IMPLEMENTATION OF ALGORITHMS FOR COMPUTING INFORMATION PROPAGATION DELAYS THROUGH SEQUENCES OF FIXED-PRIORITY PERIODIC TASKS

by

VINEET TADAKAMALLA

B.TECH, J.N.T.U, INDIA, 2009

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2011

Approved by:

Major Professor
Dr. Rodney Howell

Abstract

Nowadays, there is a rapid increase in the complexity of large automotive and control systems because of the integration of external software modules in them. Many of these systems are based on sampled-data control theory. And because of the different timing constraints of individual modules, each module has a different sampling rate. Typically, these systems operate with periodic task sequences and the information flows between the tasks. Generally the information propagates from tasks operating at one period to tasks operating at different periods. When this happens, unusually long information propagation delays can occur which can be disastrous because the system cannot respond to the changes until this delay has been elapsed. Although for arbitrary set of task sets the delays can be very long, *Howell and Mizuno (2010)* considers a set of task sequences with special constraints and some very useful bounds are derived for the worst case occurrences of them. *Howell and Mizuno (2011)* has laid out algorithms that compute the delays for certain special cases of task sequences considered in *Howell and Mizuno (2010).* The purpose of this project is to understand and implement the algorithms from *Howell and Mizuno (2011)*. The implementation is done so that it avoids the manual computation of the delays and helps in better understanding the ideas presented in *Howell and Mizuno (2011)*. The application can be tested against any valid input that meets our assumptions, and it constructs a schedule that exhibits the worst case behavior and from the schedule it computes the worst case information propagation delays.

Table of Contents

List of Figures

Acknowledgements

I would like to thank my major professor Dr. Rodney Howell for his constant guidance and help throughout the project. I would also like to thank Dr. Torben Amtoft and Dr. Simon Ou for graciously accepting to be on my committee. Finally, I wish to thank my family and friends for all their support and encouragement.

1. Introduction

In any large scale complex automotive or control systems, there are many modules that communicate with each other. So information flows from one module to another module. Most of these systems are based on sampled-data control theory [6]. And in such systems periodic sampling occurs. And because different modules have different timing constraints the sampling rates of these individual modules differ.

Since different sampling rates are used, the modules are driven at different rates using a multi-rate driver to be cost effective. Here typically information flows from modules driven at one rate to modules driven at a different rate, or in other words information propagates from tasks operating at one period to tasks operating at different periods. So when this happens, the reaction times can increase drastically because of the end to end delays. These delays can be a worry because the system cannot respond to any changes in the input until this delay has elapsed. This problem has been addressed in [2] and more work has been done in [1] and subsequently in [3].

Typically these systems operate with periodic task sequences and use rate monotonic scheduling (defined in section 2). When tasks operating at larger periods reads information from tasks operating at a smaller periods, unexpectedly long end-end delays can occur. Because the shorter period task can execute many of its instances before the larger period task finishes its instance execution, the information based on which the larger period task is operating might be very old. So the output to the latest change in the input is not reflected in the system until very late. These delays are known as end-to-end delays.

Although the end-to-end delays are a known phenomenon and some known research has been done in this area, what is more interesting is analyzing the worst case occurrences of these delays. This has been addressed in [1] and some algorithms for computing the delays have been proposed in [3].

In this report we consider two types of information propagation delays, namely, *First-First-delay* and *Last-Last-delay,* whose definitions are given in the section 2 of this report.

These delays vary depending on the scheduling algorithms used and also the priorities of the tasks and other constraints. The upper and lower bounds for worst case delays for certain kinds of task sequences with special constraints are derived in [1].

This project is an implementation that constructs a schedule that produces the worst case information propagation delays for any task sequence which is in accordance with the assumptions and special constraints given in [3].

1.1. Goal

The main goal of this project is to understand and implement the algorithms mentioned in section 4 of this report, which were originally written in [3]. Also the results of the implementation should be matched up against the upper bounds mentioned in section IV of [1]. It is our aim that these results give us a good idea about how long worst case propagation delays can be in a system. It should also show that the bounds are tight by producing many schedules where the delays approach these bounds.

