アルゴリズム 29－6
（1992. 8. 25）

準オンライン問題に対する新手法

戴陽* 今井浩** 岩野和生*** 加藤直樹*

* 神戸商科大学 管理科学科
** 東京大学 情報科学科
*** 日本IBM 東京基礎研究所

当論文は、ある問題に対して、部分的動的アルゴリズム (Partially dynamic algorithm) が与えられた時、その問題に対する準オンライン動的アルゴリズム (Fully dynamic semi-online algorithm) を求める手法を提案している。ここで、準オンラインの問題とは、オンライン問題の特殊な版であり、各更新リクエストσ時に、その後のk個の更新リクエストによって削除される要素を含む$O(k)$のサイズの集合SS_σと全部のリクエストの数lが与えられているものである。もし、$k=l$であれば、問題は通常のオフライン問題になり、$k=1$ならば、全リクエストサイズが与えられた時のオンライン問題となる。

最初に、$O(\sqrt{\#d+1}\cdot lK)$時間のサブセットサム問題の準オンラインアルゴリズムを与える。ここで、Kは目的の値であり、l個のリクエストのうち、$\#d$個の削除リクエストがあるとする。また、(a) 連結問題に対する$O(\sqrt{l(l+n)}\sqrt{\#d}\sqrt{\alpha(l,n)}+l\cdot\alpha(l,n))$時間のアルゴリズム（$n$は点の数である）(b) 整数ナップサック問題に対する$O(l\sqrt{\#d\cdot K}\sqrt{\alpha(lK^{1.5},K)}+l\cdot\alpha(lK^{1.5},K))$時間のアルゴリズム（$K$は目標の値である）(c) 0-1 ナップサック問題に対する$O(l^{9/4}c(\#d+1)^{1/4})$時間のアルゴリズム（cは最適値である）を与える。また、サブセットサム問題の準オンラインアルゴリズムの応用として、最大各差最小の等分割問題が$O(m+n^{2.5})$時間で解けることを示す。これらの時間限界は、著者らの知る限り、新しく自明ではない。

A New Approach to Semi-Online Problems

Yang Dai* Hiroshi Imai** Kazuo Iwano*** Naoki Katoh*

* Department of Management Science, Kobe University of Commerce, Gakuen-Nishimachi 8-2-1, Nishi-ku, Kobe 651-21, Japan.
** Department of Information Science, Faculty of Science, University of Tokyo, Tokyo 113, Japan.
*** Tokyo Research Laboratory, IBM Japan, 5-11 Sanbancho, Chiyoda-ku, Tokyo 102, Japan.

This paper proposes a new approach to obtain a *semi-online* dynamic algorithm, given a partially dynamic algorithm. A *semi-online* problem is a special case of online problems, in which for each update request σ we are given a superset SS_σ of size $O(k)$ which contains all items to be deleted in the succeeding k update requests as well as the total number of requests as well as the total number of requests. Thus, a semi-online problem has properties both of online and offline problems. Notice that an offline problem is also a semi-online problem. We first develop an $O(\sqrt{\#d+1}\cdot lK)$ time algorithm for the semi-online dynamic Subset Sum problem where K is a target value and a series of l requests, including $\#d$ deletions, is made to the initially empty set. We also devise the following semi-online dynamic algorithms: (a) an $O(\sqrt{l(l+n)}\sqrt{\#d}\sqrt{\alpha(l,n)}+l\cdot\alpha(l,n))$ time algorithm for the connectivity problem where l (*resp.* $\#d$) is the number of all (*resp.* delete) requests and n is the number of vertices, (b) an $O(l\sqrt{\#d\cdot K}\sqrt{\alpha(lK^{1.5},K)}+l\cdot\alpha(lK^{1.5},K))$ time algorithm for the Integer Knapsack problem where K is the target value, and (c) an $O(l^{9/4}c(\#d+1)^{1/4})$ time algorithm for the optimization 0-1 Knapsack problem where c is the optimal value. As an application of the semi-online dynamic Subset Sum problem, we devise an $O(m+n^{2.5})$ time algorithm for the minimum range balanced cut problem. To the authors' knowledge, these bounds are new and nontrivial.

