

## 可変優先キューとその応用

A Variable Priority Queue and its Applications

鈴木 均      西関隆夫      斎藤伸自  
Hitoshi Suzuki      Takao Nishizeki      Nobuji Saito

東北大学工学部通信工学科

Department of Electrical Communications, Faculty of Engineering, Tohoku University

**Abstract.** This paper proposes a new data structure called a "variable priority queue". The queue supports, in addition to the ordinary queue operations, an operation MIN to find an item of minimum key and operations to change keys of items. Any sequence of these  $m$  operations can be processed in  $O(m)$  time. The variable priority queue is useful in designing efficient algorithms for various network problems such as the multicommodity flow and edge-disjoint path problems on planar graphs.

### 1. INTRODUCTION

This paper proposes a new data structure called a "variable priority queue", which is a generalization of a queue. The variable priority queue supports, in addition to the ordinary queue operations, an operation MIN to find an item of minimum key together with two operations DECREASE and UPDATE to change keys of items in the queue. The variable priority queue is suited for efficient algorithms for network problems. These include the multicommodity flow and edge-disjoint path problems on planar graphs. Using the variable priority queue, we present a linear algorithm for finding multicommodity flows in a cycle graph.

### 2. VARIABLE PRIORITY QUEUE

A variable priority queue  $Q$  is a sequence of items ordered from left to right, and each item  $q$  in  $Q$  is associated with a real number  $\text{key}(q)$  called a key.

The following seven operations are possible on the queue:

- 1) MAKEQUEUE(Q) : make an empty queue Q;
- 2) INJECT(Q,q,key(q)): insert a new item q with key(q) into Q as the rightmost item;
- 3) POP(Q) : delete the leftmost item in Q;
- 4) DECREASE(Q,q,D) : given an item q in Q together with a nonnegative number D, decrease by D all the keys of item q and those on q's right;
- 5) UPDATE(Q,D) : add some real number D to all the keys of items in Q; and
- 6) MIN(Q) : return the minimum key of items in Q.

The following operation is permitted only if all the keys of items in Q are nonnegative:

- 7) DECREASE\*(Q,q,D) : given an item q in Q and a nonnegative number D, return  $D' = \min\{D, \min\{\text{key}(q') \mid q' \text{ is } q \text{ or on } q\text{'s right}\}\}$ , and execute DECREASE(Q,q,D').

The variable priority queue is a generalization of an ordinary queue but not one of a priority queue, because in the variable priority queue an item can be inserted only to the tail and deleted only from the head.

Clearly the variable priority queue is realized by a balanced tree such as a 2-3 tree [AHU,GMG]. However, in such a direct implementation of the queue Q, the execution of each operation spends  $O(\log n)$  time and a sequence of m operations above consumes  $O(m \log n)$  time if Q has n items. In the next section, we present a sophisticated implementation of the queue using a disjoint set union algorithm [GT], in which any sequence of m queue operations above can be executed in  $O(m)$  time.

A disjoint set union algorithm solves the problem of maintaining a collection of disjoint sets under the following operations [GT,Tar]:

- MAKESET(q) : Create a new singleton set  $S = \{q\}$ , and return the name S.
- UNITE(S,S') : Create a new set that is the union of two disjoint sets S and S'. The old sets S and S' are discarded, and the new set is named S.
- FIND(q) : Return the name of the set containing element q.

Consider a sequence of  $m$  operations consisting of the three operations above. Let  $n$  be the number of elements in sets, that is, let  $n$  be the number of MAKESET operations in the sequence. Then the sequence can be executed in  $O(ma(m,n))$  time, where  $a$  is a functional inverse of Ackerman's function [Tar].

Gabow and Tarjan gave a linear algorithm for a special case of set union in which the structure of the unions, as defined by a "union tree"  $T$ , is known in advance. That is, the elements in sets correspond to the vertices of the tree  $T$ , and the elements in each set must induce a subtree of  $T$  throughout the execution of the algorithm. Then any sequence of  $m$  set union operations is executed in  $O(m)$  time with  $O(n)$  preprocessing time [GT].

### 3. IMPLEMENTATION of QUEUE Q

In this section we show how to realize a variable priority queue  $Q$ . The key ideas are two-fold: first  $Q$  is partitioned into a collection of subsequences suited for supporting the queue operations; and then the collection of disjoint sets corresponding to the subsequences are maintained by a disjoint set union algorithm. Let  $Q$  be partitioned into a collection of subsequences  $S_1, S_2, \dots, S_k$  for some  $k$ , and let  $q_{S_i}$  be the rightmost item in  $S_i$  for each  $i$ ,  $1 \leq i \leq k$ . Then the following conditions (1), (2) and (3) must be satisfied. (See Fig. 1.)

