On Detection of Bounded Global Predicates

I-CHEN WU AND LOON-BEEN CHEN

Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan
Email: *icwu@csie.nctu.edu.tw* and *lbchen@csie.nctu.edu.tw*

Distributed programs often follow some bounded global predicates, for example, the total number of certain tokens is always the same or bounded in a range. In order to detect bounded global predicates, we can first derive the minimum and maximum global snapshots and then check if the minimum and maximum are out of the range. Recently, Chase and Garg proposed an efficient method to derive the minimum global snapshot by reducing this problem to a maximum network flow problem. A restriction of this method is that all message values (e.g., the token number in messages) must be zero and all process state values (e.g., the token number in processes) must be non-negative. In this paper, we propose an elegant technique, called normalization. By using this technique, we can remove the above restriction and also derive the minimum and maximum global snapshots at the same time.

Received March 1, 1997; revised May 11, 1998

1. INTRODUCTION

Error detection and debugging have been very important when programmers develop code. Most previous experiences and research reports showed that error detection and debugging are very time consuming in a software development cycle [1]. This is because a bug may happen in an unexpected way at an unexpected spot. In single-processor systems, users usually debug programs by setting breakpoints in programs and then tracing the code step by step. Sometimes, programmers also put some assertions into the code in order to detect the correctness of the code.

With the rapid development of networks and distributed systems, programming on distributed environments is becoming more common. However, the difficulty of distributed programming is much higher than that of sequential programming. Let us consider an example of debugging a distributed program on two processors. If we want to halt at a certain breakpoint of the program on one processor, it is very hard to halt the program on the other processor simultaneously. This makes distributed debugging very difficult. Since distributed debugging is difficult, error detection in a distributed program becomes more significant.

It is well understood that distributed programs are usually designed to obey certain invariant conditions [2]. For example, in a distributed program, there may be a number of tokens distributed over processors (e.g., the token may represent the number of resources and critical sections), and the number of these tokens is bounded in a range at any snapshot, no matter how tokens are moved over different processors. The conditions are usually formulated as predicates, called global predicates in [3].

In fact, it is non-trivial to detect the global predicates because message-carrying tokens can only be detected at the sending and receiving times. Therefore, if we need to detect global predicates, we need to keep track of all process states and then judge from all the states whether the global predicate holds.

Chase and Garg [4] proved that the problem of general global predicate detection is NP-complete. Most researchers use the following three kinds of approaches to solve global predicate detection problems.

(i) Exhaustively search all possible combinations to detect general global predicates. This approach may be impractical for many applications due to the exponential worst-case time complexity [2].
(ii) Periodically check the satisfiability of global predicates [3, 5]. However, this approach works only for problems with stable global predicates. Stable global predicates are predicates with the following property: once the global predicates turn true, they will remain true forever. However, these methods cannot detect unstable global predicates because such a predicate may be true for a short instant (between two checkpoints).
(iii) For specific problems, detect unstable global predicates in polynomial times. Garg and Waldecker [6] presented a tractable algorithm to detect unstable predicates that are formed by a conjunction of local predicates. Recently, Groselj [7] proposed an interesting method to derive the global state with minimum cost, called the minimum global snapshot [7], by reducing the problem to a maximum network flow (or minimum cut) problem. However, Groselj only considered message costs and assumed that all messages had cost one. Later, Chase and Garg [4] also used the maximum flow algorithm to derive the global state with minimum

cost, but they only considered costs in processes (not in messages).

Although for arbitrary flow networks detecting the maximum cut is an NP-hard problem [8], we show in this paper that the flow networks used for predicate detection do permit polynomial time detection of the maximum cut. We first show a straightforward technique that can be applied to systems, where messages do not carry tokens, such as those considered by Chase and Garg. Our technique allows both maximum and minimum cuts to be calculated, even when the number of tokens in a state is allowed to be negative. We then propose a technique, called normalization, which allows this technique to be applied to systems where messages carry tokens.

The remainder of this paper is organized as follows. In Section 2, we describe our model and the notation used in this paper. Section 3 presents the normalization technique and derives the minimum and maximum global snapshots from this technique. Finally, we conclude our results in Section 4.

2. MODEL AND NOTATION

A distributed program is composed of processes communicating via a network. These processes share no memory and no global clock. Each pair of these processes needs to communicate via a channel of the network. The state of such a program is distributed over these processes and channels at each snapshot.

