Copyright
by
Trokon Edward Clinton
2017

The Report Committee for Trokon Edward Clinton
certifies that this is the approved version of the following report:

Using the Jump Number Problem to Efficiently Detect Global Predicates in Distributed Systems

APPROVED BY

SUPERVISING COMMITTEE:

Vijay K Garg, Supervisor

Jonathan W Valvano

Using the Jump Number Problem to Efficiently Detect Global Predicates in Distributed Systems

by

Trokon Edward Clinton, B.S.E.E.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2017

Acknowledgments

I would like to thank Dr. Vijay Garg for his expert guidance throughout the course of this report. Dr Garg's theoretical research in the areas of lattice theory and distributed computation formed the foundation of this report. Special thanks to Dr. Jonathan Valvano, the reader of this report, for valuable feedback and comments. I would also like to thank Himanshu Chauhan for helping me in the experimental setup of this project.

Using the Jump Number Problem to Efficiently Detect Global Predicates in Distributed Systems

Trokon Edward Clinton, M.S.E.
The University of Texas at Austin, 2017

Supervisor: Vijay K Garg

Detecting global predicates of a distributed computation is a key problem in testing and debugging distributed programs. It consists of searching the global state space of events to determine whether a given predicate could have occurred. For example a programmer may be interested in verifying whether a parallel program violates a global invariant, or detect a race condition between concurrent threads. This is a challenging problem because the number of consistent global states can grow exponentially when the number of events in the computation increases. This paper presents techniques that tackle the state explosion problem and help detect whether an arbitrary predicate is true in polynomial time. We first present a brute force algorithm, and then improve the performance with an exact and heuristic algorithm inspired by *the jump number* problem.

Table of Contents

Acknowledgments **iv**

Abstract **v**

List of Tables **vii**

List of Figures **viii**

Chapter 1. Introduction **1**

1.1 Model And Background 2

1.1.1 Breadth-First Traversal of Lattice of Consistent Cuts . . 7

1.1.2 Related Work 7

Chapter 2. Uniflow Chain Partition **8**

Chapter 3. The Jump Number Problem **12**

3.1 An Example 13

3.2 Mapping the jump number problem to the uniflow partition problem 14

3.3 Brute force algorithm 15

3.4 Exact algorithm 17

3.4.1 Algorithm description 20

3.4.2 Time and space analysis of exact algorithm 23

3.5 Heuristic Algorithm 23

Chapter 4. Experimental Evaluation **26**

Chapter 5. Conclusion **28**

Bibliography **29**

List of Tables

4.1 Experimental Evaluation of Various Test Cases. *OOM = Out Of Memory 27

List of Figures

1.1 Space-Time Diagram of a Distributed Computation 2

1.2 Distributed computation (a) and its corresponding DAG of consistent cuts (b) 6

2.1 Uniflow Partition of poset in figure 1.1(a) and figure 1.2(b) . . 9

3.1 Tasks under precedence constraints 14

3.2 Mapping the jump number problem (a) to the uniflow partition problem (b) 15

3.3 Partial state space graph of poset in figure 1.1 20

Chapter 1

Introduction

Parallel programming poses many challenges, they are strictly harder to implement compared to sequential programs and are often harder to debug and verify. Parallel programs involve issues that do not occur in the sequential world, such as a program deadlock, incorrect synchronization, or a race condition. Bugs introduced in a parallel program can be non-deterministic and appear rarely and unpredictably. Thus, the ability to efficiently evaluate a predicate over a distributed computation adds a great value to parallel programming because it can notify the programmer when the state of the system satisfies a particular condition.

To detect global predicates, the events of a distributed computation are recorded and the trace of events is modeled as a partial order. Then all consistent global states of the system are enumerated and visited to determine whether a global invariant could have been violated. This approach is widely used in distributed computing to detect deadlock, termination, token loss and data races [9].

The main contribution of this paper includes an efficient way of representing a distributed computation to minimize the time and space complexities

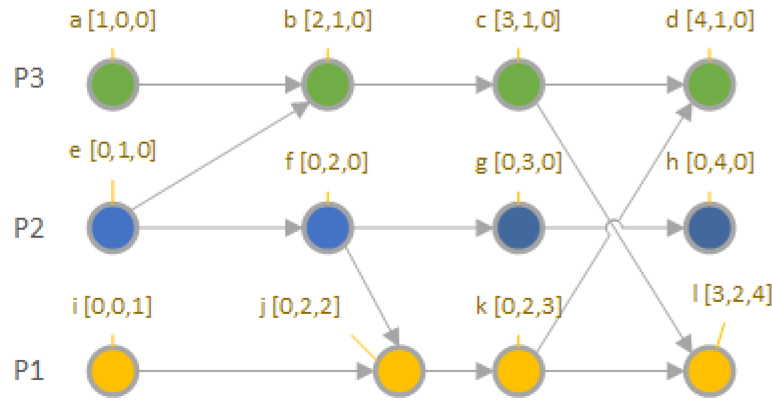

Figure 1.1: Space-Time Diagram of a Distributed Computation

of breadth-first global state enumeration algorithms.

1.1 Model And Background

A distributed system is a set of sequential *processes* $p_1, p_2, ..., p_n$ and a strongly connected communication network that implements unidirectional channels between pairs of processes [9]. Processes communicate via message exchanges through these reliable channels. A distributed computation is the execution of a distributed program by a set of processes. Each sequential process executes a sequence of events that are either local, or involves communication with other processes.

