Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

On Detection of Bounded Global Predicates

Loon-Been Chen and I-Chen Wu

Department of Computer Science and Information Engineering
National Chiao Tung University
Hsin-Chu, Taiwan
lbchen@csie.nctu.edu.tw and icwu@csie.nctu.edu.tw

Abstract

Distributed programs often follow some bounded global predicates, e.g., 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, Groselj [9] 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 values must be non-negative.

In this paper, we propose an elegant technique, called normalization. *By using this technique, we can easily derive the minimum and maximum global snapshots and also remove the restriction of being non-negative.*

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 part in a software development cycle [12]. This is because a bug may happen in an unexpected way at an unexpected spot. In single-processor systems, users usually debug programs by breaking programs at some points and then tracing the code step by step. Sometimes, programmers also put some assertions in code in order to detect the correctness of code.

With the fast development of network and distributed systems, programming on distributed environment is getting 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. When we want to break at a certain line of the program on one processor, it is very hard to stop 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.

From our past experiences on implementing a large distributed system (for load balancing) in [15], we found that a distributed program usually needs to follow some rules in order to make it easy to run correctly. 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 remains constant or bounded in a range at any snapshot, no matter how tokens are moved over different processors. The kind of rules are usually formulated as predicates, called *global predicates* in [2]. Note that in some cases if some resources have been used or consumed in advance, it is possible to use a negative number to represent the number of tokens consumed in advance.

In fact, it is non-trivial to detect above global predicates because when tokens are sent to another process via a message these tokens are hidden from detection. 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 absolutely.

Chase *et al.* [3] 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.

1. Exhaustively search all the combinations [4] to detect general global predicates. This kind of solutions are intractable in the sense of time complexity.

2. Periodically check satisfiability of global predicates [1, 2]. However, this approach works only for the 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 become true for a short period time (between two check points).

3. 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 conjunction of local states. Miller et. al. [13] and Hufin et. al. [11] respectively investigated some other different global predicates. Recently,

Groselj [9] proposed an interesting method to derive the minimum of the total numbers of tokens at all snapshots, called *the minimum global snapshot* [9], by reducing the detection problem to a maximum network flow (or minimum cut) problem. This method can be easily extended to solve the (unstable) bounded global predicate: the total number of tokens is always bounded from below by a constant at all snapshots. Later, Garg *et al.* [3] developed the same algorithm independently. Since the maximum network flow problem requires the restriction on non-negative edge values, the above approach needs to restrict non-negative state values.

Although Groselj and Garg can derive the minimum global snapshot, we find it non-trivial to reduce the above result to the maximum global snapshot. This is because deriving a minimum network flow is an NP-complete problem[7], much more complex than deriving a maximum network flow.

In order to derive the maximum global snapshot efficiently, we first propose an elegant technique, called *normalization*, in this paper. Then, based on the technique of normalization, we can easily derive the minimum and maximum global snapshots in the same way and we can also remove the restriction of being non-negative.

The remainder of this paper is organized as follows. In Section 2, we describe our model and notations 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 notations

A distributed program is composed of *processes* communicating via a network. These processes share no memory and no global clock. Each pair of processes need to communicate via a *channel* of the network. The state of such a program is distributed over these processes and channels at each snapshot. For simplicity of discussion, we assume in this paper that the system has p processes.

The states of processes and channels change only when *events* [10], atomic actions, are executed. There are three kinds of events on each processor P that we are concerned:

- **Internal event**: does a local computation. It may change the state of process P.
- **Send event**: sends a message from process P to another via a channel. It may also change the state of process P.
- **Receive event**: receives a message from another process via a channel. It may also change the state of process P.

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:

1. 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 the two events in the same process.
2. 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:

1. Events e_i immediately happens before event e_j.
2. There exists another event e_k with the following two relations: $e_i \rightarrow e_k$ and $e_k \rightarrow e_j$.

For simplicity of discussion, we can use an *event graph* to represent a run of a distributed program as follows. (1) Each event is referred to by a vertex. (2) 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, representing the event transition from e_i to e_j. 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.