1 Introduction

We propose a new approach to obtain an *semi-online* fully dynamic algorithm, given a partially dynamic algorithm by introducing a new way of handling delete requests. Here, a *dynamic* algorithm solves the problem for the current instance every time when an add/delete request is made to change the instance. When an dynamic algorithm allows only add requests, we call it *partially dynamic*, otherwise we call it *fully dynamic*. An *offline* algorithm gives a set of solutions of a dynamic algorithm when the entire request sequence is known beforehand. A *semi-online* problem is a special case of online problems, in which for each update request σ we are given a superset SS_σ of size $O(k)$ which contains all items to be deleted in the succeeding k update requests. Thus, a semi-online problem has properties both of online and offline problems. As typical semi-online problems, we have offline problems and minimum range problems [13].

Since these online/semi-online/offline dynamic algorithms have practical importance, we can find an extensive list of previous research activities. For example, Frederickson studied online updating of minimum spanning trees [7], Eppstein et al. considered the maintenance of minimum spanning forest in a dynamic planar graph [5], and Buchsbaum et al. studied the path finding problem when only arc insertions are allowed [2]. Eppstein also devised an offline algorithm for dynamic maintenance of the minimum spanning tree problem [4]. In the design of a fully dynamic algorithm, we often face the following difficulty: that is, since a delete request may drastically change the basic structure of a problem, we have to rebuild necessary data structures from scratch in such a case. This is a reason why the design of efficient fully dynamic algorithms is hard.

However, we can overcome the above difficulty for a certain class of problems including the Subset Sum problem, the connectivity problem, the Integer Knapsack problem, and the optimization 0-1 Knapsack problem. We introduce a new mechanism which regards each delete request as a set of add requests, and by allowing incremental maintenance of necessary data structures. As an instance, we develop an $O(\sqrt{\#d+1}\cdot lK)$ time algorithm for the semi-online dynamic Subset Sum problem with a target value K, when a series of l requests, including $\#d$ deletions, is made to the initially empty set. We also devise semi-online dynamic algorithms for the connectivity problem, Integer Knapsack problem, and optimization 0-1 Knapsack problem which runs in $O(\sqrt{l(l+n)}\sqrt{\#d}\sqrt{\alpha(l,n)}+l\cdot\alpha(l,n))$ time, $O(l\sqrt{\#d\cdot K}\sqrt{\alpha(lK^{1.5},K)}+l\cdot\alpha(lK^{1.5},K))$ time, and $O(l^{9/4}c(\#d+1)^{1/4})$ time, respectively, where l and $\#d$ are the numbers of requests and delete requests, n is the number of vertices in the connectivity problem, K is the target value in the Integer Knapsack problem, and c is the optimal value for the optimization 0-1 Knapsack problem. Notice that $\alpha(\cdot)$ is the functional inverse of the Ackermann function. To the authors' knowledge, these bounds are new and nontrivial.

Finally, we consider the minimum range balanced cut problem which can be viewed as an instance of the semi-online Subset Sum problem: Let $G=(V,E)$ be a connected undirected multigraph with n vertices and m edges. A *cut* C associated with a partition $(X, V-X)$ of the vertex set V with $X\neq\emptyset, V$ is defined as $C=\{(u,v)\in E \mid u\in X, v\in V-X\}$, and $|C|$ is called a *cut value* of C. Moreover, C is *balanced* when $|X|=|V-X|$, and C is *ϵ-balanced* when $(1-\epsilon)n/2\leq|X|\leq(1+\epsilon)n/2$. Given an edge weight function $w(\cdot)$, the *range* of a cut C is defined as the maximum difference of its edge weights: that is, $range(C)=\max_{e\in C}w(e)-\min_{e\in C}w(e)$. Then, the *minimum range balanced cut* problem is the problem of finding a cut with the minimum range among all balanced cuts. As discussed in [3], a minimum range balanced cut algorithm can be used for finding an approximate solution for the minimum balanced cut problem. Since the minimum balanced cut problem, a NP-complete problem [9], has an important application such as the circuit partitioning problem in the VLSI design and has been studied well [6, 11], an approximate solution by using an efficient minimum range balanced cut algorithm would be of broad interest. We then develop an $O(m+n^{2.5})$ time minimum range balanced cut algorithm, which improves an $O(m+n^3)$ time algorithm based on Martello et al.'s general approach to minimum range problems [13]. We also devise an $O(m+n^2/\sqrt{\epsilon})$ time algorithm for the minimum range ϵ-balanced cut problem.