2.1. Events

The states of processes and channels change only when events [9] (atomic actions) are executed. There are three kinds of events occurring on each processor P with which we are concerned:

- **internal event**: does a local computation;
- **send event**: sends a message from process P to another via a channel;
- **receive event**: receives a message from another process via a channel.

Note that each process should start with an initial internal event and end with a final internal event.

In order to define the chronological order of events, we define that event e_i immediately happens before event e_j, if and only if one of the two following conditions holds.

(i) Events e_i and e_j happen in the same process, the time of e_i (happening) is earlier than that of e_j, and no other event happens between these two events in the same process.
(ii) Event e_i is the send event of a message and event e_j is the receive event of the same message.

Furthermore, we define that event e_i happens before event e_j, denoted by $e_i \rightarrow e_j$, if and only if one of the following two conditions holds:

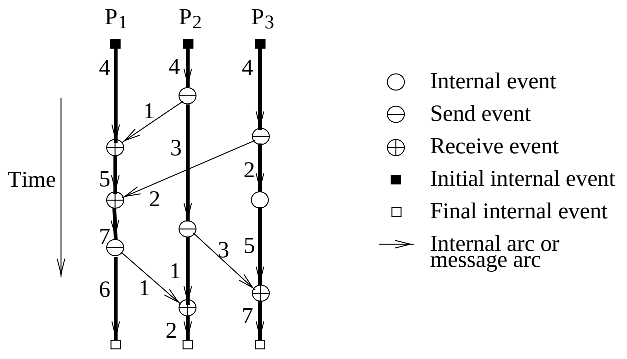


FIGURE 1. An event graph.

(i) event e_i immediately happens before event e_j;
(ii) there exists another event e_k with the relations $e_i \rightarrow e_k$ and $e_k \rightarrow e_j$.

2.2. Event graph

We can use an event graph to represent a run of a distributed program as follows. (i) A vertex denotes an event. (ii) If an event e_i immediately happens before event e_j, there is a corresponding arc, denoted by (e_i, e_j), from e_i's corresponding vertex to e_j's. An arc (e_i, e_j) is called a message arc if there is a corresponding in-transit message from event e_i to e_j. Otherwise, an arc is called an internal arc because it corresponds to the internal event transition inside a process. Clearly, each process has an internal path from the vertex of its initial event to the vertex of its final event without going through any message arcs. An event graph is illustrated in Figure 1. For each internal arc $a = (e_i, e_j)$, we denote the token number of the corresponding process by S_a, after executing the event corresponding to e_i. For each message arc a, we denote the number of tokens of the message by S_a.

2.3. Global states

Consider a possible run of a distributed program. The system can proceed from one state to another by executing events.

Definition 2.1. *A set E_C of events is said to be consistent if event $e \in E_C$ and event $e' \rightarrow e$ imply $e' \in E_C$. A global state is a collection of states, one from each process and channel, right after executing a consistent set of events.*

The set of all global states of an execution of a distributed program forms a lattice. In the lattice, a node (global state) S_1 has a link to another node S_2 if the system can proceed from S_1 to S_2 by executing only one event. Figure 2a shows the event graph of a distributed program and Figure 2b shows the corresponding lattice. A possible run of a distributed program can be viewed as a path in the lattice from the initial node (initial global state) to the final node (final global state). For example, the path depicted with bold lines in Figure 2b represents a possible execution order of the events occurring in the program.

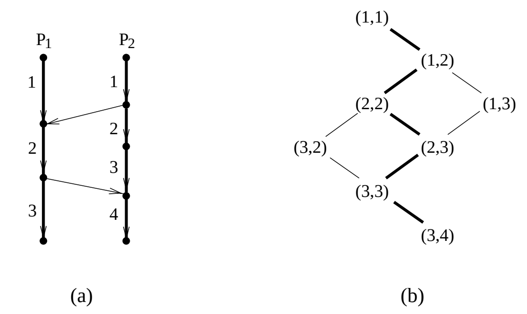
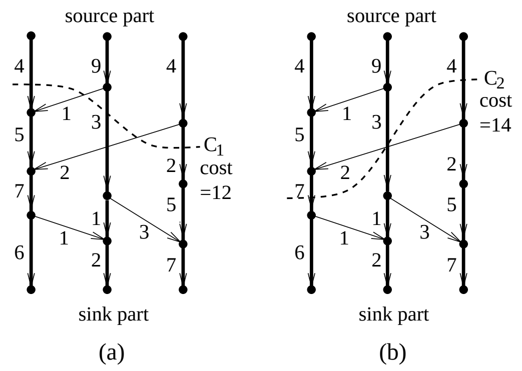


FIGURE 2. (a) The event graph of a distributed program. (b) The lattice of (a).