Distributed computations are often illustrated using a space-time diagram as shown in Figure 1.1.

The horizontal lines are *chains* and correspond to the various events on a single process. The arrows corresponds to messages that are sent between a

pair of processes. The base of the arrow represents the send event, while the head of the arrow is the receive event.

State space diagrams can also be viewed as a partially ordered set $(P, \rightarrow)$ (poset) consisting of events ordered by Lamport's *happened-before* $(\rightarrow)$ relation [8]. We use $e < f$ or $e \rightarrow f$ to indicate that e occurred before *f*. Two events e and f in P are concurrent if $e \not\rightarrow$ and $f \not\rightarrow$ e (denoted by e || f). For example, in Figure 1.1, $j < d$ while $c||g$.

In order to determine the causal relationship between events, we will use vector clocks proposed by Mattern [10] and Fidge [4]. For a distributed computation with n processes, a vector clock is an array of length n, with one clock per process. All events are timestamped with a vector clock (VC) such that for any event e on process P_j such that $i \neq j$, $VC(e)[i]$ denotes how many events from P_i causally precedes e. When $i = j$ it simply denotes how many events occurred on P_i.

Definition 1.1.1. Given two n-dimensional vectors V_1 and V_2 the *less than* relation (<) between them is defined as follows:

$$V_1 < V_2 \equiv (V_1 \neq V_2) \wedge (\forall k : 1 \leq k \leq n : V_1[k] \leq V_2[k]) \tag{1.1}$$

The following definition trivially follows:

Definition 1.1.2.

$$e \rightarrow f \equiv VC(e) < VC(f) \tag{1.2}$$

Figure 1.1 illustrates the vector clocks of all events in the given distributed computation.

A *cut* of a distributed computation is a subset $C \subseteq E$ where E is the set of events executed. A cut is consistent if the following definition holds.

Definition 1.1.3. (Consistent Cut) Given a computation $(E, \rightarrow)$, a cut C is consistent iff C contains an event e implies that C contains all events that *happened-before* e. In other words a cut C is *consistent* if for all events e and f:

$$(e \in C) \wedge (f \rightarrow e) \Rightarrow f \in C \tag{1.3}$$

A consistent cut defines the global state of the system at a particular time during its execution. Note that a cut may or may not be consistent. For example in Figure 1.1, the subset of events $\{a, b, e\}$ is a consistent cut. However $\{a, b, c\}$ is not because $e \rightarrow b$ but e is not in the subset. An inconsistent cut denotes an execution that could not have occurred, i.e. every *receive* event must have a corresponding *send* event.

If a distributed computation is partitioned into n chains, then any cut C can be represented using a vector clock of length n such that $C[i]$ denotes how many events from P_i were included in the Cut C . For example, in the execution illustrated in Figure 1.1, consider the state after P_1 executed i, P_2 executed e , P_3 executed a, b. This state is a consistent cut $\{i, e, a, b\}$ and is represented by the vector clock $[2, 1, 1]$ (events from P_i are at the ith index from the right). The cut $[2, 0, 1]$ is inconsistent since it includes event

b without including the event e which precedes it under the *happened-before* ($\rightarrow$) relation.

The consistent cuts of a distributed computation can be represented as a lattice or equivalently a directed acyclic graph (DAG) with one source and one sink such that the vertices represent consistent cuts and the edges represent transitions between consistent cuts. There is a directed edge from u to v if the state represented by v can be reached from u by executing a single event. The source of the DAG represents the state where no events have been executed while the sink is the state where all events have been executed. Global predicate detection consists of traversing the DAG of consistent cuts from source to sink and verifying whether a particular event of interest occurred. Many graph traversal algorithms such as Breadth First Search (BFS) or Depth First Search (DFS) can be used. Figure 1.2 illustrates a distributed computation along with its corresponding DAG. The execution has fourteen consistent cuts.

A useful definition that help characterize a cut C is the rank of a cut:

Definition 1.1.4. (Rank of a cut) Given a cut C, $rank(C)$ is the total number of events that have been executed to reach the cut. The rank of a cut can be computed from its vector clock representation as follows $rank(C) = \sum C[i]$. Informally, the rank of a cut can also be determined from the DAG of consistent cuts. It corresponds to the distance from the source vertex to the cut in the DAG. For example, in Figure 1.2, $rank([2, 2]) = 2+2 = 4$, this also corresponds to the distance from $[0, 0]$ to $[2, 2]$ in the DAG.