Since an arc represents an event transition without any other event in-between, we can extend the definition of chronological relation $\rightarrow$ to arcs as follows. For an arc $a = (e_i, e_j)$, $e_i \rightarrow a$ and $a \rightarrow e_j$. Since all the states are changed only by events (as mentioned above), each arc a can use a unique value, denoted by S_a, to represent the corresponding process state value or the message content. Figure 1 illustrates an event graph corresponding to a run of a distributed program. The value on each arc represents, for example, the number of tokens in a process or a message. A complete path is from some initial internal event to some final internal event. From the figure, we can also easily see that for each process there must exist a complete path from its initial to final internal event without going through any message arcs. Such a path is called a *process internal path*, on which all arcs are internal arcs. These process internal paths do not intersect. Arcs between different process internal paths are all messages arcs. In addition, it is also obvious that for each arc there exists a complete path that passes through the arc.

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*, denoted by G_s, contains all the initial internal events and the other called the *sink part*, denoted by G_t, contains all the final internal events. Thus, it is trivial to see that the cut has at least one arc in each process internal path. The cost of a cut is defined as follows:

$$\sum_{\forall a: a=(u,v) \in C, u \in G_s, v \in G_t} S_a$$

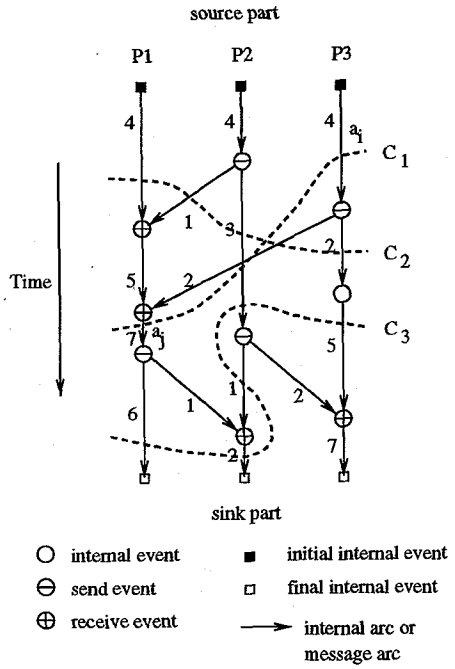

Figure 1: Event graph of a run of a distributed program.

For example, in Figure 1, the costs of cut C_1, C_2 and C_3 are respectively 14, 12 and 16. 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 *consistent cut* is a cut in which each arc a does not happen before another arc a'. For example, cut C_2 in Figure 1 are consistent and cuts C_1 and C_3 are inconsistent. From Lemma 1, a cut C is consistent if and only if each arc on C must be from the source part to the sink part.

Lemma 1 *From the above definition, a cut C is consistent if and only if each arc on C must be from the source part to the sink part.*

Proof. First, we will prove that if a cut C is consistent then each arc on C must be from the source part to the sink part. For each arc a on C, there must exist a complete path passing through arc a. From the definition of consistent cut, cut C will cut across no other arcs in the path. Apparently, arc a must be from the source part to the sink part.

Second, we will prove in the reversed direction that a cut C is consistent if each arc on C is from the source part to the sink part. Let us contradictorily assume that each arc on cut C is from the source part to the sink part, but C is not consistent, say it cuts across two arcs, a_1 and a_2, with $a_1 \rightarrow a_2$. Let arc $a_1 = (v_1, v_1')$ and $a_2 = (v_2, v_2')$, as illustrated in Figure 2. Then, v_1 and v_2 are in the source part and v_1' and v_2' are in the sink part. But, since $v_1' \rightarrow v_2$ (because

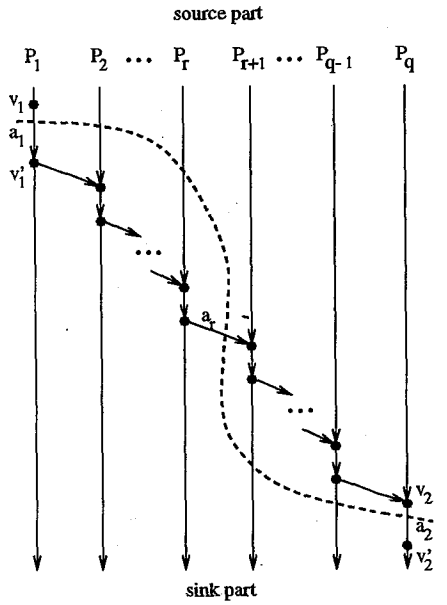

Figure 2: The arc from the sink part to the source part.