- (1)  $S_i$ ,  $1 \leq i \leq k$ , is a consecutive nonempty subsequence of  $Q$ ;
- (2)  $\text{key}(q_{S_1}) \leq \text{key}(q)$  for every  $q \in S_i$ ; and
- (3)  $\text{key}(q_{S_{i-1}}) < \text{key}(q_{S_i})$  for every  $i$ ,  $2 \leq i \leq k$ .

Clearly  $q_{S_1}$  is the rightmost item having the minimum key among all the items in  $Q$ , and  $q_{S_i}$ ,  $2 \leq i \leq k$ , is the rightmost item having the minimum key among those on  $q_{S_{i-1}}$ 's right. We use four pointers pred, succ, right and left. Each set  $S_i$  is accessed from  $S_{i+1}$  by pointer pred( $S_{i+1}$ ). The header of the list

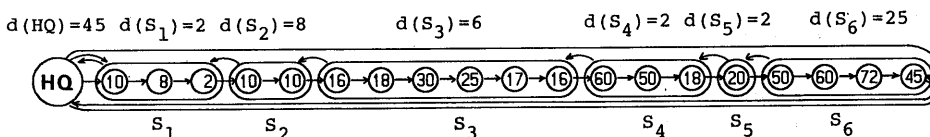


Fig. 1 Variable priority queue  $Q$ .

representing queue  $Q$  is denoted by  $HQ$ , and  $S_1$  and  $S_k$  are accessed from  $HQ$  by  $\text{succ}(HQ)$  and  $\text{pred}(HQ)$ , respectively. That is,

```

pred(Si)=Si-1 for each i, 2<i<k;
pred(S1)=HQ;
pred(HQ)=Sk; and
succ(HQ)=S1.

```

For each  $q \in Q$ , the element next to  $q$ 's right is accessed by  $\text{right}(q)$ . If  $q$  is the rightmost element in  $Q$ , then  $\text{right}(q)=HQ$  and  $\text{left}(HQ)=q$ . The leftmost element in  $Q$  is accessed by  $\text{right}(HQ)$ . Instead of maintaining all the keys, we maintain real numbers  $d(S_i)$  and  $d(HQ)$ . Number  $d(S_i)$  is associated with  $S_i$ ,  $1 \leq i \leq k$ , and  $d(HQ)$  with  $HQ$ , and are defined as follows:

```

d(S1)=key(qS1) (=min{key(q) | q ∈ Q});
d(Si)=key(qSi)-key(qSi-1) for each i, 2<i<k; and
d(HQ)=key(qSk).

```

Then clearly the condition (3) is equivalent with

(3)'  $d(S_i) > 0$  for every  $i$ ,  $2 \leq i \leq k$ .

Operation  $\text{MIN}(Q)$  is simply performed by returning  $d(\text{succ}(HQ))$ . Furthermore operation  $\text{UPDATE}(Q, D)$  is performed simply by executing two substitutions:  $d(HQ) := d(HQ) + D$  and  $d(\text{succ}(HQ)) := d(\text{succ}(HQ)) + D$ .

In order to show how to implement other operations, we first present a procedure  $\text{REFORM}$ . Let  $Q$  be an incompletely structured queue, which is partitioned into  $S_1, S_2, \dots, S_k$  so that conditions (1), (2) and (3)' are satisfied except that there exists only one set  $S_i$  having possibly nonpositive  $d(S_i)$ . Then the following procedure  $\text{REFORM}(Q, S_i)$  using operation  $\text{UNITE}$  of the disjoint set union algorithm makes  $Q$  to satisfy all the conditions (1), (2) and (3)'.

```

procedure REFORM(Q,S);
  begin
    while pred(S) ≠ HQ and d(S) ≤ 0 do
      begin
        S' := pred(S);
        d(S) := d(S') + d(S);
        pred(S) := pred(S');
        UNITE(S,S')
      end
    end;