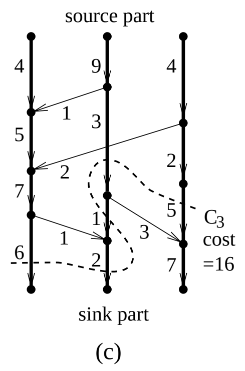


FIGURE 3. (a) Consistent cut C_1, (b) inconsistent cut C_2 and (c) inconsistent cut C_3.

A global predicate is a predicate on global states. In this paper, we are interested in detecting error states where the given global predicates hold, for example the total number of certain tokens is not bounded in a given range. Such global predicates are called bounded global predicates in this paper and can be formulated as: (i) at some global state of the system, $N < K$ (or $N \leq K$), or (ii) at some global state of the system, $N > K$ (or $N \geq K$), where N is the total number of tokens and K is a constant.

For example, assume that in a distributed system the token number is supposed to be smaller then K, but in a global state S the token number is actually larger than K. Then, all the runs containing the global state S have errors. Although this does not guarantee that all runs will have an error global state, detection of such possible errors is very significant since it proves that the system is not robust.

2.4. Cuts

In a common graph, if we separate the vertices into two sets, a cut is the set of all the arcs each of which is incident to these two disjoint vertex sets. In an event graph G, we define that a cut must partition the event graph into two disjoint graphs such that one, called the source part and denoted by $G_s = (V_s, E_s)$, contains all the initial internal events and the other, called the sink part and denoted by $G_t = (V_t, E_t)$, contains all the final internal events. Thus, it is trivial to see that the cut has at least one arc in each internal path. The cost of a cut is defined as follows:

$$\sum_{\forall a:a=(u,v),u\in V_s,v\in V_t} S_a.$$

For example, in Figure 3, the costs of cuts C_1, C_2 and C_3 are 12, 14 and 16, respectively. The minimum (maximum) cut is the cut with the least (largest) cost among all cuts. The least (largest) cost is called the minimum (maximum) cut cost.

A cut of the event graph is consistent if and only if the set of all events corresponding to vertices in the source part are consistent. From this definition, the following lemma apparently holds.

LEMMA 2.1. *For an event graph, a cut C is consistent if and only if each arc on C is from the source part to the sink part.*

Note that the cost of a consistent cut may represent the number of tokens of the corresponding global state. Figure 3 illustrates cuts of an event graph. In this figure, only cut C_1 is consistent, since for cut C_2 the message arc with cost 2 is from the sink part to the source part and for C_3 the internal arc with cost 1 is from the sink part to the source part.

The minimum (maximum) consistent cut is the consistent cut with the least (largest) cost among all consistent cuts. The minimum (maximum) consistent cut cost is the cost of the minimum (maximum) consistent cut. The minimum (maximum) consistent cut is also called the minimum (maximum) global snapshot.

2.5. Predicate detection

In our model, we assume that the event graph with costs on arcs is given in advance. In practice, Garg and Waldecker [6] suggested that in each event each process sends its state information (such as the number of tokens currently held) to a process, called the checker process, which runs an algorithm to detect the global predicate. In order to detect the bounded global predicates, we need to derive the minimum and maximum global snapshots (or the minimum and maximum consistent cuts) in a given event graph and then check if its cost is less than or greater than the constant K. In the next section, we will investigate the methods of deriving the minimum and maximum global snapshots.

3. MINIMUM AND MAXIMUM GLOBAL SNAPSHOTS

In this section, we will derive the minimum and maximum global snapshots in a given event graph. In Subsection 3.1, we will describe the basic method modified from Groselj's paper [7] that can efficiently derive the minimum global snapshot for all event graphs without negative-cost arcs. In Subsection 3.2, we propose two operations, translation and reflection, by which we can derive both minimum and maximum snapshots for zero-message-cost event graphs. Zero-message-cost event graphs are event graphs whose message arcs all have zero cost (whose internal arcs may have negative costs). In Subsection 3.3, we propose an operation, normalization, to derive both minimum and maximum global snapshots for all event graphs.