$a_1 \rightarrow a_2$), the path from v_1' to v_2 must have an arc (see a_r in Figure 2) from the sink part to the source part. This contradicts to our assumption. Therefore, the cut C must be consistent. □

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. In [9], the minimum (maximum) consistent cut cost is called the minimum (maximum) global snapshot.

The definition of consistent cut above has an important implication: all transitions corresponding to arcs of a consistent cut *may* potentially happen at the same time. The reason is as follows. In a distributed system, it is common that a process may suspend running due to context switching to another job. Since each process in a distributed system may delay operations unexpectedly due to context switching, if we let each process delay an appropriate time, all transitions corresponding to arcs of a consistent cut may happen at the same time. For example, in Figure 1, those arcs in a consistent cut C_2 may happen at the same times after delaying appropriate times in each process as shown in Figure 3. As for cut C_1 in Figure 1, since it is not consistent (event a_i happens before a_j), there is no way adjusting delays to make a_i and a_j happening at the same time.

A global predicate is a predicate on all states of channels and processes. Consider the following example. A distributed program often needs to keep the total number of tokens in all messages and processes bounded from below by a constant. Let the value of a message (internal) arc be the number of tokens in

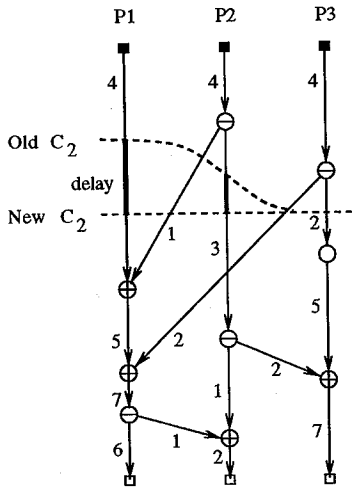

Figure 3: Appropriate delays to make arcs in a consistent cut happening at the same time.

the corresponding message (process) . Then, for this example, the following global predicate holds at each time t,

$$(\sum_{\forall arc a} S_a(t)) \geq L \quad (1)$$

where $S_a(t)$ is the number of tokens in the message or process corresponding to a at time t. However, it is very difficult to have an accurate global clock and to keep checking the global predicate at each time. Therefore, in order to prove that a global predicate holds, most researchers check, instead, if the global predicate holds for all consistent cut C,

$$(\sum_{\forall \text{ arc } a \in C} S_a) \geq L \quad (2)$$

In fact, the detection on consistent cuts (Formula (2)) is more rigorous or stronger than the detection at each time t (Formula (1)) for the following reason. For each time t, there must exist a consistent cut whose arcs covers the states of all processes and channels. Therefore, if Formula (2) holds for all consistent cuts, then Formula (1) holds at all time t.

Furthermore, in order to check if the global predicate on each consistent cut holds, we simply need to derive the minimum global snapshot (or the minimum consistent cut cost) and then check if the cost is greater than the constant L. In this paper, our goal is to detect bounded global predicates. From the above discussion, we can achieve our goal by simply deriving the minimum and maximum global snapshots of an event graph. In the next section, we will focus on the minimum and maximum global snapshots.

3 Minimum and maximum global snapshots

In this section, we will derive the maximum and minimum global snapshots in an event graph. In Subsection 3.1, we will describe the method modified from Groselj's [9] that can efficiently solve the minimum global snapshot problem. In Subsection 3.2, we propose a new technique, called *normalization*, based on which we can elegantly derive the minimum and maximum global consistent cuts as described in Subsection 3.3. Our solution in Section 3.3 does not assume that the arc values should be non-negative as Groselj and Garg's solutions did [3, 9].

3.1 Basic technique for the minimum consistent cut

Groselj [9] proposed a method to derive the minimum consistent cut cost of an event graph based on the common flow network algorithms. In this subsection, we describe a method, based on Groselj's, but simpler than his.

Actually, an event graph can also be viewed as a flow network with multiple sources and sinks [14], defined in Definition 1. 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 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 are designated sinks. A cut is the set of all the arcs each of which is incident to two vertex sets partitioned from V, where all sources are on one set called the source set and all sinks are on the other 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 of 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 to 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 [5], we can use some efficient maximum network flow algorithms such as [8] to find the minimum cut cost.