This paper is organized as follows: In Section 2, we solve the semi-online dynamic Subset Sum problem in $O(\sqrt{\#d+1} \cdot lK)$ time. In Section 3, we generalize a technique developed in Section 2, and apply for developing semi-online dynamic algorithms for the connectivity problem, the Integer Knapsack problem, and the optimization 0-1 Knapsack problem. In Section 4, we introduce an $O(m + n^{2.5})$ time minimum range balanced cut algorithm.

2 The semi-online dynamic Subset Sum problem

In this section, we consider the semi-online dynamic Subset Sum problem and introduce a new approach for handling delete requests. Our algorithm takes $O(\sqrt{\#d+1} \cdot lK)$ time with a target value K, when a series of l requests, including $\#d$ deletions, is made to the initially empty set.

For simplicity, we first consider the offline dynamic Subset Sum problem. In the next section, we will see that this offline algorithm can be easily applied to the semi-online case with an appropriate modification. We assume that at each update request, a query for the feasibility for the Subset Sum problem is issued. Then, the *offline dynamic Subset Sum problem* can be formulated as follows:

Input: A positive integer K and a sequence, $Request = (r_1, r_2, \ldots, r_l)$, of l requests. Each request is either an addition request (add, x) or a deletion request $(delete, x)$ where x indicates an item to be added or deleted.
Output: A sequence $(bit_1, bit_2, \ldots, bit_l)$ where $bit_i = 1$ (*resp.* 0) indicates the feasibility (*resp.* infeasibility) of the Subset Sum problem for S_i.

For the Subset Sum problem, there is an $O(sK)$ time algorithm based on dynamic programming ([14]) for an s-item set, as shown in Figure 1.

Input: A positive integer K and the initial set $S = \{c_1, c_2, \ldots, c_s\}$.
Output: A bit indicating the feasibility.
Procedure *Subset Sum*
(1) Mark the node 0. $M := \{0\}$;
(2) **for** $j = 1, 2, \ldots, s$ **do**
For each marked node v, mark the node u and add it to M such that $u = v + c_j$;
end
(3) **if** $K \in M$, **then** return (1); **else** return (0);

Figure 1. Algorithm *Subset Sum*

In the algorithm *Subset Sum*, the set M, which we call the *marking set*, consists of all values which can be expressed as $\sum_{k=1}^{s} x_k \cdot c_k$ where $x_k \in \{0, 1\}$. Notice that we can regard the above algorithm *Subset Sum* as an $O(sK)$ time online algorithm which allows only add requests. However, if there exists a delete request, we have to reconstruct the marking set M, which takes $O(sK)$ time. Therefore, an offline dynamic Subset Sum algorithm using the above algorithm takes $O((\#d+1)lK)$ time for l requests including $\#d$ delete requests. We, thus, develop a new technique which enables us to avoid costly reconstruction of the marking set at each delete request.

We first divide l requests into disjoint $\lceil l/k \rceil$ stages of which each possibly except the last consists of consecutive k requests. We now consider the i-th stage. Suppose that we have a set of items $S(i)$ at the beginning of this stage. Let $S(i) = Remain(i) \cup Delete(i)$ where $Remain(i)$ (*resp.* $Delete(i)$) consists of items in $S(i)$ which will not (*resp.* will) be deleted in the stage. Note that $|Delete(i)| = O(k)$. We first create a marking set R (*resp.* D) with respect to $Remain(i)$ (*resp.* $Delete(i)$) in $O(lK)$ time. If $K \in R$, this implies that for all requests in this stage we