3.1. Basic technique for the minimum consistent cut

Groselj [7] proposed a method to derive the minimum consistent cut cost of an event graph (without negative-cost arcs) and which is based on flow network algorithms. In this subsection, we describe a simple method which is modified from Groselj's.

In fact, an event graph can also be viewed as a flow network (defined in Definition 3.1) with multiple sources and sinks [10]. All the initial (final) internal events correspond to the source (sink) nodes and the value on each arc corresponds to the capacity of the arc.

DEFINITION 3.1. *A flow network $N = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a non-negative capacity $c(u, v) \geq 0$. Some nodes are designated sources and some others are designated sinks. A cut is a set of arcs all incident to two disjoint vertex sets partitioned from V, where one set with sources is called the source set and the other with sinks is called the sink set. The capacity of a cut is the total capacity of all arcs (on the cut) from the source set to the sink set. A minimum cut of a flow network is the cut with the least capacity. The least capacity is also called the min-cut capacity.*

The key to Groselj's technique is to reduce an event graph to another flow network while keeping the following property satisfied:

- the min-cut capacity of the reduced flow network equals the minimum consistent cut cost of the original event graph.

Since the minimum cut problem in a flow network is equivalent to the maximum network flow problem as shown by Ford and Fulkerson [11], we can use some efficient maximum network flow algorithm, such as that of [12], to find the minimum cut cost.

In order to reduce an event graph to a flow network with the above property, we use the following reduction operation.

R1 For each arc (v_i, v_j), add a reverse arc (v_j, v_i) with the value ∞.

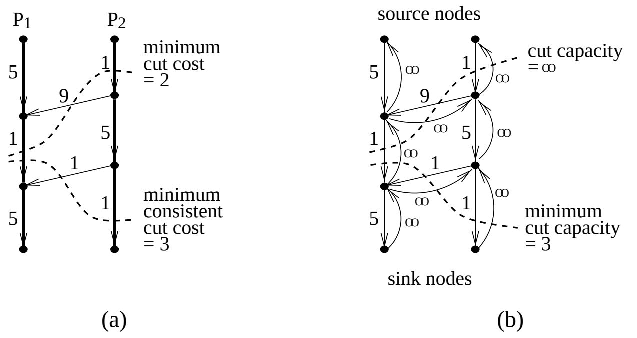


FIGURE 4. (a) Before operation R1 and (b) after operation R1.

Note that the initial (final) internal nodes become the source and sink nodes.

Lemma 3.1 shows that after the reduction operation R1 the min-cut capacity of the reduced flow network is the same as the minimum consistent cut cost of the original event graph.

Since the vertex set is unchanged in the reduction algorithm R1, we define that the cut C_G in the original graph G is equivalent to the cut C_N in the reduced network N if and only if the two vertex sets partitioned by cut C_G in G are the same as those partitioned by cut C_N in N.

LEMMA 3.1. *Given an event graph G, use the reduction operation R1 as described above to reduce G to a flow network N. Then, the minimum cut in N is also a minimum consistent cut of G. Also, the min-cut capacity in N equals the minimum consistent cut cost of G.*

Proof. For each cut C_N in the reduced flow network N, its equivalent cut C_G in G is either consistent or inconsistent. Assume that cut C_G is consistent. According to Lemma 2.1, each arc in cut C_G must be from the source part to the sink part. Thus, after the reduction operation R1, the equivalent cut still has the same cost (or capacity) since the costs of reversed arcs are not counted, as illustrated in Figure 4.

If C_G is inconsistent, there exists an arc (u, v) where u is in the sink part and v is in the source part from Lemma 2.1. On the equivalent cut C_N in N, since the reversed arc (v, u) (added from the reduction operation R1) is from the source part to the sink part, the capacity of the arc, ∞, will be added into the total capacity of the cut C_N, which will become ∞, as illustrated in Figure 4.

From the above discussion, the min-cut capacity in N is also the minimum consistent cut cost in G. The minimum cut in N is also a minimum consistent cut of G. □

3.2. Zero-message-cost event graphs

Chase and Garg derived the minimum consistent cut for each zero-message-cost event graph [4]. In this subsection, we propose two operations, translation and reflection, in order to derive both minimum and maximum consistent cuts for each zero-message-cost event graph, respectively.