```

Using procedure REFORM together with set union operations MAKESET, UNITE and FIND, we can implement the remaining queue operations MAKEQUEUE, INJECT, POP, DECREASE and DECREASE\* as follows.

```

procedure MAKEQUEUE(Q);
  begin
    succ(HQ)=pred(HQ)=left(HQ)=right(HQ):=HQ;
    d(HQ):=-∞
  end;

procedure INJECT(Q,q,key);
  begin
    {see Fig. 2(a)}
    S:=MAKESET(q);
    S':=pred(HQ);
    q':=left(HQ);
    pred(S):=S'; pred(HQ):=S;
    right(q):=HQ; right(q'):=q; left(HQ):=q;
    if q'=HQ then d(S):=key
      else d(S):=key-d(HQ);
    d(HQ):=key;
    REFORM(Q,S)
  end;

```

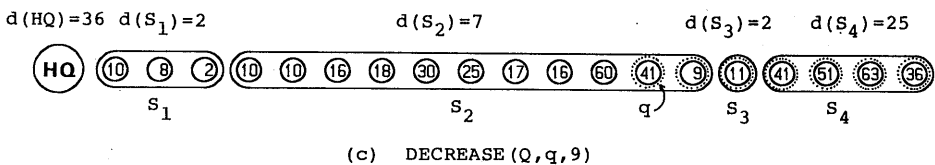
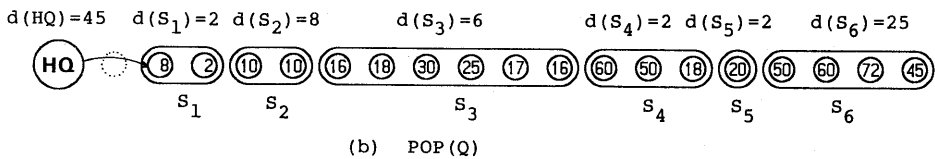
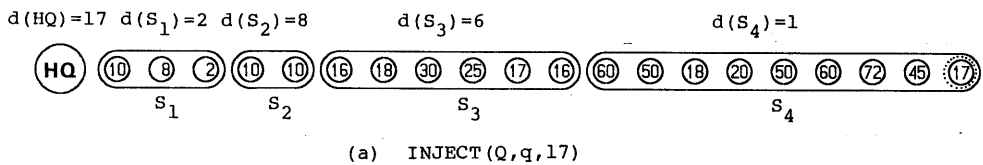


Fig. 2 Queues after INJECT, POP and DECREASE are applied to Q in Fig. 1.

```

procedure POP(Q);
  begin
    {see Fig. 2(b)}
    if right(HQ)≠HQ then {Q has at least one item}
      if right(right(HQ))=HQ then {Q has exactly one item}
        MAKEQUEUE(Q);
      else {Q has at least two items}
        begin
          q':=right(right(HQ));
          S:=succ(HQ);
          S':=FIND(q');
          right(HQ):=q';
          if S≠S' then
            begin
              pred(S'):=HQ;
              succ(HQ):=S';
              d(S'):=d(S)+d(S')
            end
          end
        end;

```

```

procedure DECREASE(Q,q,D);
  begin
    {see Fig. 2(c)}
    d(HQ):=d(HQ)-D;
    S:=FIND(q);
    d(S):=d(S)-D;
    REFORM(Q,S)
  end;

```

```

procedure DECREASE*(Q,q,D);
  begin
    S:=FIND(q);
    { key(qS)=min{key(q')|q' is q or on q's right},
      key(qS)=Σ{d(S')|S' is S or precedes S in Q} }
    KQS:=d(S);
    while KQS<D and pred(S)≠HQ do
      begin
        S:=pred(S);
        KQS:=KQS+d(S)
      end;
    D':=min{D,KQS};
    DECREASE(Q,q,D');
    return D'
  end;

```

(If there would exist negative keys, D' could be negative, so DECREASE\* could work like INCREASE. In this case a sequence of O(m) queue operations could solve

the sorting of  $m$  items, and consequently requires  $O(m \log m)$  time. By this reason we do not allow DECREASE\* when there is a negative key.)

One can easily verify the correctness of the algorithms above: if queue  $Q$  satisfies the conditions (1), (2) and (3), then the queue modified by any of operations above also satisfies the conditions.

We now analyse the execution time. Clearly each of MAKEQUEUE, UPDATE and MIN is executed in  $O(1)$  time. INJECT is done in  $O(1)$  time plus the time required for executing MAKESET and REFORM once. POP is done in  $O(1)$  time plus the time required for executing FIND once. On the other hand DECREASE is done in  $O(1)$  time plus the time required for executing FIND and REFORM once. Furthermore the execution time of DECREASE\* is dominated by the time required for executing DECREASE called in DECREASE\*, and is consequently dominated by the time for executing FIND and REFORM called in the DECREASE. The execution time of REFORM is dominated by the time required for executing UNITE operations called in it. Thus the execution time of all the queue operations are dominated by the time required for executing the operations of disjoint set union.