have already had a feasible solution. If $K \notin R$, we do the following. For each item addition/deletion request, we maintain $Delete(i)$ by adding/deleting a requested item, and update a marking set D with respect to this request by using the above naive method. That is, for each add request we incrementally update D, which takes $O(K)$ time, and for each delete request we reconstruct D from scratch with respect to the current $Delete(i)$, which takes $O(kK)$ time. Whenever we obtain a new marked node x in D, we check whether $K - x \in R$ or not. If so, we have a feasible solution with the current request. Thus, each stage takes $O(lK + \#a(i) \cdot K + \#d(i) \cdot kK)$ time, where $\#a(i)$ (*resp.* $\#d(i)$) is the number of add (*resp.* delete) requests in stage i. Therefore, our algorithm in total takes $O(\lceil l/k \rceil lK + \#a \cdot K + \#d \cdot kK) = O(\lceil l/k \rceil lK + \#d \cdot kK)$, where $\#a$ (*resp.* $\#d$) is the number of add (*resp.* delete) requests among l requests. Hence we have the following theorem:

Theorem 1 *The offline dynamic Subset Sum problem with l requests, including $\#d$ delete requests, and a target value K can be correctly computed in $O(\sqrt{\#d+1} \cdot lK)$ time.* □

Since the naive algorithm takes $O((\#d+1)lK)$ time, our algorithm speeds up the running time by a factor of $\sqrt{\#d+1}$. Figure 2 shows one stage of our algorithm *Offline Dynamic Subset Sum* described above.

Procedure *Stage*(i)

(1) • Let $Remain(i)$ and $Delete(i)$ be defined as in the text. Let $Request(i)$ be a sequence of all requests in stage i.
• Create two marking sets R and D with respect to $Remain(i)$ and $Delete(i)$, respectively.

(2) **if** $K \in R$ **then** {For each request $r_j \in Request(i)$, set $bit_j := 1$;}

(3) **else for** each request $r_j \in Request(i)$ **do**

(3.1) • **if** r_j is an add request (add, α) **then** {Update D by α; Add α to $Delete(i)$;}

(3.2) • **if** r_j is a delete request $(delete, \alpha)$ **then**
{Delete α from $Delete(i)$; Reconstruct D with respect to $Delete(i)$;}

(3.3) • Check whether there exists $x \in D$ such that $K - x \in R$. If so, set $bit_j := 1$. Otherwise, set $bit_j := 0$.

end

Figure 2. *Stage*(i) of Algorithm *Offline Dynamic Subset Sum*

3 Further Applications of the Technique

We first assume that a query for the feasibility check is issued after each update request. Let $T_q(l)$ be the running time of each query where l is the total number of requests. We denote a query made right after all requests in S by *the query for* S. Suppose we have a problem which has the following properties:

(1) There exists an offline partially dynamic algorithm which runs in $T_a(\cdot)$ time for each add request.

(2) For a set S of add requests, the result of the query for S does not depend on the order of requests in S, but only depends on S itself. Let $T_f(|S|)$ denote the running time to answer the query for S by an algorithm A_f.

(3) For a set S of add requests, let it be separated into two disjoint subsets S_1 and S_2. Then we assume that the result of the query for S can be obtained in $T_c(|S_1|, |S_2|)$ time by *combining* information obtained when we run the algorithm A_f for getting results of two queries for S_1 and S_2.

For a problem with the above properties, we can construct a fully dynamic semi-online algorithm in the same way as in the previous section. Remark that in a semi-online problem for each update request σ we are given a superset SS_σ of size $O(k)$ which contains a set DS_σ of all items to be deleted in the succeeding k update requests.

That is, $DS_\sigma \subset SS_\sigma$ and $|SS_\sigma| = O(k)$. Notice that the technique in the previous section works correctly by handling SS_σ instead of handling DS_σ.

Theorem 2 *For a problem with the above properties, there is an*

$$\lceil l/k \rceil \cdot T_f(l) + \#d \cdot T_f(k) + \#a \cdot T_a(\cdot) + l \cdot T_c(k,l) + l \cdot T_q(l)$$

time semi-online dynamic algorithm where l, $\#d$, and $\#a$ are the total number of requests, the number of delete requests, and the number of add requests, and k is the number of requests in each stage. □

3.1 The Connectivity Problem

The semi-online dynamic version of the connectivity problem is as follows: Given an initial undirected graph $G_0 = (V, \emptyset)$ with n vertices and no edges, we have a sequence of the following requests: *insert-edge*(v,w) requests for an insertion of an edge (v,w), *delete-edge*(v,w) requests for an deletion of an edge (v,w), and *check*(v,w) requests for checking whether v and w are in the same component or not.