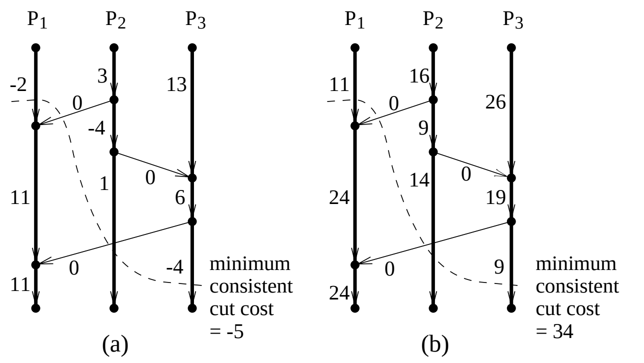

P1
P2
P3
minimum consistent cut cost = -5
(a)
minimum consistent cut cost = 34
(b)

FIGURE 5. (a) The original zero-message-cost event graph and (b) the translated graph ($M = 13$).

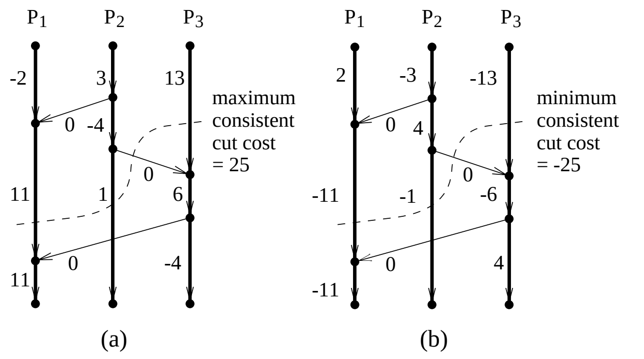

P1
P2
P3
maximum consistent cut cost = 25
(a)
minimum consistent cut cost = -25
(b)

FIGURE 6. (a) The original zero-message-cost event graph and (b) the reflected graph.

3.2.1. Translation and the minimum consistent cut

DEFINITION 3.2. *For a given event graph, a translation operation does the following: for each internal arc, add M to its cost, where $M = max(|S_a|)$ for all internal arc costs S_a.*

After making the translation operation (as defined in Definition 3.2) on a zero-message-cost event graph, we can derive the following two properties on the graph (as illustrated in Figure 5).

- Since each consistent cut only cuts across one arc on each internal path, each consistent cut cost in the event graph will increase by a constant pM. Therefore, the minimum consistent cut remains the same.
- Each arc in the event graph has a non-negative cost.

As stated above, after making the translation operation on a zero-message-cost event graph, we can apply the basic method described in the previous subsection to this graph in order to solve the minimum consistent cut (due to the second property).

3.2.2. Reflection and the maximum consistent cut

DEFINITION 3.3. *For a given event graph, a reflection operation does the following: for each internal arc cost S_a, set the cost to $-S_a$.*

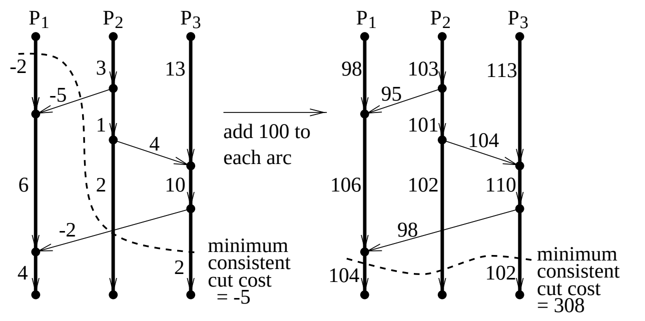

P1
P2
P3
add 100 to each arc
minimum consistent cut cost = -5
minimum consistent cut cost = 308

FIGURE 7. Adding 100 to each arc cost.

After making the reflection operation (as defined in Definition 3.3) on a zero-message-cost event graph, we can derive the following property on the graph (as illustrated in Figure 6):

- Each consistent cut cost C in the event graph will become $-C$. Hence, the original maximum consistent cut becomes the minimum consistent cut in the new event graph.