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

R1 For each arc (v_i, v_j), add a reverse arc (v_j, v_i) with value ∞. Note that the initial (final) internal nodes become the source and sink nodes.

Lemma 2 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.

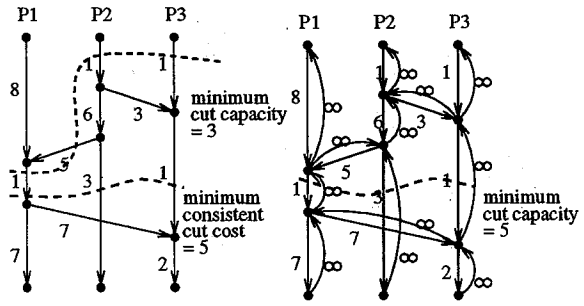

Figure 4: The event graph before and after the reduction operation.

Lemma 2 *Given an event graph G, use the reduction operation R1, described above, to reduce G to a flow network N. Then, the min-cut capacity in N equals to the minimum consistent cut cost of G.*

Proof. Since the reduction operation does not change the vertex set, we define, for simplicity of discussion, that the cut C_G in the event graph G is equivalent to the cut C_N in the reduced flow 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.

For each cut C_N in the reduced flow network N, its equivalent cut C_G in G is either consistent or inconsistent. Suppose that cut C_G is consistent. According to Lemma 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.

Suppose that C_G is inconsistent. From Lemma 1, there exists an arc (u, v) where u is in the sink part and v is in the source part. 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 ∞. Thus, the min-cut capacity in N is also the minimum consistent cut cost in G. □

Figure 4 illustrates that after the reduction operation R1 the minimum cut will not cut across a message arc such that its receive event is in the source part and its send event is in the sink part.

3.2 Normalization

The above approach in the previous subsection still has the following two problems:

1. The arc cost should be non-negative. Intuitively, it seems easy to deal with negative arc costs by adding a large enough value into each arc cost to make all arc costs non-negative. However, since cuts may cut across different numbers of messages arcs, the straightforward method does not work well. This can be illustrated in Figure 5. After adding 100 into each arc value, the minimum consistent cut in the event graph (see the left in this figure) cuts message arcs twice and therefore its cost becomes higher than those with less message arcs. Thus, the cut is no longer the minimum cut in the reduced flow network and we will get the wrong answer for the minimum cut cost.

2. Since the maximum cut problem is intractable, it is not efficient to derive the maximum consistent cut by directly reducing it to the maximum cut problem.

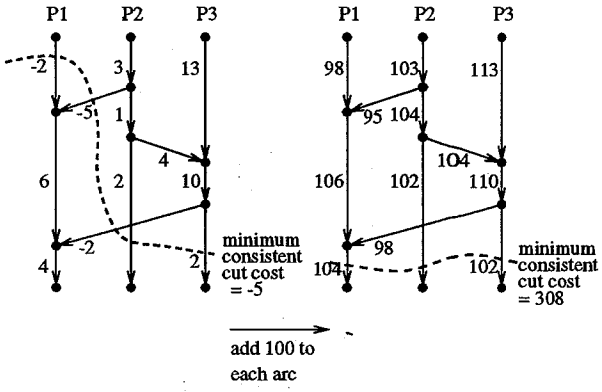

Figure 5: Adding 100 to each arc cost.

In this subsection, we propose a new technique, called *normalization*, to solve the above problems elegantly. The key of the normalization technique is to shift the costs of message arcs to internal arcs in order to clear the costs of message arcs to zeros without changing any consistent cut cost. The normalization operation repeats the following primitive operation until each message arc cost S_{a_m} is zero:

1. Find a message arc $a_m = (v_i, v_j)$ with $S_{a_m} \neq 0$.
2. Add S_{a_m} into the cost of each arc a on the process internal path with v_i, where $v_i \rightarrow a$.
3. Add $-S_{a_m}$ into the cost of each arc a on the process internal path with v_j, where $v_j \rightarrow a$.
4. Clear the cost of arc a_m to zero (that is, add $-S_{a_m}$ into the cost of arc a_m).

Figure 6 illustrates arc costs before and after normalization. It is easy to observe from the figure that any cut cost doesn't change after normalization. Lemma 3 will prove this.