Suppose that a sequence of  $m$  queue operations including  $n$  INJECTs is executed. Then MAKESET is executed  $n$  times, and consequently UNITE is executed at most  $n-1$  times in total during the execution of the sequence. FIND is executed at most  $m$  times. Clearly the structure of the unions is represented by a union tree which is simply a path in our case. Thus we can conclude:

Theorem 1. A sequence of  $m$  queue operations containing  $n$  INJECTs can be executed in  $O(ma(m,n))$  time if the ordinary disjoint set union algorithm [Tar] is used. Moreover the sequence is executed in  $O(m)$  time with  $O(n)$  preprocessing time if the special case disjoint set union algorithm [GT] is used. []

#### 4. APPLICATIONS

In this section we present several applications of the variable priority queue to planar multicommodity flow problems.

A planar network  $N=(G,P,c)$  is a triplet satisfying (1), (2) and (3) below.

(1)  $G=(V,E)$  is a finite undirected simple connected planar graph with vertex set  $V$  and edge set  $E$ .

(2)  $P$  is a set of source-sink pairs  $(s_i, t_i)$ , where source  $s_i$  and sink  $t_i$  are distinct vertices. Both source and sink are often called terminals.

(3)  $c: E \rightarrow R^+$  is the capacity function. ( $R$  (or  $R^+$ ) denotes the set of (nonnegative) real numbers.)

In what follows, we assume that  $G$  has  $n$  vertices and  $P$  contains  $k$  source-sink pairs, i.e.  $|V|=n$  and  $|P|=k$ . Each source-sink pair  $(s_i, t_i)$  of  $N$  is associated with a positive demand  $d_i$ . Although  $G$  is undirected, we orient the edges of  $G$  arbitrarily so that the sign of a value of a flow function can indicate the real direction of the flow through an edge. A set of functions  $\{f_1, f_2, \dots, f_k\}$  with each  $f_i: E \rightarrow R$  is  $k$ -commodity flows of demands  $d_1, d_2, \dots, d_k$  if it satisfies (a) and (b) below.

(a) For each  $e \in E$

$$\sum_{1 \leq i \leq k} |f_i(e)| \leq c(e);$$

(b) Each  $f_i$  satisfies

$$IN(f_i, v) = OUT(f_i, v)$$

for each  $v \in V - \{s_i, t_i\}$ , and

$$OUT(f_i, s_i) - IN(f_i, s_i) = IN(f_i, t_i) - OUT(f_i, t_i) = d_i,$$

where  $IN(f_i, v)$  is the total amount of flow  $f_i$  of commodity  $i$  entering  $v$ , and  $OUT(f_i, v)$  is the total amount of flow  $f_i$  emanating from  $v$ .

We now define several classes of planar networks  $N=(G, P, c)$ .

(0) Class  $C_0$ : Graph  $G$  is a cycle.

(1) Class  $C_1$ : One face boundary  $B_1$  of  $G$  is specified, and all the source-sink pairs are located on  $B_1$ .

(2) Class  $C_{12}$ : Two face boundaries  $B_1$  and  $B_2$  of  $G$  are specified, and each of the source-sink pairs lies on  $B_1$  or  $B_2$ . That is, the set  $P$  is partitioned into  $P_1$  and  $P_2$  so that

if  $(s_i, t_i) \in P_1$  then  $s_i, t_i \in B_1$ ; and

if  $(s_i, t_i) \in P_2$  then  $s_i, t_i \in B_2$ .

(3) Class  $C_{01}$ : one face boundary  $B_1$  together with a vertex  $v_c$  on  $B_1$  are specified, and some of the source-sink pairs are located on  $B_1$ , while the sinks of all the other pairs must lie on  $v_c$  but their sources can lie anywhere in  $G$ . That is, the set  $P$  is partitioned into  $P_0$  and  $P_1$  so that

if  $(s_i, t_i) \in P_0$  then  $t_i = v_c$ ; and

if  $(s_i, t_i) \in P_1$  then  $s_i, t_i \in B_1$ .