From above, after making the reflection operation on a zero-message-cost event graph, we can apply the translation method to this graph in order to solve the maximum consistent cut of the new event graph.

3.3. Normalization

Although the translation and the reflection operations can deal with zero-message-cost event graphs, these two operations cannot be applied to all event graphs directly. For example, if the translation operation is also applied to message arcs in Figure 7, in order to make all costs non-negative, all consistent cuts may cut across different numbers of message arcs and therefore their costs will increase differently.

In this subsection, we propose an operation, called normalization, to reduce a general event graph to a zero-message-cost event graph. The key of the normalization technique is to clear the costs of message arcs to zeros by shifting the costs of message arcs to internal arcs without changing any consistent cut cost. The normalization operation repeats the following primitive operation until each message arc cost S_{a_m} is zero:

(i) Find a message arc $a_m = (e_i, e_j)$ with $S_{a_m} \neq 0$.
(ii) Add S_{a_m} into the cost of each arc (u, v) on the internal path containing e_i, where $e_i \rightarrow v$.
(iii) Add $-S_{a_m}$ into the cost of each arc (u, v) on the internal path containing e_j, where $e_j \rightarrow v$.
(iv) Clear the cost of arc a_m to zero (that is, add $-S_{a_m}$ into the cost of arc a_m).

Figure 8 illustrates arc costs before and after normalization. It is easy to observe from the figure that any cut cost does not change after normalization. Lemma 3.2 proves this property.

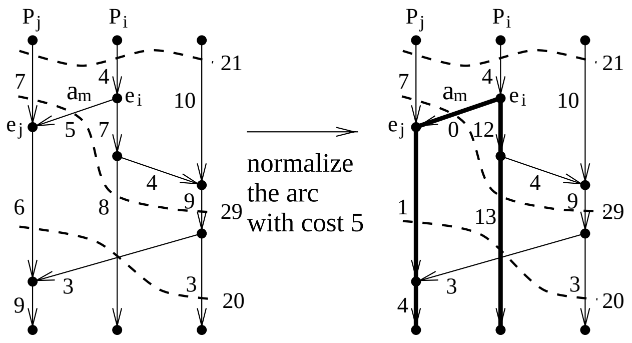


FIGURE 8. Normalizing an arc (see the emboldened arrows).

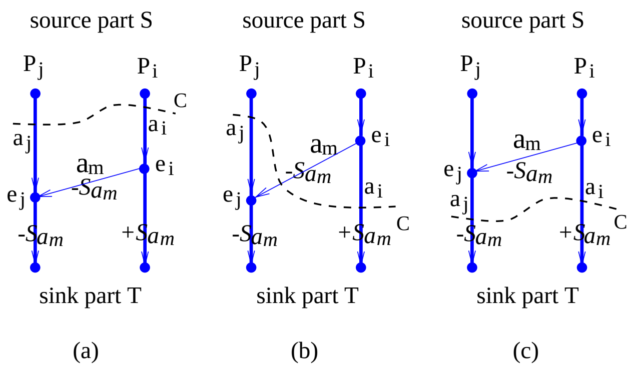


FIGURE 9. Normalizing an arc does not change cut costs.

LEMMA 3.2. *Let G' be normalized from event graph G as described above. The cost of each consistent cut of G equals the cost of the corresponding cut of G'.*

Proof. It suffices to prove that, for each primitive normalization operation, no consistent cut cost is changed. Consider a primitive operation normalizing a message arc $a_m = (e_i, e_j)$, where e_i and e_j are vertices on the internal paths P_i and P_j, respectively. Let C be a consistent cut which partitions the vertices into the source part S and the sink part T; then there are three ways C can cut across internal paths P_i and P_j.

(i) Suppose that $e_i \in T$ and $e_j \in T$ (as shown in Figure 9a). In this case, the cut cost obviously is unchanged because all arc values in the cut are unchanged.

(ii) Suppose that $e_i \in S$ and $e_j \in T$ (as shown in Figure 9b). Assume that C cuts internal arcs a_i and a_j in P_i and P_j, respectively. In this case, the cut must also cut across the arc a_m. Then, the value S_{a_i} increases by S_{a_m}, S_{a_j} remains the same and S_{a_m} decreases (to zero) by S_{a_m}. Thus, the total cut cost is still the same.