Lemma 3 *Let G' be normalized from event graph G as described above. The cost of each consistent cut of G equals to 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 from vertex v to v'. For each consistent cut, there are three ways to cut across arcs

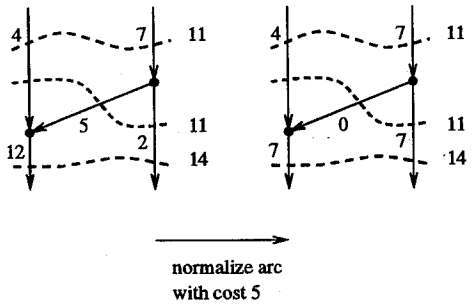

Figure 6: Normalizing an arc.

in the two process internal paths with v and v'. Let the consistent cut cut across arc a in the internal path of process P with v and arc a' in process P' with v'. There are only three cases (as illustrated in Figure 6):

1. Assume $a \rightarrow v$ and $a' \rightarrow v'$. The cut cost obviously is not changed because all arc values are not changed.

2. Assume $v \rightarrow a$ and $a' \rightarrow v'$. In this case, the value S_a increases by S_{a_m} while S_{a_m} decreases to zero. Therefore, the total cut cost is still the same.

3. Assume $v \rightarrow a$ and $v' \rightarrow a'$. In this case, the value S_a increases by S_{a_m} while $S_{a'}$ decreases by S_{a_m}. Therefore, the total cut cost is still the same.

From 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. □

3.3 Minimum and maximum consistent cuts

Based on the normalization techniques, we will show in this subsection that we can easily derive the minimum and maximum consistent cut costs of an event graph even with negative arc costs. We will first derive the minimum consistent cut cost in an event graph and then the maximum consistent cut cost.

In order to derive the minimum consistent cut in an event graph, we will take the following steps:

1. Normalize the event graph.

2. For each internal arc a, add M into S_a, where $M = \max(|S_a|)$ for all internal arc cost S_a.

3. Use the method in Subsection 3.1 to derive the minimum consistent cut.

We illustrate changes of the event graph for each step in Figure 7. Figure 7(a) is the original event graph. Figure 7(b), 7(c), and 7(d) are the reduced flow networks after the first, second and third step, respectively.

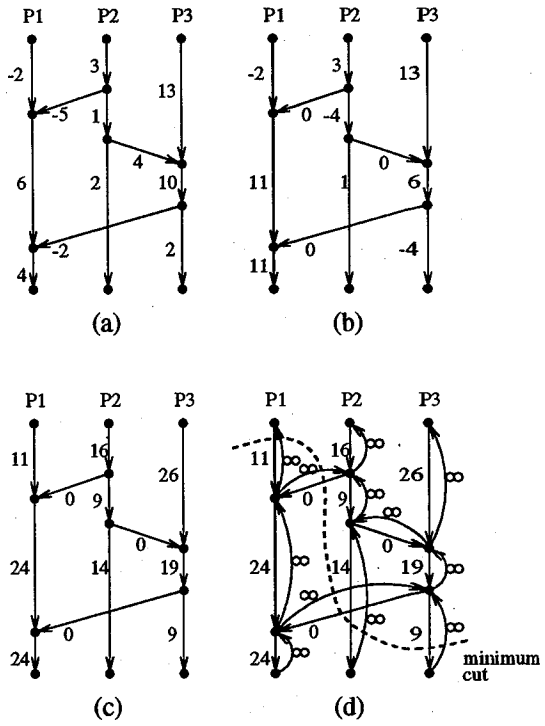

Figure 7: Example of deriving minimum consistency cut: (a) the original event graph, (b) the graph after Step 1, (c) the graph after Step 2 ($M = 13$), and (d) the graph after Step 3.

The operation in the first step does not change any consistent cut cost. The operation in the second step has two effects: (1) increase each consistent cut cost by a fixed number pM because a consistent cut must cut across exactly p internal arcs (whose costs each increases by M) and some message arcs (whose costs are zeros by normalization), as mentioned in the previous subsection; (2) make all arc cost non-negative. Due to the first effect, the minimum consistent cut in the original graph is still the minimum consistent cut in the new graph. Due to the second effect, we can use the method described in Subsection 3.1 to derive the minimum consistent cut. From Subsection 3.1, the minimum cut is the minimum consistent cut in the graph after step 2, which is also the minimum consistent cut in the original graph.