Notice that we have an partially online algorithm which takes $\alpha(l,n)$ amortized time for each insertion where l is the total number of add requests by using UNION-FIND data structures [18]. Note that $\alpha(\cdot)$ is the functional inverse of the Ackermann function.

Notice that the second term of the time complexity appeared in Theorem 2 (that is, $\#d \cdot T_f(k)$) can be replaced by $\#d \cdot k \cdot T_a(\cdot)$, since instead of using algorithm A_f we may use k incremental updates for each delete requests. Therefore, we have the following theorem:

Theorem 3 *We have an $O(\sqrt{l(l+n)}\sqrt{\#d}\sqrt{\alpha(l,n)} + l \cdot \alpha(l,n))$ time semi-online dynamic algorithm for the connectivity problem.* □

Notice that Frederickson's fully dynamic online algorithm [7] takes $O(\sqrt{m_i})$ amortized time for each add/delete request where m_i is the current number of edges. For the offline dynamic version of the connectivity problem, we can implement each update request in $O(\log n)$ amortized time by using Sleator and Tarjan's dynamic trees [16] as follows: We first define an edge weight as ∞ for an edge which will not be deleted and i for an edge which will be deleted by the i-th delete request. Then, the problem becomes the maintenance of maximum spanning tree, which takes $O(\log n)$ amortized time for each request. Since our algorithm above takes $O(\sqrt{(l+n)/l}\sqrt{\#d}\sqrt{\alpha(l,n)} + \alpha(l,n))$ amortized time for each request, our algorithm runs faster than Frederickson's algorithm and algorithm with dynamic trees when $\#d$ is small.

3.2 The Integer Knapsack Problem

As an application of the semi-online dynamic connectivity problem, we devise an semi-online dynamic Integer Knapsack algorithm in this subsection. Here, the Integer Knapsack problem is defined as follows:

The Integer Knapsack problem [14]: Given integers c_j, $j = 1, \ldots, n$ and K, are there integers $x_j \geq 0$, $j = 1, \ldots, n$ such that $\sum_{j=1}^{n} c_j x_j = K$?

It is well known that the Integer Knapsack problem can be solved by checking whether 0 and K are in the same component or not in the graph $G = (V, E)$ which is defined as follows: $V = \{0, 1, \ldots, K\}$, $E = \{(i,j) \mid i, j \in V, j - i = c_j \text{ for some } j \in \{1, \ldots, n\}\}$. Therefore, we can implement an semi-online dynamic algorithm for the Integer Knapsack problem by making use of the above defined semi-online dynamic algorithm for the connectivity problem. Notice that at each add (*resp.* delete) request (add, c_j) (*resp.* $(delete, c_j)$), we have to add (*resp.* delete) at most K edges to (*resp.* from) G. Therefore, we have the following theorem:

Theorem 4 *We have an* $O(l\sqrt{\#d \cdot K \cdot \alpha(lK^{1.5}, K)} + l \cdot \alpha(lK^{1.5}, K))$ *time semi-online dynamic algorithm for the Integer Knapsack problem.* □

Notice that the above time complexity is faster than the naive bound of $O(l^2K)$ which creates an associated graph and runs a linear time connectivity algorithm at each request. The above time complexity is also faster than $O((lK)^{1.5})$ time obtained by using Frederickson's fully dynamic online algorithm [7] for the connectivity problem.

3.3 The Optimization 0-1 Knapsack Problem

The optimization 0-1 Knapsack problem is defined as follows:

The optimization 0-1 Knapsack problem [14]: Given the integers $(w_1, \ldots, w_n; c_1, \ldots, c_n; K)$, maximize $\sum_{j=1}^{n} c_j x_j$ subject to $\sum_{j=1}^{n} w_j x_j \leq K$ and $x_j = 0, 1$.