Polynomial algorithms have been obtained for the multicommodity flow problem for these classes of planar networks. Matsumoto, Nishizeki and Saito gave an algorithm which finds multicommodity flows in a network belonging to class  $C_1$



in  $O(kn+nT(n))$  time [MNS], where  $T(n)$  denotes the time required for finding the single-source shortest paths in a planar graph which has  $n$  vertices and nonnegative edge weights. Since  $C_0 \subset C_1$ , the algorithm can find multicommodity flows in a network  $N \in C_0$ . In this case, the planar graph for which we must solve the single-source shortest path problem is indeed a star, and consequently  $T(n) = O(n)$ . Therefore the algorithm runs in  $O(kn+n^2)$  time for  $N \in C_0$ . On the other hand we gave algorithms which find multicommodity flows in networks in classes  $C_{12}$  and  $C_{01}$ , and run in  $O(n(k + \min\{b_1, b_2\}T(n)))$  time and  $O(n(b_1^2 + T(n)))$  time, respectively [SNS]. Where  $b_1 = |B_1|$  and  $b_2 = |B_2|$ .

Using the variable priority queue, we can improve the time complexity of the algorithms above. For a network  $N \in C_0$  we can obtain a representation of multicommodity flows in  $O(k+n)$  time as shown later. Furthermore for a network  $N \in C_1$ , we can find values  $f_i(e)$  for a single fixed edge  $e \in B_1$  and all  $(s_i, t_i) \in P$  in  $O(k+T(n))$  time. For a network  $N \in C_{12} \cup C_{01}$  we can implement the algorithms in [SNS] to run in  $O(kn+nT(n))$  time.

In the remaining of this section, we present algorithms for class  $C_0$ . First we implement an algorithm to test the feasibility, that is, to decide whether a given network  $N = (G, P, c) \in C_0$  has multicommodity flows. For each  $e_i, e_j \in E$ ,  $e_i \neq e_j$ , define

$$m(e_i, e_j) = c(e_i) + c(e_j) - \sum \{d_i | s_i \text{ and } t_i \text{ lie in distinct components of } G - \{e_i, e_j\}\}.$$

The following theorem is an immediate consequence of a result in [OS].

Theorem 2. Network  $N \in C_0$  has multicommodity flows if and only if  $m(e_i, e_j) \geq 0$  for all  $e_i, e_j \in E$ ,  $e_i \neq e_j$ . []

Clearly values  $m(e_i, e_j)$  can be computed in  $O(k+n)$  time for a fixed edge  $e_i \in E$  and all edges  $e_j \in E$ . Therefore a straightforward method which computes all  $m(e_i, e_j)$  spends  $O(kn+n^2)$  time to test the feasibility. However, using the variable priority queue we can test the feasibility in  $O(k+n)$  time as follows. Let  $v_0, v_1, \dots, v_{n-1}$  be the sequence of vertices appearing on cycle  $G$  in clockwise order, and let  $e_i = (v_i, v_{i+1})$ ,  $i = 0, 1, \dots, n-1$ , where conventionally  $v_n = v_0$ . The following procedure MARGIN computes values  $m(e_i) = \min\{m(e_i, e_j) | e_j \in E - e_i\}$  for all  $e_i \in E$  total in  $O(k+n)$  time. The key idea to note is that values  $m(e_i, e_j)$  can be efficiently updated from values  $m(e_{i-1}, e_j)$  if the variable priority queue is used.

```

procedure MARGIN;
  begin
    MAKEQUEUE(Q);
    for each edge  $e_j$ ,  $j=1,2,\dots,n-1$  do INJECT(Q, $e_j$ , $m(e_0,e_j)$ );
    {each edge  $e_j$  in Q has key  $m(e_0,e_j)$ }
     $m(e_0) := \text{MIN}(Q)$ ;
    for each edge  $e_i$ ,  $i=1,2,\dots,n-1$  do
      begin
        { $Q=e_i e_{i+1} \dots e_{i-2}$ , and currently each edge  $e_j$  in Q has
        key  $m(e_{i-1},e_j)$ . Keys are now updated to give  $m(e_i,e_j)$ }
        POP(Q); {delete  $e_i$  from Q}
        INJECT(Q, $e_{i-1}$ , $2c(e_{i-1})$ );
        UPDATE(Q, $c(e_i)-c(e_{i-1})+\Sigma\{d_\ell | s_\ell \text{ or } t_\ell \text{ is } v_i\}$ );
        for each terminal  $s_\ell$  (not necessarily source) on  $v_i$  do
          begin
            let  $e_j$  be the edge joining  $t_\ell$  and the clockwise next
            vertex;
            DECREASE(Q, $e_j$ , $2d_\ell$ )
          end;
        {each edge  $e_j$  in Q has key  $m(e_i,e_j)$ }
         $m(e_i) := \text{MIN}(Q)$ 
      end
    end;
  