In order to derive the maximum consistent cut cost in an event graph with negative arc costs, we simply need to change the second step as follows.

- For each internal arc a, change S_a to $M - S_a$, where $M = \max(|S_a|)$ for all internal arc cost S_a.

This operation has two effects: (1) each consistent cut cost K in the original event graph becomes $pM - K$; (2) all arc costs are non-negative. Thus, the maximum consistent cut in the original graph will become the minimum consistent cut.. We illustrate changes of the event graph after Step 2 and Step 3 in Figure 8.

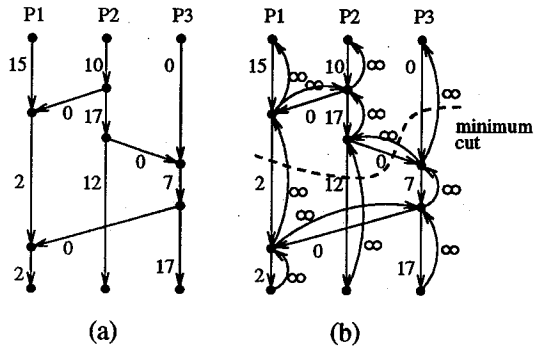

Figure 8: Example of deriving maximum consistency cut: (a) the graph after Step 2 ($M = 13$) and (b) the graph after Step 3.

4 Discussions

Traditionally, researchers in [3, 9] can only derive the minimum consistent cut cost with non-negative arc costs. In this paper, we propose a new and elegant technique, called normalization, by which we can easily 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 [9]. This is useful for a control system (e.g., a dynamic load balancing system) whose consistent cut costs should be bounded in a range.

The algorithms for the minimum and maximum consistent cut problems described in Subsection 3.3 basically consist of three operations: the normalization operation, the cost modification operation, and the maximum network flow operation. For the normalization operation, since each message arc cost are added into some internal arc costs, the time complexity is at most $O(m^2)$, where m is the number of arcs. Note that since this time complexity is not the dominant part, we will omit the discussion for optimization in this part. For the cost modification operation, since it only changes each arc cost, the time complexity is only $O(m)$. For the maximum network flow operation, we can use the fastest max-flow algorithm, the preflow-push algorithm [8] that runs in $O(nm \log(n^2/m))$, where n is the number of vertices. In our problem, since each vertex (corresponding to an event) in an event graph has only at most three incident arcs, two internal arcs and one message arc. Therefore, $m = O(n)$ in an event graph and the time complexity for the preflow-push algorithm is $O(n^2 \log n)$. Thus, the time complexity for detection of bounded global predicates is $O(n^2 \log n)$ in total.

References

[1] L. Bouge. Repeated snapshots in distributed systems with synchronous communication and their implementation in csp. *Theoretical Comput. Sci*, 49:145–169, 1987.

[2] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[3] C.M. Chase and V.K. Garg. Efficient detection of restricted classes of global predicates. In *The 9th International Workshop on Distributed Algorithms*, September 1995.

[4] R. Copper and K. Marzullo. Consistent detection of globla predicates. *Sigplan Notices*, pages 167–174, 1991.

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

[6] V.K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Tran. Parallen and Distributed Systems*, 5(3):299–307, March 1994.

[7] M.R. Garey, D.S. Johnson and L. Stockmeyer, Some Simplified NP-complete graph problems, *Theor. Comput. Sci.*, 1:237-267.

[8] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, October 1988.

[9] B. Groselj. Bounded and minimum global snapshots. *IEEE Parallel and Distributed Technology*, pages 72–83, November 1993.

[10] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communic. ACM*, 21(7):558–565, July 1978.

[11] N. Plouzeau M. Hurfin and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proc. of ACM/ONR Workshop in Parallel and Distributed Debugging*, pages 32–42, May 1993.

[12] C. Ghezzi M. Jazayeri and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, 1991.

[13] B.P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proc. of the 8th International Conference of Distributed Computing Systems*, pages 316–323, July 1988.

[14] C.E. Leiserson T.H. Cormen and R.L. Rivest. *Introduction to Algorithms.* The MIT press, 1989.

[15] I.C. Wu. *Multilist Scheduling: A New Parallel Programming Model.* PhD thesis, School of Computer Science, Carnegie Mellon University, July 1993.