We now have the following theorem:

Theorem 5 *We have an* $O(l^{9/4}c(\#d+1)^{1/4})$ *time semi-online dynamic algorithm for the optimization* 0-1 *Knapsack problem where* c *is the optimal value.* □

Notice that the above time complexity is faster than the naive $O(l^3c)$ bound using $O(l^2c)$ time dynamic programming [14] at each request.

4 The minimum range balanced cut problem

From now on, we assume for simplicity that all edge weights are distinct. We will discuss the case in which some edge weights are not distinct in a full version of this paper. Let $E[\alpha, \beta] = \{e \in E \mid \alpha \leq w(e) \leq \beta\}$. An interval $[\alpha, \beta]$ is said to be *feasible* if $E[\alpha, \beta]$ contains a balanced cut. Otherwise, we say that the interval $[\alpha, \beta]$ is *infeasible*. An interval $[\alpha, \beta]$ is said to be *critical* when it is feasible and any proper sub-interval is infeasible. From now on, let T_{min} (*resp.* T_{max}) be a minimum (*resp.* maximum) spanning tree of G.

The feasibility of an interval $[\alpha, \beta]$ can be tested in $O(m + n^2)$ time as follows. First, we contract all edges in $E - E[\alpha, \beta]$ because any of these edges cannot be a member of any cut in $E[\alpha, \beta]$. When an edge (u, v) is contracted, u and v are merged into one to form a supernode. Let $G' = (V', E[\alpha, \beta])$ be the resulting graph, where V' is the set of supernodes. G' can be constructed in $O(m)$ time. Let $f(v)$ for $v \in V'$ denote the number of vertices of V which are contracted into a single supernode v. The feasibility of $[\alpha, \beta]$ is then reduced to the problem of whether there exists a subset $V'' \subset V'$ such that $\sum_{v \in V''} f(v) = n/2$. This is exactly equivalent to the Subset Sum problem, and can be solved in $O(n^2)$ time [14]. Since for any cut C, an edge e with the maximum (*resp.* minimum) weight among the edges in C belongs to T_{max} (*resp.* T_{min}) on the assumption that the edge weights are distinct [10], we shall assume in this paper that E has been already reduced to $T_{min} \cup T_{max}$. Furthermore, applying the general approach proposed by Martello et al. [13], we can show that $O(n)$ feasibility tests are sufficient.

We now briefly explain our algorithm called *MRBC* based on [13]. Let $w_1, w_2, \cdots, w_p$ be edge weights sorted in ascending order, and let e_i be an edge such that $w(e_i) = w_i$. Letting $l = 1$ and $u = 1$, we start with the feasibility test of $[w_l, w_u]$ and $G' = (\{v_0\}, \emptyset)$ consisting of a single supernode v_0 obtained by contracting all the edges. In general, the algorithm proceeds by alternately executing unfolding and contraction phases. As long as $[w_l, w_u]$ is infeasible, the unfolding phase repeats the following steps: (1) we unfold e_{u+1} and update u by $u + 1$, and (2) test the feasibility of $[w_l, w_u]$. Eventually, the unfolding phase finds the lowest $\tilde{u}$ such that $[w_l, w_{\tilde{u}}]$ is feasible. Once $[w_l, w_{\tilde{u}}]$ becomes feasible, the algorithm enters into the contraction phase. The contraction phase

repeats the following steps: (1) we contract e_l and update l by $l+1$, and (2) test the feasibility of $[w_l, w_{\bar{u}}]$. When $[w_l, w_{\bar{u}}]$ becomes infeasible, the algorithm concludes that an interval $[w_{l-1}, w_{\bar{u}}]$ is critical, and then it enters into the unfolding phase again with $[w_l, w_{\bar{u}}]$. It is known [13] that the interval $[w_{l^*}, w_{u^*}]$ with the minimum range among all feasible intervals generated in the above process is the desired minimum range, and that the minimum range cut is easily constructed from $E[w_{l^*}, w_{u^*}]$. Notice that since $G' = (V, E[w_l, w_u])$ is maintained at each step and from Lemma 1, each feasibility test can be done in $O(n^2)$ time including the construction of G'. Thus, since this algorithm executes the feasibility tests $O(n)$ times [13], we need a total of $O(m+n^3)$ time for the feasibility testing. Since T_{min} and T_{max} can be computed in $O(m+n\log n)$ time [8] and sorting the edges of $T_{min} \cup T_{max}$ requires $O(n\log n)$ time, we can establish the following theorem.