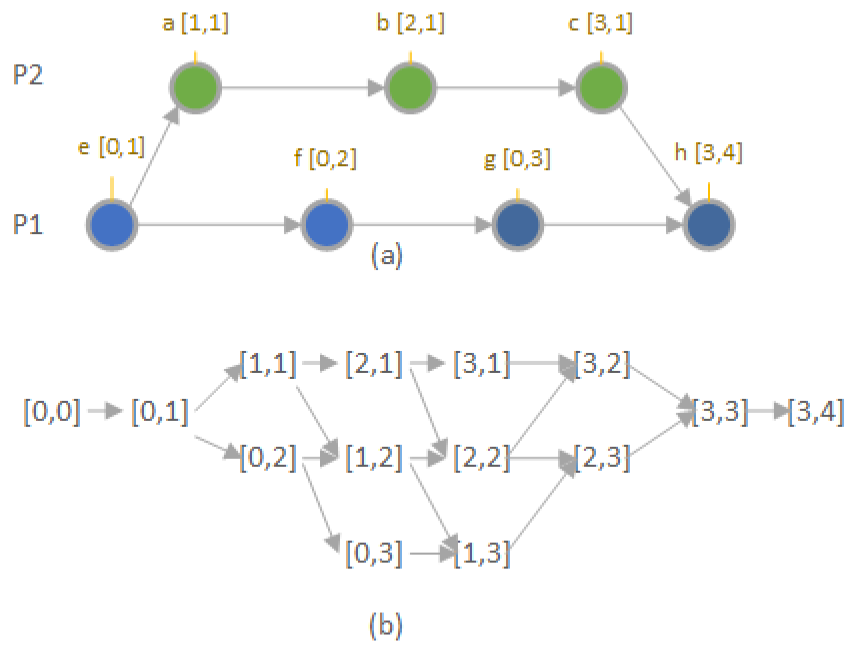

Figure 1.2: Distributed computation (a) and its corresponding DAG of consistent cuts (b)

1.1.1 Breadth-First Traversal of Lattice of Consistent Cuts

A Breadth-first search (BFS) of the lattice starts from the source node and visit nodes in one "layer" at a time. Thus we start with the source vertex and visit all the cuts with rank 1, then visit all the cuts with rank 2. The algorithm continues in this way until no new cuts are encountered. A BFS traversal of the lattice in figure 1.2.b will visit cuts in the following order $[0,0] \rightarrow [0,1] \rightarrow [1,1] \rightarrow [0,2] \rightarrow [2,1] \rightarrow [1,2] \rightarrow [0,3] \rightarrow [3,1] \rightarrow [2,2] \rightarrow [1,3] \rightarrow [3,2] \rightarrow [2,3] \rightarrow [3,3] \rightarrow [3,4]$.

1.1.2 Related Work

Cooper and Marzullo [3] gave the first algorithm based on breadth first search for global state enumeration. Let M represent the total number of consistent cuts of a poset P and n the number of processes. The algorithm proposed by Cooper and Marzullo requires $O(n^2M)$ time and exponential space in the size of the input computation. The main issue is that the Cooper-Marzullo algorithm may run out of memory if the number of events is large. It requires a space at least as large as the maximum number of cuts on a given level which may require exponential space.

Garg [6] presented a BFS traversal that achieves polynomial space $O(nE)$ at the expense of a worst time complexity of $O(En^2M)$ where E is the number of events in the computation and n is the number of processes. This paper aims to achieve improved polynomial space and time complexities without sacrificing performance.

Chapter 2

Uniflow Chain Partition

A uniflow chain partition is a special partition that allows efficient enumeration of consistent cuts. A chain partition is uniflow if it partitions the poset P into n chains $\{P_i \mid 1 \leq i \leq n\}$ such that no element in a higher numbered chain is smaller than any element in a lower numbered chain [5]. Formally, let $\alpha : P \rightarrow \mathbb{N}$ be a chain partition of P that maps every element of P to a chain number.

α is a uniflow chain partition if:

$$\forall x, y \in P : \alpha(x) < \alpha(y) \Rightarrow \neg(y \leq x). \tag{2.1}$$

Visually, this represents a state space diagram where all arrows are pointing upwards.

Lemma 2.0.1. *Every poset has at least one uniflow chain partition.*

Proof : If every element is in a separate chain by itself, then the resulting partition is uniflow.

The number of chains in the uniflow partition is at most equal to the number of events in P and any non-trivial partition will place at least two events in the same partition.

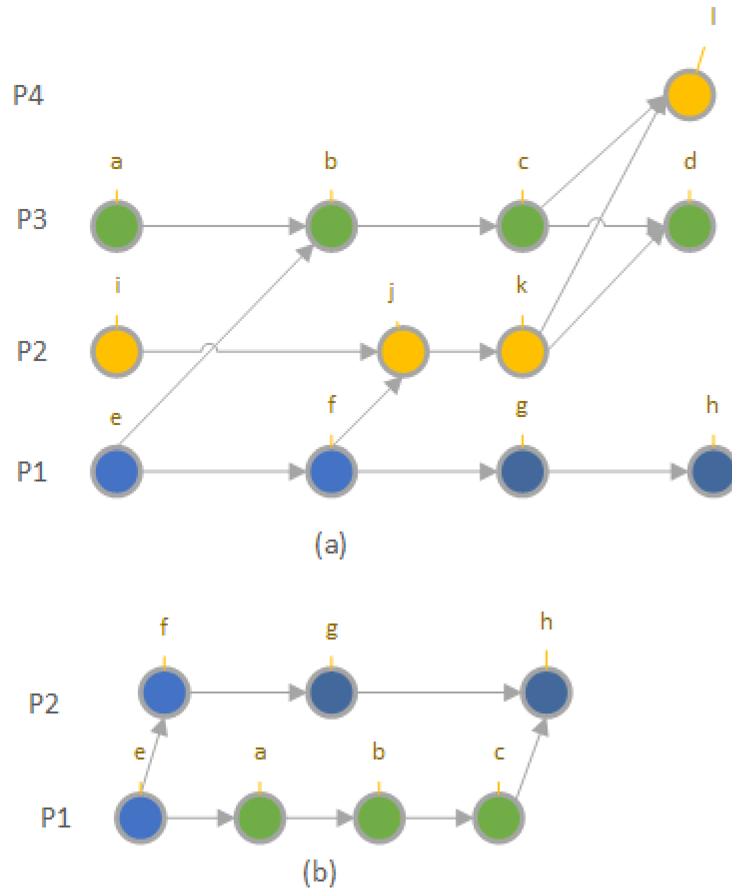

Figure 2.1: Uniflow Partition of poset in figure 1.1(a) and figure 1.2(b)