Since in the course of the implementation there is always the issue of efficiency, the secondary goal is to implement the algorithms in the most efficient way possible so that its run time analysis is in accordance with the ones predicted in the paper.

One more goal is to prepare a good number of test cases that cover many possible scenarios and perform testing on them to assure that the implementation meets all the requirements and produces the results according to our expectations.

The final goal is to provide an easy user interface so that any naive user without much knowledge about the technology can use it to provide their own set of test cases and match the results with the expected ones.

1.2. Motivation

Although [3] has laid out the algorithms and also the correctness proofs in a pretty comprehensive manner, in order to better understand the ideas presented, an implementation of it is required.

This implementation works as a tool for showcasing different examples and avoids the tedious task of computing the delays manually for each test case and explaining the results of it on paper. So in this way it avoids a lot of time and effort required from the end user side.

The implementation also allows the users to test it with various sets of their own test cases, rather than depending on the ones provided by us. They can use the results for better understanding the delays and also if possible, they can figure out a way to reduce them.

The implementation can work as a prototype for extending it to work as an analysis tool for computing worst case propagation delays in different kinds of task sequences within a given task set and different priority assignments apart from fixed priority scheme.

1.3. Challenges

The main challenge was learning the theory behind these systems and understanding the algorithms and ideas presented in [1], [3] and to some extent [2] completely before starting the implementation.

The problem is not only to implement the algorithms exactly as described in the paper but also to implement them efficiently so that it takes as little time as possible to compute the worst case propagation delays.

The other challenge when developing the solution was to decide what data structures are useful for implementing the algorithms. Since there was considerable freedom in the design of things, the other challenge was how to reuse components and functions inside the program so that we do not generate the same things more than once. The details on how this is implemented are mentioned in section 5 of this report.

1.4. Software and Hardware Requirements

Since the nature of the algorithm uses data structures like priority queues, lists, arrays etc, The Java language is a well suited choice because of its inbuilt and efficient implementation of data structures. Also my familiarity with the Java language helped me choose it to develop this application.

So the major software requirements for developing this application would be

1. Eclipse Galileo,
2. Java.

There are no specific hardware requirements for this application since we are developing this as a simple application and would run on small test cases, but for better user experience and minimum response time we need to have a faster RAM and a higher CPU speed for the computer or server where we would be hosting our application. Also since the technology used is Java and is platform independent, the application can run on any operating system.

1.5. Overview

Since now we have given an introduction and specified the goals, motivation and challenges behind this project, the rest of the report has the definitions, assumptions and constraints necessary to understand the implementation in section 2, and mentions the algorithms and theorems from [3] that have been implemented in section 3. Section 4 describes how the design and implementation has been done. It also has some figures to better understand the classes and methods in the code. Section 5 talks about the various test cases considered and analyses and interprets the results of executing them. Section 6 has a performance analysis which gives an idea about the time complexity or the run time analysis of the implementation. Section 7 concludes this report by mentioning the scope of this implementation and also the future work that can be done.

2. Definitions and Assumptions

In our implementation, a *Task* is a structure that has the primary properties Task name, Period, Maximum Execution Time and Priority. This task becomes available for execution at time kp, where p is the period of the task for each natural number k, and each of these instances must be scheduled to completion before the next instance is available. Each instance is guaranteed to require no more than its Maximum Execution Time.

A *Task Set* is a set of *Tasks* that are to be scheduled. Here each Task is as above. A *Task Sequence* is a sequence of *Tasks* from a Task set where information flows from one task in the sequence to the next in that order. The Information Flows and Information Propagation Delays (defined later in this section) have to be computed for this Task sequence.