Theorem 6 *The minimum range balanced cut problem can be solved in $O(m+n^3)$ time.* □

Notice that, in the above Algorithm *MRBC*, the feasibility test of an interval $[w_i, w_j]$ corresponds to the feasibility test of the Subset Sum problem of a set $\{f(v) \mid v \in V'\}$ such that V' is the vertex set obtained by contracting all edges in $E - E[w_i, w_j]$. Let $G_{i,j}$ be the graph obtained by the above contractions, and let $f(v)$ be the size of a supernode v in $G_{i,j}$. When $[w_i, w_j]$ is infeasible, Algorithm *MRBC* unfolds the edge e_{j+1} and creates $G_{i,j+1}$ from $G_{i,j}$. This unfolding corresponds to the following three requests to the current Subset Sum instance: $(delete, f(p)+f(q))$, $(add, f(p))$, and $(add, f(q))$, where $e_{j+1} = (p, q)$ in $G_{i,j+1}$. On the other hand, when $[w_i, w_j]$ is feasible, Algorithm *MRBC* contracts the edge e_i and creates $G_{i+1,j}$ from $G_{i,j}$. This contraction corresponds to the following three requests to the current Subset Sum instance: $(delete, f(q))$, $(delete, f(q))$, and $(add, f(p)+f(q))$, where $e_i = (p, q)$ in $G_{i,j}$.

When we apply our offline dynamic Subset Sum algorithm to the minimum range balanced cut problem, the major difficulty lies in that, when Algorithm *MRBC* executes the contraction (*resp.* unfolding) phase, we cannot predict beforehand when it will switch to the unfolding (*resp.* contraction) phase. This implies that it is impossible to generate the entire sequence of requests to be made to the initial Subset Sum instance at the beginning. However, we can overcome this difficulty by using the following fact: *Stage*(i) in Figure 2 can run correctly as long as it uses a subset $A(i) \subseteq$ *Remain*(i) (*resp.* $B(i) = S(i) - A(i)$) instead of *Remain*(i) (*resp.* *Delete*(i)), even if it does not know either *Delete*(i) or *Remain*(i) in advance. This is because, since $A(i)$ is a subset of *Remain*(i), $A(i)$ will not be destroyed in this stage. Moreover, the time complexity of the algorithm remains the same if $|B(i)| = O(k)$.

Suppose that we are given the current interval $[w_i, w_j]$ and the corresponding graph $G_{i,j}$. In this case we have a full knowledge of the k edges to be contracted (*resp.* unfolded) if k consecutive contractions (*resp.* unfoldings) occur in the next k feasibility tests. This is because contractions and unfoldings of edges occur in the ascending order of edge weights. Therefore, we can obtain a superset of $2k$ edges that are contracted or unfolded in the next k feasibility tests. In terms of the Subset Sum problem, in the succeeding k feasibility tests, the items corresponding to both endpoints of these edges may be deleted, while others may not be deleted. This implies that we can determine $O(k)$ items that may be deleted, but that the other items are not deleted during solving the succeeding k Subset Sum instances. These $O(k)$ items are called *dangerous* and the others *safe*. We are sure that any safe item will not be deleted by any means in the succeeding k Subset Sum instances.

In order to translate our problem into the semi-online dynamic Subset Sum problem, we divide a sequence of $O(n)$ feasibility tests that will occur in Algorithm *MRBC* into $O(\sqrt{n})$ disjoint stages of which each, possibly except the last, consists of $\sqrt{n}$ consecutive feasibility tests. At the beginning of each stage i, we construct the current interval $[w_h, w_j]$ and the corresponding graph $G_{h,j}$, and compute the two sets of dangerous and safe items. We then treat dangerous items as *Delete*(i) and safe items as *Remain*(i) and construct a corresponding marking set R. With this preprocessing, we execute Algorithm *Stage*(i) in an online manner. Since we do not know the sequence of $O(n)$ feasibility tests, we shall modify the algorithm *Stage*(i) as follows: That is, every time we execute a contraction or unfolding of edge e, we generate the corresponding three requests, as described above, and perform

Step (3) for each generated request. Since the marking set R is not destroyed by the succeeding k contractions or unfoldings, $Stage(i)$ runs correctly after the above modifications.