Figure 2.1 illustrates a uniflow partition of the posets in Figure 1.1 and 1.2. Both posets were partitioned into a minimum number of chains. Note that an artificial chain P_4 was added in figure 2.1.a to enforce the uniflow property. The number of consistent cuts remains the same for both chain partitions and there is a one to one mapping between cuts.

Lemma 2.0.2. *Uniflow Cuts [5]. Let P be a poset with a uniflow chain par-*

tition $\{P_i | 1 \le i \le n\}$*, and* G *be a consistent cut of* P*. Then any* $H_k \subseteq P$ *for* $1 \le k \le n$ *is also a consistent cut of* P *if it satisfies:*

$$\begin{aligned} \forall i : k < i \le n : H_k[i] = G[i], and \\ \forall i : 1 \le i \le k : H_k[i] = |P_i|. \end{aligned} \tag{2.2}$$

Proof: Given that G *is a consistent cut, this Lemma specifies that if the events in the higher chains of* G *are included in* H_k*, then* H_k *is consistent as long as all the events in the lower chains are also included. This is true since all the events in lower chains either precede or are concurrent with events in higher chains. By including them, this guarantees that all events that causally precede the events from* G *are included in the cut. This leads to a consistent cut.*

Garg and Chauhan [7] showed that the lattice of consistent cuts of a uniflow poset can be traversed in polynomial time while using polynomial space. Let M represent the number of consistent cuts, n_c is the number of chains, n is the number of processes and E is the set of events in a distributed computation. Garg's and Chauhan's breadth-first traversal algorithm takes $O(({n_c}^2 + n^2)M)$ time and $O((n_c + n)|E|)$ space which is a considerable improvement compared to known BFS enumeration algorithms. It's main limitation is that it requires a poset with a uniflow chain partition. This means that a poset needs to be first partitioned into uniflow chains before the lattice of consistent cuts is traversed. In addition, since the time and space complexities depend directly on the number of chains in the computation, it is beneficial to find a uniflow partition with a minimum number of chains n_c.

As we have shown in Lemma 2.0.1, every poset has at least one uniflow chain partition. We seek a polynomial time algorithm to find an optimal uniflow chain partition of a poset (one that uses the smallest number of chains). This is a challenging open problem that will directly impact the runtime performance of global state enumeration of uniflow posets.

Chapter 3

The Jump Number Problem

Finding the minimum number of chains in a uniflow partition can be mapped to *the jump number problem*. The jump number problem seeks to find a total order of a given partially ordered set that minimizes the number of jumps, i.e., the total number of consecutive pairs of elements that are not comparable.

Definition 3.0.1. Total Order.

A total order of a poset P, also known as a linear extension, is a permutation of the elements $e_1, e_2....$ of P such that $e_i < e_j$ or $e_i \rightarrow e_j$ implies that $i < j$. In other words, a total order of P is a permutation of the events that preserves the partial order P.

The jump number problem is often stated as a scheduling problem: a single core machine runs a set of jobs one at a time; several precedence constraints prevent the start of certain jobs until others complete. A job which executed immediately after a job which is not constrained to precede it requires a "jump" and has some constant additional cost. The jump number problem is to construct a schedule to minimize the number of jumps.

3.1 An Example

Suppose there are six tasks a, b, c, d, e, f that need to execute on a single core machine. Some tasks have precedence constraints as follows:

- Task a is the first task
- Task f cannot be scheduled before tasks b, e, a complete
- Task d cannot be scheduled before tasks a, e complete
- Task b cannot be scheduled before tasks a, e complete

The tasks and precedence constraints are also illustrated by the poset in figure 3.1.

There are several schedules (total orders) that comply with these constraints such as: $a \rightarrow e \rightarrow d \rightarrow c \rightarrow b \rightarrow f$ or $a \rightarrow c \rightarrow e \rightarrow d \rightarrow b \rightarrow f$. We aim to find a schedule that minimizes the total delay between tasks. Consecutive pairs of tasks that are not linked by a precedence constraint will have a delay equal to 1, while consecutive tasks that are linked by a constraint have no cost. For example if task c was followed by the execution of task e then this would be considered a jump since these two tasks are unrelated. *Example* : The sequence $a \rightarrow e \rightarrow d \rightarrow c \rightarrow b \rightarrow f$ has 2 jumps, since the pairs (d, c) and (c, b) are unrelated.

