n)$ time. Then, we search in the lower-right submatrix, which is upper triangular, in $T(n/2)$ time. Finally, we compare the column maxima of the upper-right submatrix and those of the lower-right submatrix in $O(n)$ time. Hence, $T(n) = 2T(n/2) + O(n \log\log n) = O(n \log n \log\log n)$. □

We can improve the time complexity of the above algorithm by a factor of $O(\log\log n)$; the new algorithm is based on the algorithm of Aggarwal-Klawe [2]. Because of space limitation, we omit it in this version.

Theorem 12. *The dynamic programming for a convex problem can be solved in* $O(m + n \log n)$ *time.*

4 Applications

4.1 Sequential partition of a graph

The optimal partition of a weighted graph is defined as follows: Given a graph $G = (V, E)$ with n nodes and m edges, an edge-weight function $w : E \to R^+$, a vertex-weight function $\phi : V \to R^+$, and a positive number K. A feasible partition is the partition of V into subsets V_1,..,V_s such that the sum of weights of the vertices in each cluster is no more than K. The optimal partition is the feasible partition that minimizes the total weight of the edges connecting different subsets.

The problem of computing an optimal partition of a graph occurs in several areas in computer science. Unfortunately, the problem of finding an optimal partition of a weighted graph is NP hard. Consequently, a popular heuristic for finding an approximation uses the sequential partition technique [9]. In this

technique, there is a linear ordering of the vertices, say, labeled from 1 to n. A sequential partition of G is a partition such that each cluster V_i consists of consecutive vertices with respect to this order. An optimal sequential partition is a sequential partition which minimizes the total cost of the edges connecting different clusters. We can consider a vertex of G as an integral point of the interval $I = [1, n]$, the edge between i and j a sub-interval of I, and a cluster of a sequential partition can also be regarded as an interval. Consequently, the problem becomes that of computing the functions $E(i)$ on U, such that $E(0) = 0$ and

$$E(i) = \max_{i-f(i) \le j \le i} \{E(j) + W(i, j)\} \qquad (2)$$

for a positive integer valued function f that is defined to be the smallest integer such that the summation of vertex weights of $\{v_{i-f(i)}, .., v_i\}$ does not exceed K.

Kernighan[9] first gave an algorithm for finding an optimal sequential partition. His algorithm is based on dynamic programming and runs in $O(n^2)$ time. Asano[1] improved the time complexity to $O(m \log n)$ time; we obtain the following:

Theorem 13. *The optimal sequential partition of G is obtained in $O(m+n \log n)$ time.*

Proof. It is easy to see that the function f can be computed in $O(m)$ time. The rest of the computation can be formulated as a convex dynamic programming on intervals by replacing each weight by its negative; this can be solved in $O(m + n \log n)$ time. □

4.2 Partial clique covering of interval graph

Given an interval graph G with m intervals on n terminals. Each interval has a positive weight. It is well-known that its maximal clique can be computed in $O(m)$ time, and the minimal clique covering can be computed in $O(m)$ time if the terminals are sorted. Consider the following two problems:

K-large clique problem: Given a number K, find K cliques $C_1, .., C_K$ of G such that the total weight in $\cup_{i=1}^{K} C_i$ is maximized.

Parametric partial clique covering problem: Given a nonpositive parameter t, find cliques $C_1, .., C_s$ such that the sum of ts and the total weight in $\cup_{i=1}^{s} C_i$ is maximized.

It can be easily seen that the partial parametric clique covering problem is a concave dynamic programming on intervals. Thus, we have the following:

Theorem 14. *The parametric partial clique covering problem can be solved in $O(m + n \log \log n)$ time.*

For K-large clique problem, we can easily give an $O(m+Kn \log \log n)$ time by applying matrix searching K times. Furthermore, by using a parametric searching algorithm, we can solve the problem by solving parametric partial clique covering problem on $\sqrt{K \log n}$ different parameters (for details, see [4]).

Theorem 15. *The K-large clique problem can be solved in $O((\min\{K, \sqrt{K \log n}\} n \log\log n) + n)$ time.*

5 Concluding remarks

In this paper, we only dealt with the convex case and the concave case of the dynamic programming problem of intervals although there are several problems which can be formulated as dynamic programming on intervals but with mixed weights (i.e., $w(e)$ may have arbitrary real value). It remains open whether the general case is solved in $o(m \log n)$ time.

References

1. T. Asano, "Dynamic Programming on Intervals," Proc. of ISA, LNCS 557, Springer Verlag (1991), 199-207.
2. A. Aggarwal and M. Klawe, "Applications of Generalized Matrix Searching to Geometric Algorithms," Discrete Appl. Math. 27(1990), 2-23
3. A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber, "Geometric Applications of a Matrix-Searching Algorithm," Algorithmica 2 (1987), 209-233.
4. A. Aggarwal, B. Schieber, and T. Tokuyama, "Finding a Minimum Weight K-Link Path in Graphs with Monge Property and Applications", Proc. 9th ACM Symp. on Computational Geometry (1993), 189-197.
5. B. Chazelle and L. Guibas, "Fractional Cascading: I. A Data Structure Technique," Algorithmica 1(1986), 133-162
6. R. Cole, "Searching and Storing Similar Lists," J. of Algorithms 7 (1986), 202-220.
7. D. Epstein, Z. Galil, R. Giancarlo, and G. Italiano, "Sparse Dynamic Programming," Proc. of the First ACM-SIAM Symp. on Discrete Algorithms, (1990), 513-522.
8. L. Larmore and B. Schieber, "On-line Dynamic Programming with Applications to the Prediction of RNA Secondary Structure," J. of Algorithms 12 (1991), 490-515.
9. B. Kernighan, "Optimal Sequential Partitions of Graphs," J. ACM 18(1971) 34-40.
10. M. Klawe, "A Simple Linear Time Algorithm for Concave One-Dimensional Dynamic Programming," Technical Report, University of British Columbia, Vancouver, 1989.
11. M. Klawe and D. Kleitman, "An Almost Linear Time Algorithm for Generalized Matrix Searching," Technical Report, IBM Almaden R.C., 1988.
12. F. Preparata and M. Shamos, Computational Geometry – An Introduction, 1988 (2nd edition), Springer-Verlag.