```

During one execution of MARGIN the queue operations are executed  $O(k+n)$  times in total, and consume  $O(k+n)$  time by Theorem 1. Clearly the other task can be done in  $O(k+n)$  time. Thus we can conclude:

Theorem 3. The feasibility of a network in  $C_0$  can be tested in  $O(k+n)$  time. []

Next we give an algorithm for computing  $f_\ell(e_0)$  for a single edge  $e_0$  and all  $(s_\ell, t_\ell) \in P$ . Let network  $N \in C_0$  satisfy  $m(e_i, e_j) \geq 0$  for all  $e_i, e_j \in E$ ,  $e_i \neq e_j$ . We may assume that, for every source-sink pair  $(s_\ell, t_\ell)$ , first the source  $s_\ell$  and then the sink  $t_\ell$  appear on the cycle  $G$  clockwise starting from  $v_0$ , that is, if  $s_\ell = v_i$  and  $t_\ell = v_j$  then  $i < j$ .

```

procedure MFLOW;
  begin
    MAKEQUEUE(Q);
    for each edge  $e_i$ ,  $i=1,2,\dots,n-1$  do INJECT(Q, $e_i$ , $m(e_0,e_i)$ );
     $D := \text{MIN}(Q)$ ;
     $c(e_0) := c(e_0) - D$ ; {reduce the residual capacity of  $e_0$ }
    UPDATE(Q, $-D$ ); {update keys due to the reduction}
    INJECT(Q, $e_0$ , $2c(e_0)$ );
  
```

```

for each vertex  $v_i$ ,  $i=1,2,\dots,n-1$  do
  for each source  $s_\ell$  on  $v_i$  do
    begin
      let  $e_j$  be the edge joining  $t_\ell$  and the clockwise next
      vertex;
       $f_\ell(e_0) := \text{DECREASE} * (Q, e_j, 2d_\ell) / 2$ 
      { $f_\ell(e_0) = \min\{c(e_0), d_\ell, \min\{m(e_p, e_r) / 2 \mid v_j, v_p, v_0, v_r \text{ and } v_i$ 
      appear clockwise on the cycle in this order}\}}
    end
  end;

```

During one execution of MFLOW the queue operations are executed  $O(k+n)$  times in total, and consume  $O(k+n)$  time. Clearly the other task in MFLOW can be done in  $O(k+n)$  time. Thus the flow values  $f_\ell(e_0)$  can be computed in  $O(k+n)$  time for a single edge  $e_0$  and all  $(s_\ell, t_\ell) \in P$ . Furthermore, from these values, one can easily find flow values for all other edges since a cycle graph has exactly two paths between  $s_\ell$  and  $t_\ell$ , the clockwise path and counterclockwise one. Therefore we can conclude:

Theorem 4. A representation of multicommodity flows in a network  $N \in C_0$  can be obtained in  $O(k+n)$  time. []

#### References

- [AHU] A. V. Aho, J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.
- [GT] H. N. Gabow and R. E. Tarjan, A linear time algorithm for a special case of disjoint set union, Jour. Comput. Syst. Sci., 30, pp.209-221, 1985.
- [GMG] Z. Galil, S. Micali and H. Gabow, An  $O(EV \log V)$  algorithm for finding a maximal weighted matching in general graphs, SIAM J. Comput., 15, 1, pp.120-130, 1986.
- [MNS] K. Matsumoto, T. Nishizeki and N. Saito, An efficient algorithms for finding multicommodity flows in planar networks, SIAM J. on Compt., 14, 2, pp.289-302, 1985.
- [OS] H. Okamura and P. D. Seymour, Multicommodity flows in planar graphs, Journal of Combinatorial Theory, Ser. B, 31, pp.75-81, 1981.
- [SNS] H. Suzuki, T. Nishizeki and N. Saito, Multicommodity flows in planar undirected graphs and shortest paths, Proc. 17th Annual ACM Symp. on Theory of Computing, pp.195-204, 1985.
- [Tar] R. E. Tarjan, Data Structures and Network Algorithms, SIAM, Philadelphia, PA, 1983.