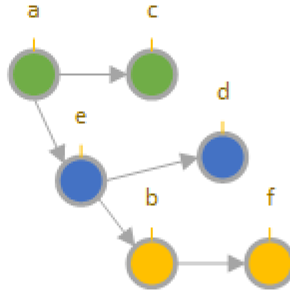

Figure 3.1: Tasks under precedence constraints

3.2 Mapping the jump number problem to the uniflow partition problem

Recall from Definition 3.0.1 that a total order T of a poset is an ordering of events such that if $e_i \to e_j$ in the poset, then e_i precedes e_j in the total ordering. Recall also that, in a uniflow chain partition of a poset, all events in lower numbered chains either precede or are concurrent with events in higher numbered chains. In addition, the jumps in T split T into chains P_i such that $T = P_1 \oplus P_2 \oplus ....P_n$. Therefore, in order to partition a poset to a mimimum number of uniflow chains, we first find the total ordering T with the smallest jump number n, then split T n times at the jump points. This will form a uniflow poset with exactly $n - 1$ chains. Note that minimizing the number of jumps will also minimize the number of chains. The procedure is illustrated in figure 3.2: (a) is an ordering of events in figure 1.1 with the minimum number of jumps, (b) uses (a) to form a uniflow chain partition with the minimum number of chains.

The following sections present three algorithms to solve the jump num-

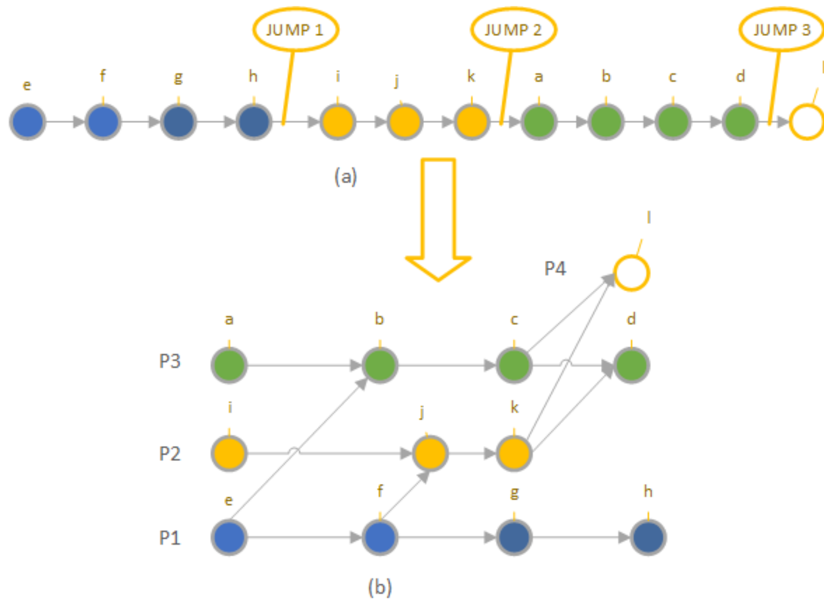

Figure 3.2: Mapping the jump number problem (a) to the uniflow partition problem (b)

ber problem: a brute force algorithm, an exact algorithm and a heuristic algorithm that were all implemented using the Java programming language.

3.3 Brute force algorithm

We first present a brute force algorithm that solves the jump number problem. The trivial algorithm consists of first finding all the possible total orders of a poset and returning the one with the minimum number of jumps as shown in Algorithm 1.

The first step in algorithm 1 is to compute the total orders of a poset. This is achieved using an implementation presented by Pruesse and Ruskey [11]

Algorithm 1 Brute force jump number computation

```
Input: Poset P
Output: Linear extension of P with the min number of jumps.
T ← TotalOrders(P) // T is an array of all total orders.
jumpmin ← |P|
bestorder ← ∅
while T is not empty do
    e ← T.pop()
    jumpcount ← 0
    for i = 0 to i = |e| − 2 do
        breakloop ← false
        if ¬ has_edge(e[i], e[i + 1]) then //unrelated
            jumpcount++
            if jumpcount ≥ jumpmin then
                breakloop ← true
                break
            end if
        end if
    end for
    if breakloop == false then
        jumpmin ← jumpcount
        bestorder ← e
    end if
end while
    return bestorder
```

in which all linear extensions of a poset are computed in constant amortized time $O(|E(P)|)$ where P is a poset, $E(P)$ is its set of linear extensions. This is the fastest known algorithm for generating the linear extensions of a poset and corresponds to the $TotalOrders$ procedure in algorithm 1. Pruess and Ruskey's algorithm operates on a poset P represented as a graph and was adapted to work on posets whose events are represented by vector clocks.

Algorithm 1 has a worst case time complexity of $O(n! + n! * n)$. Proof: consider n events with no precedence constraints between them. There are thus $n!$ possible permutations of events. This means that the procedure $TotalOrders$ completes in $O(n!)$ time in the worst case. The *while* loop in algorithm 1 loops over all $n!$ total orders and contains a nested loop which iterates over all n events. Therefore the *while* loop completes after $O(n! * n)$. This shows that algorithm 1 has a time complexity of $O(n! + n! * n)$ in the worst case.

3.4 Exact algorithm

The jump number problem is known to be NP-hard [2]. Bianco et al. proposed the first exact algorithm for solving the jump number problem for general posets [1]. The exact algorithm in this report is based on their paper and implemented using the Java programming language. It was also adapted to compute on a vector clock representation of events and transformed to a non-recursive implementation.

We first present several definitions that will help explain the imple-

mentation of the exact algorithm to compute the jump number of an arbitrary poset.

We have seen that $u < v$ means that u precedes v while $u \parallel v$ means that u and v are incomparable. Similarly, an element v of a poset $(X, <)$ covers another element u provided that $u < v$ and there exists no third element y in the poset for which $u \leq y \leq v$. v covers u is expressed as $u \prec v$. Recall that a *chain* is a set of pairwise comparable elements of the poset P.