(iii) Suppose $e_i \in S$ and $e_j \in S$ (as shown in Figure 9c). Assume that C cuts internal arcs a_i and a_j in P_i and P_j, respectively. In this case, the value S_{a_i} increases by S_{a_m} and S_{a_j} decreases by S_{a_m}. Therefore, the total cut cost is still the same.

As illustrated above, the total cut cost is still the same for all cases. That is, for all consistent cuts, their cut costs are still the same. □

According to Lemma 3.2, any general event graph can be normalized to become a zero-message-cost event graph. Then, after the normalization operation, we can derive the minimum and maximum consistent cuts of all event graphs by using the approach discussed in Subsection 3.2.

4. DISCUSSIONS

Traditionally, researchers [4, 7] can only derive the minimum consistent cut cost with non-negative arc costs. In this paper, we propose a new and elegant technique, normalization, by which we can efficiently find the minimum and maximum consistent cut cost of an event graph without limiting arc costs to non-negatives. Our results can be extended to the detection of bounded global predicates. This is useful for a control system (e.g., a dynamic load balancing system) whose consistent costs should be bounded in a range.

The algorithms for the minimum and maximum consistent cut problems described in Subsection 3.2 basically consist of the following operations.

- The maximum network flow operation: we can use the fastest maximum network flow algorithm, the preflow-push algorithm [12] that runs in $O(nm \log(n^2/m))$, where m is the number of edges and n is the number of vertices. In our problem, each vertex (corresponding to an event) in an event graph has at most three incident arcs. Therefore, $m = \Theta(n)$ in an event graph and the time complexity for the preflow-push algorithm is $O(n^2 \log n)$.
- The translation operation: since it only finds the maximum arc cost M and then changes each arc cost, the time complexity is $O(n)$.
- The reflection operation: since it only changes each arc cost once, the time complexity is $O(n)$.
- The normalization operation: for each message arc $a = (u, v)$, we mark $+S_a$ at event u and mark $-S_a$ at event v. Then for each internal path, update each arc cost by scanning from its initial internal arc to its final internal arc. Thus, the normalization operation can be finished in $O(n)$.

Thus, the time complexity for detection of bounded global predicates is $O(n^2 \log n)$ in total.

ACKNOWLEDGEMENTS

The authors would like to thank the National Science Council of the Republic of China for financially supporting this research under contract No. NSC-83-0408-E-009-027. The authors would also like to thank the referees for their valuable comments.

REFERENCES

[1] Pfleeger, P. L. (1991) *The Production of Quality Software*. Macmillan.

[2] Cooper, R. and Marzullo, K. (1991) Consistent detection of global predicates. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, May 20–21, Santa Cruz, CA., pp. 167–174.

[3] Chandy, K. M. and Lamport, L. (1985) Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, **3**, 63–75.

[4] Chase, C. M. and Garg, V. K. (1995) Efficient detection of restricted classes of global predicates. In *9th Int. Workshop on Distributed Algorithms*, September 13–15, Le Mont-Saint-Michel, France. Springer, Heidelberg.

[5] Bouge, L. (1987) Repeated snapshots in distributed systems with synchronous communication and their implementation in CSP. *Theor. Comput. Sci.*, **49**, 145–169.

[6] Garg, V. K. and Waldecker, B. (1994) Detection of weak unstable predicates in distributed programs. *IEEE Tran. Parallel Distrib. Syst.*, **5**, 299–307.

[7] Groselj, B. (1993) Bounded and minimum global snapshots. *IEEE Parallel Distrib. Technol.*, **1**, 72–83.

[8] Johnson, D. S., Garey, M. R. and Stockmeyer, L. (1976) Some simplified NP-complete graph problems. *Theor. Comput. Sci.*, **1**, 237–267.

[9] Lamport, L. (1978) Time, clocks and the ordering of events in a distributed system. *Commun. ACM.*, **21**, 558–565.

[10] Leiserson, C. E., Cormen, T. H. and Rivest, R. L. (1989) *Introduction to Algorithms*. MIT Press, Cambridge, MA.

[11] Ford, L. R. and Fulkerson, D. R. (1956) Maximal flow through a network. *Can. J. Math.*, **8**, 399–404.

[12] Goldberg, A. V. and Tarjan, R. E. (1988) A new approach to the maximum-flow problem. *J. ACM.*, **35**, 921–940.