Scheduling is the process of deciding and executing the task instances in a deterministic way according to their priorities. Highest priority tasks are executed first. *Fixed Priority Scheduling* is a scheme of scheduling where every *task* in the task set is assigned a fixed priority at the time of initialization. So the priorities remain unchanged throughout the execution. In our implementation we consider task sets that follow only Fixed Priority Scheduling.

Preemption occurs when a higher priority task becomes available for execution and a lower priority task is executing. In this scenario, the lower priority task stops execution and higher priority task begins executing. The lower priority task resumes its execution later. In our implementation preemption is allowed.

A *Schedule* encapsulates the entire information about the execution times of the task instances. It is a timeline which shows what individual task instance is executing at a particular time unit. In our implementation a *Schedule Prefix* is a structure that, for each instance of every task in the task set has the task name, start time and end time of its execution. We execute the task instances based on the priorities and record the task name, start and end times of the task instance. The Schedule Prefix is then the list of all the information for each task instance within a certain time period.

Rate Monotonic Scheduling, defined in [5] is a method of scheduling where priorities are assigned based on the periods. Shortest period tasks are given the highest priority and vice versa.

If each task *period* is an integer multiple of every other task period then the periods are said to be *Harmonic periods*. In our implementation we consider task sequences that have harmonic periods.

A task schedule is said to be a *Feasible Schedule*, if every instance of each task in the sequence gets executed before the next instance of the same task is ready for execution. In our implementation we assume that every task sequence has a feasible schedule.

Worst-case Response Time, R(T) is the maximum difference between the time at which an instance of task T becomes available and the completion time of that instance.

The rest of the definitions in this section were defined in [3] and are quoted below.

> Let $T = <T_1, . . . , T_n>$ be a task sequence within some feasible task set. Given a schedule of the task set and a time t_0, the *First-First Information Flow* from t_0 through T in this schedule is the sequence of times $t_0 < \ldots\ldots < t_n$ such that for $1 \leq i \leq n$, t_i is the finish time of the first instance of T_i that begins executing no earlier than $t_i−1$ [3].

From the above definition, The *First-First Information Propagation Delay ($dFF_T\,(t_o)$))* of T from time t_0 within the given schedule is the length $t_n - t_0$ of the first-to-first information flow from t_0 [3].

Worst case First-First Information Propagation Delay (DFF_T) is the maximum duration from a moment at which input data changes to the earliest moment at which the change and possibly subsequent changes due to periodic sampling of input data is reflected in the output.

From the above two definitions, it is the maximum of *First-First Information Propagation Delays* taken over all schedules produced using the given task priorities and all times t_0 plus 1 [3].

This is because in a control system external events do not necessarily occur at integer time values. So the actual delay might be nearly 1 more than the delay we have defined. Hence 1 is added to compensate for it.

> Let T = $<T_1, \ldots, T_n>$ be a task sequence within some feasible task set. Given a schedule of the task set and a time t_{n+1}, the *Last-Last Information Flow ($dLL_T\,(t_{n+1})$))* to t_{n+1} through T in this schedule is the sequence of times $t_1 < \ldots\ldots.. < t_{n+1}$ such that for $1 \le i \le n$, t_i is the starting time of the last instance of T_i that finishes executing no later than t_{i+1}. (Note that for small values of t_{n+1}, there may be no Last-Last Information Flow to t_{n+1}.) [3].
>
> The *Last-Last Information Propagation Delay* of T to time t_{n+1} within the given schedule is the *length* $t_{n+1} - t_1$ of the *Last-Last Information Flow* to t_{n+1}, if it exists. If there is no last-to-last information flow to t_{n+1}, then the delay is undefined [3].

Worst case Last-Last Information Propagation Delay (DLL_T) is the maximum age of data used in the computation. That is the maximum duration from the moment the system has responded to a change in the input to the moment at which the input has changed.

From the above two definitions , it is the maximum of *Last-Last Information Propagation Delays* taken over all schedules produced using the given task priorities and all times t_{n+1}, such that t_{n+1} is the finish time of an instance of T_n. This is required to make this definition consistent with DFF_T [3].