Definition 3.4.1. Greedy Starting Chain [1]. Let $V = \{v_1, v_2, v_3 ...., v_n\}$ be a finite set of elements. We denote $Pred(v) = \{u \in V | u < v\}$, where $v \in V$. $Pred(v)$ is essentially the set of elements that precede v in the poset P. If $Pred(v)$ is a chain C of P and for no $x \in P$ that covers v the set $Pred(x)$ is a chain, then C is a *greedy starting chain.* A greedy starting chain thus describes a set of pairwise comparable elements, a chain, that cannot be extended any further by adding another element to the chain. Adding a new element e to the greedy starting chain will cause $Prev(e)$ to not be a chain. Given a graphical representation of a poset, a greedy starting chain (GSC) can be computed by traversing the graph starting from a minimal element of the poset and adding all the reachable elements e_i to the GSC until $pred(e_i)$ is no longer a chain (i.e. the node e_i has an indegree greater than 1).

A total order T of P is greedy if for some n, $T = C_1 \oplus C_2 \oplus ... \oplus C_n$, where each C_i is a greedy starting chain of the poset $P_i = P|(V - \cup_{j<i} C_j)$. Note that the problem of finding a greedy total order T of P can be broken

down into several related subproblems. If C_1 is a greedy starting chain of the poset P, then C_2 is a greedy starting chain of $P - C_1$, C_3 is a greedy starting chain of $P - C_1 - C2$ and so forth.

The jump number problem can be solved by finding a greedy total order with the minimum number of greedy starting chains. Let the state (X, C_i) represent the poset X and its greedy starting chain C_i. Let $f(X, C_i)$ represent the minimum jump number computed over all total orders of X with C_i as the starting subset. Let G be a state space graph whose vertices consist of the states (X, C_i) and whose edges is the set $\{(X, C_i), (X\prime, C_j)\}$ and represents valid transitions from state (X, C_i) to state $(X\prime, C_j)$ such that $X\prime = X - C_i$. Each edge has a cost of 1 and represents a jump. The minimum jump number of the subposet X that has a starting chain C_i can thus be computed using the following dynamic programming recursion [1]:

$$\begin{gathered} f(X, C_i) = min(f(X\prime, C_j) + 1) \\ such\ that\ X\prime = X - C_i, f(C, C) = 0 \end{gathered} \tag{3.1}$$

Figure 3.3 shows a partial state space graph of all possible states in figure 1.1.

The minimal jump number of a poset P corresponds to the shortest path from the source of the DAG of states to any sink. The path highlighted in blue in figure 3.3 is one of many optimal solutions (minimum jump number) and also corresponds to the solution found in figure 2.1.a .

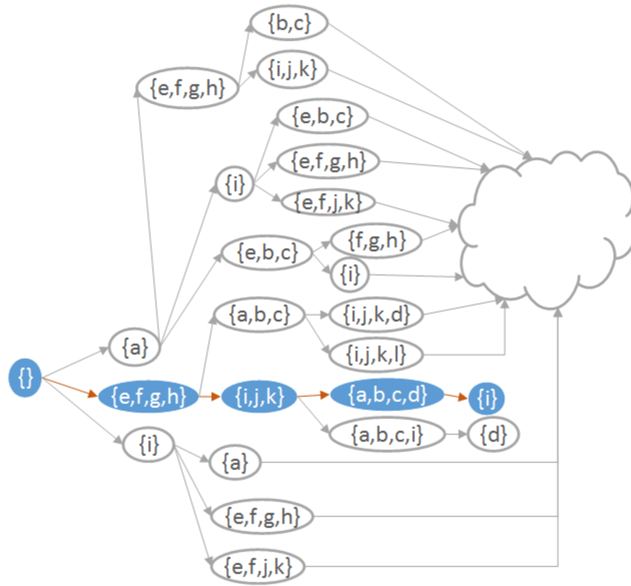

Figure 3.3: Partial state space graph of poset in figure 1.1

3.4.1 Algorithm description

The algorithm operates in three main steps:

1. It first generates all the states in Breadth-First order which means that all states at level n are computed before the states at level $n+1$

2. Each state is visited in reverse order of the BFS enumeration in step 1, starting from the sink nodes. The state with minimal cost at each level is recorded. The algorithm uses the state costs of higher levels to identify the minimum cost of states at lower levels i.e., $mincost(n) = mincost(n+1) + 1$. Note that a sink node has a cost of 0 and the min cost of a non-sink node i is the min distance from i to a sink. At the

end of this step, the cost of the root node will be the shortest path from the root node to any sink node.

3. The state space graph is re-traversed and follows the shortest path identified in step 2. This path is the solution to the jump number problem.

Algorithm 2 illustrates the various steps in pseudocode form.

Line (3) of algorithm 2 first computes all the greedy starting chains of the poset P. The variable cut refers to the starting event of each chain and is represented in vector clock format. Thus, the pair (P, cut) always represents a subposet of P. The procedure *AllGSC* returns all the greedy starting chains of the subposet (P, cut) and the set of events covered along the path to each GSC.

In line (6-10), all the greedy starting chains of P and subposets $P_i = P|(V - \cup_{j<i} C_j)$ where each C_j is a greedy starting chain are enumerated and stored in *all_gcs*.