Therefore, the minimum range balanced cut problem can be solved as an instance of the semi-online dynamic problem.

Theorem 7 *The minimum range balanced cut problem can be solved in* $O(m + n^{2.5})$ *time.* □

Furthermore, we can extend the above result to the ϵ-balanced problem which we give only the results in this abstract.

Theorem 8 *The minimum range* ϵ*-balanced cut problem can be solved in* $O(m + n^2/\sqrt{\epsilon})$ *time.* □

参考文献

[1] Ahuja, R.K., T.L. Magnanti, and J.B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, Englewood Cliffs, N.J., 1992.

[2] Buchsbaum, A.L., P.C. Kanellakis, and J.S. Vitter, A data structure for arc insertion and regular path finding, *Proc. 1st ACM/SIAM Symp. Discrete Algorithms*, (1990), pp. 22-31.

[3] Dai, Y., H. Imai, K. Iwano, N. Katoh, K. Ohtsuka, and N. Yoshimura, A new unified approximate approach to the minimum cut problem and its variants using minimum range cut algorithms, manuscript, 1992.

[4] Eppstein, D., Offline algorithms for dynamic minimum spanning tree problems, *Proceedings of 2nd Workshop, WADS '91*, Lecture Notes in Computer Science 519 Springer-Verlag, (1991), pp. 392-399.

[5] Eppstein, D. G.F. Italiano, R. Tamassia, R.E. Tarjan, J. Westbrook, and M. Yung, Maintenance of a minimum spanning forest in a dynamic planar graph. *Proc. 1st ACM/SIAM Symp. Discrete Algorithms*, (1990), pp. 1-11.

[6] Fiduccia, C.M. and R.M. Mattheyses, A linear time heuristic for improving network partitions, In *Proceedings of the 19th Design Automation Conference*, ACM/IEEE, (1982), pp. 175-181.

[7] Frederickson, G.N., Data structures for on-line updating of minimum spanning trees, *SIAM J. Computing*, 14 (1985), pp. 781-798.

[8] Fredman, M.L. and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *Journal of the ACM*, Vol. 34, No. 3, (1987) pp. 596-615.

[9] Garey, M.R. and D.S. Johnson, *Computers and Intractability - A Guide to the Theory of NP-completeness*, W. H. Freeman and Company, New York, NY, 1979.

[10] Katoh, N. and K. Iwano, Efficient algorithms for minimum range cut problems, IBM Research Report, RT0057 (1991) (also appeared in *Proceedings of 2nd Workshop, WADS '91*, Lecture Notes in Computer Science 519 Springer-Verlag, (1991), pp. 80-91).

[11] Kernighan, B.W. and S. Lin, An effective heuristic procedure for partitioning graphs, *BSTJ*, Vol.49, No.2, (1970), pp. 291- 307.

[12] Lengauer, T., *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley & Sons, West Sussex, England, 1990.

[13] Martello, S., W.R. Pulleyblank, P. Toth, and D. de Werra, Balanced optimization problems. *Operations Research Letters*, Vol. 3, No. 5, 275-278. 1984.

[14] Papadimitriou, C.H. and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[15] Saran, H. and V.V. Vazirani, Finding k-cuts within twice the optimal. *Proc. 32nd IEEE Symp. Found. Compt. Sci.*, (1991), pp. 743-751.

[16] Sleator, D.D. and R.E. Tarjan, A data structure for dynamic trees, *Journal of Computer and System Sciences*, 26, pp. 362-391, (1983).

[17] Stone, H.S., Multiprocessor scheduling with the aid of network flow algorithms, *IEEE Trans. Software Engineering*, SE-3 (1977), pp. 85-93.

[18] Tarjan, R.E., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.