Lines (11 - 36) corresponds to step 2 above, while lines (37 - 44) corresponds to step 3.

The procedure *AllGSC* illustrated in algorithm 3 computes all the greedy starting chains of a subposet (P, cut). A chain i of the poset corresponds to events executed by a process i and the events are represented in vector clock format. Line (7) of algorithm 3 iterates over all the chains of the distributed computation. The loop in line (9) iterates over all the events e_i

Algorithm 2 Exact algorithm for jump number computation

```
1: input : poset P; output: min jump number and corresponding gscs.
2: cut ← [0, 0, ...0]
3: head ← AllGSC(P, cut)
4: all_gscs ← ∅
5: all_gscs ← all_gscs ∪ head
6: for i = 0 to i = all_gscs.size − 1 do
7:     c ← all_gscs.get(i)
8:     c.nxt ← AllGSC(P, c.cut)
9:     all_gscs ← all_gscs ∪ c.nxt
10: end for
11: for element in reversed(all_gscs) do
12:     if element.nxt.size == 0 then //sink
13:         element.mincost ← 1
14:     else
15:         mincost ← ∞
16:         best_child_index ← 0
17:         for i = 0 to i = element.nxt.size − 1 do
18:             gsc ← element.nxt.get(i)
19:             if gsc.mincost < mincost then
20:                 mincost ← gsc.mincost
21:                 best_child_index ← i
22:             end if
23:         end for
24:         element.mincost ← mincost + 1
25:         element.best_child_index ← best_child_index
26:     end if
27: end for
28: mincost ← ∞
29: best_child_index ← 0
30: for i = 0 to i = head.size − 1 do//find the best GSC of P
31:     gsc ← head.get(i)
32:     if gsc.mincost < mincost then
33:         mincost = gsc.mincost
34:         best_child_index = i
35:     end if
36: end for
37: ans ← ∅
38: curr = head.get(best_child_index)
39: ans ← ans ∪ curr.gscs
40: for i = 1 to i = mincost − 1 do
41:     min_index ← curr.best_child_index
42:     curr ← curr.next.get(min_index)
43:     ans ← ans ∪ curr.gscs
44: end for
    return mincost − 1, ans
```

of a single process *pid* and lines (10 - 35) determine if $prev(e_i)$ is a chain C. Events are added to the GSC as long as $prev(e_i)$ is a chain, and the algorithm moves to the next pid as soon as $prev(e_i)$ is no longer a chain.

3.4.2 Time and space analysis of exact algorithm

Let n represents the number of events in a poset P and let w be the minimum number of jumps. The exact algorithm for jump number computation generates $O(n^w)$ states in the worst case since each state in the state space graph G has at most n outgoing arcs; and the depth of G is $w \leq n$ based on lemma 2.0.1. Let n_p represent the number of processes in a distributed computation. Then *AllGSC*() runs in $O(n_p * n^3)$ time because it contains three nested loops that iterate over all events for each process . Since this procedure is called for all the generated states, it follows that the exact algorithm runs in $O(n_p * n^{w+3})$ time and can take $O(n_p * n^{n+3})$ in the worst case.

3.5 Heuristic Algorithm

We have seen that the exact algorithm to compute the minimum jump number is not practical since it has an exponential time complexity and space usage in the worst case. We thus seek a fast heuristic algorithm that finds a solution to the jump number problem that is close to the optimal one in polynomial time.

The proposed heuristic algorithm is based on a greedy algorithm in which only the maximal greedy starting chain on a given level in the state space

Algorithm 3 procedure AllGSC()

```
 1: input: Poset P
 2: input: cut specifying the first events in the Poset
 3: output: greedy starting chains
 4: start ← cut
 5: vc_curr ← cut
 6: ans ← ∅
 7: for pid = 0 to pid = P.chain_count − 1 do
 8:     sol ← ∅
 9:     while vc_curr.get(pid) < P.sizeOfChain(pid) do
10:         candidate ← vc_curr.get(pid) + 1
11:         candidate_vc ← vector clock of candidate
12:         invalid ← false
13:         new_elements ← []
14:         new_incs ← []
15:         //find prev(candidate)
16:         for i = 0 to vc_curr.size − 1 do
17:             if candidate_vc.get(i) − vc_curr.get(i) > 0 then
18:                 //new events available
19:                 new_incs.add(candidate_vc.get(i) − vc_curr.get(i))
20:                 for j = vc_curr.get(i) + 1 to candidate_vc.get(i)  do
21:                     new_elements.add(event_ij)
22:                 end for
23:             else
24:                 new_incs.add(0)
25:             end if
26:         end for
27:         for i = 0 to new_elements.size() − 2 do
28:             a ← vector clock of new_element.get(i)
29:             b ← vector clock of new_element.get(i + 1)
30:             if a.isConcurrentWith(b) then
31:                 //not a chain
32:                 invalid ← true
33:                 break
34:             end if
35:         end for
36:         if invalid then
37:             break // go to next pid
38:         else
39:             for i = 0 to i = vc_curr.size() do
40:                 vc_curr[i] ← vc_curr[i] + new_incs[i]
41:             end for
42:             sol ← sol ∪ new_elements
43:         end if
44:     end while
45:     ans ← ans ∪ {sol, vc_curr}
46:     vc_curr ← start //reset vc_curr
47: end for
        return ans
```

graph of a poset is considered and all others are discarded. In other words, only the maximal GSC returned by *AllGSC* is used for further computation. This limits the number of generated states to $O(n)$ since a poset with n events can have at most n chains (Lemma 2.0.1). Algorithm 4 illustrates the proposed heuristic algorithm which runs in $O(n_p * n^4)$ since there are at most n states and *AllGSC*() executes in $O(n_p * n^3)$ time. We have thus achieved a polynomial time complexity which is a big improvement over the exact algorithm.

Algorithm 4 Heuristic algorithm for jump number computation

1: **input** : *poset P*
2: **output**: approximate jump number and corresponding gscs
3: $cut \leftarrow [0, 0, ...0]$
4: $head \leftarrow AllGSC(P, cut)$
5: $all_gscs \leftarrow \emptyset$
6: $all_gscs \leftarrow all_gscs \cup \mathbf{max}(head)$
7: **for** $i = 0$ *to* $i = all_gscs.size - 1$ **do**
8: $\quad c \leftarrow all_gscs.get(i)$
9: $\quad c.nxt \leftarrow AllGSC(P, c.cut)$
10: $\quad all_gscs \leftarrow all_gscs \cup \mathbf{max}(c.nxt)$
11: **end for**
$\quad$ **return** $all_gscs.size() - 1, all_gscs$

Chapter 4

Experimental Evaluation

All experiments were conducted on a Linux machine with an Intel Xeon 2.80GHz CPU. I used Java 8 and limited the java heap size to 2GB. Random posets describing various distributed computations ranging in size from 5 events to 105,282 events were evaluated. The table below illustrates the execution times for the brute force, exact and heuristic algorithms for computing uniflow partition chains presented in this paper. n is the number of processes in the poset, $|E|$ is the number of events while n_u determines how many uniflow chains the poset was partitioned into.

The exact and brute force algorithms consistently exceeded the java heap space limit of 2GB when more than 50 events were used. The heuristic algorithm is the best performing one in terms of both space usage and execution time.

test case	$\|E\|$	n	heuristic n_u	heuristic time	exact n_u	exact time	brute n_u	brute time
d-5.txt	5	2	2	3ms	2	6ms	2	1ms
d-10.txt	10	10	9	4ms	9	47s	9	675ms
d-50.txt	50	10	15	8ms	OOM	OOM	OOM	OOM
d-100.txt	100	10	24	10ms	OOM	OOM	OOM	OOM
bank-8.txt	96	8	8	11ms	OOM	OOM	OOM	OOM
copy-8.txt	92	8	8	8ms	OOM	OOM	OOM	OOM
w-480-4.txt	480	4	4	40ms	OOM	OOM	OOM	OOM
hedc-12.txt	216	12	19	14ms	OOM	OOM	OOM	OOM
d-300.txt	300	10	54	18ms	OOM	OOM	OOM	OOM
w-480-8.txt	480	8	8	22ms	OOM	OOM	OOM	OOM
d-500.txt	500	10	76	25ms	OOM	OOM	OOM	OOM
w-480-12.txt	480	12	12	21ms	OOM	OOM	OOM	OOM
w-480-16.txt	480	16	16	23ms	OOM	OOM	OOM	OOM
double-8.txt	1292	8	37	42ms	OOM	OOM	OOM	OOM
d-10k.txt	10k	10	1504	133ms	OOM	OOM	OOM	OOM
elevator-12.txt	38528	12	2908	3ms	OOM	OOM	OOM	OOM
tsp-8-file16large.txt	105282	8	330	18.4s	OOM	OOM	OOM	OOM

Table 4.1: Experimental Evaluation of Various Test Cases. *OOM = Out Of Memory

Chapter 5

Conclusion

This report presents several algorithms to partition a distributed computation into uniflow chains by using the jump number problem. This special chain partition allows efficient breadth-first-search based traversal of global states of parallel programs. Once a poset has been partitioned into uniflow chains, it can be used to detect predicates in concurrent programs while consuming polynomial space and time.

Our results show that although the jump number problem is NP-hard, we are able to find a close to optimal jump number solution in polynomial time using a heuristic algorithm.

Bibliography

[1] L. Bianco, P. Dell'Olmo, and S. Giordani. An optimal algorithm to find the jump number of partially ordered sets. *Computational Optimization and Applications*, 8(2):197–210, 1997.

[2] Vincent Bouchitte and Michel Habib. Np-completeness properties about linear extensions. *Order*, 4(2):143–154, 1987.

[3] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, PADD '91, pages 167–174, New York, NY, USA, 1991. ACM.

[4] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):5666, 1988.

[5] Vijay K. Garg. *Introduction to lattice theory with computer science applications*. Wiley.

[6] Vijay K. Garg. Enumerating global states of a distributed computation. In *IN INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING SYSTEMS*, pages 134–139, 2003.

[7] Vijay K. Garg and Himanshu Chauhan. Space efficient breadth-first traversal of consistent global states of parallel programs.

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

[9] Keith Marzullo and Özalp Babaoglu. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Technical Report UBLCS-93-1, Piazza di Porta S. Donato, 540127 Bologna, Italy, January 1993.

[10] Friedemann Mattern. Virtual time and global states of distributed systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*, pages 215–226. North-Holland, 1988.

[11] Gara Pruesse and Frank Ruskey. Generating linear extensions fast. *SIAM J. Comput.*, 23(2):373–386, 1994.