The point at which we break a schedule into two parts, where the first part of the schedule has certain properties and also the second part of it has other properties is called a pivot. More formal definition is given below and is used in various parts of this report. This is useful because all the schedules which have the worst case occurrences of the information propagation delays pivot at some value which we discuss later in the report.

> Let $T = < T_1, \ldots, T_n >$ be a task sequence with monotonically increasing priorities within a feasible task set. For a given natural number *t*, we say that a schedule S *pivots* at time t if the following properties are satisfied:
>
> **1.** Any task instance that begins executing before time t receives 1 unit of execution time.
>
> **2.** All other instances of tasks receive their maximum execution time.
>
> **3.** To break ties in priority, prior to time t, tasks in T are favored over tasks not in T , but beginning at time t, tasks not in T are favored over tasks in T [3].

3. Algorithms and Theorems

In this implementation we consider two kinds of task sequences. One is a sequence with monotonically decreasing priorities and other is a sequence with monotonically increasing priorities.

3.1 Monotonically Decreasing Priorities

Theorem for First-First Information Propagation Delay:

The following is quoted from [3].

> Let T = <T_1, . . . , T_n> be a task sequence with monotonically decreasing priorities within a feasible task set. Suppose P is the largest period of any task in the task set. Let S be any schedule that pivots at time P. Then this schedule contains a First-First information flow <t_0, . . . , t_n> with maximum length. Furthermore, t_n can be chosen to be P + R(T_n), where R(T_n) is the response time of $T_{n.}$

There is a formal proof of this in [3].

Theorem for relationship between dFF_T and dLL_T :

The following is quoted from [3].

> Let T = <T_1, . . . , T_n> be a task sequence within some feasible task set, and let us fix some schedule for this task set. Then for any time t, dFF_T (t) ≤ dFF_T (t + 1) = d iff dLL_T (t + d + 1) ≤ dLL_T (t + d) = d.

There is a formal proof of this in [3].

Algorithm:

The algorithm for computing worst case First-First Information Propagation Delay(DFF_T) first builds a portion of a schedule that pivots at time P, then finds the Last-Last Information

Flow to P + R(Tn) − 1 in this schedule. From above two theorems, DFF_T is then the length of this information flow plus 1.

3.2 Monotonically Increasing Priorities

Theorem for First-First Information Propagation Delay:

The following is quoted from [3].

> Let $T = <T_1, . . . , T_n>$ be a task sequence with monotonically increasing priorities within a feasible task set. Suppose P is the largest period in the task set. Then there is a schedule S containing a First-First information flow $<t_0, . . . , t_n>$ such that $0 < t_0 \leq P$, S pivots at t_0, and $t_n - t_0 = DFF_T - 1$.

There is a formal proof of this in [3].

Theorem for Last-Last Information Propagation Delay:

The following is quoted from [3].

> Let $T = <T_1, . . . ,T_n>$ be a task sequence with monotonically increasing priorities within a feasible task set, and suppose n > 1. Let P be the largest period in the task set. Then there is a schedule S containing a Last-Last information flow $<t_1, . . . , t_{n+1}>$ with length DLL_T such that $0 \leq t_1 < P$, t_{n+1} is the finish time of an instance of T_n, and S pivots at time $t_1 + 1$.

There is a formal proof of this in [3].

Algorithm:

The algorithm for computing worst case First-First Information Propagation Delay (DFF_T) and worst case Last-Last Information Propagation Delay (DLL_T) has to now compute several schedules that pivot at any time t_0, which is between 0 and P. And t_0 is one time unit after the start time of an instance of T_1. So for each of these possible t_0 values we compute schedules that pivot at t_0 . And from these schedules we compute the maximum First-First Information Propagation Delay and Last-Last Information Propagation Delay which are DFF_T and DLL_T respectively. While we are doing this it is necessary to adapt the algorithm for computing DFF_T, to also compute DLL_T in order to increase efficiency and avoid duplication of work. The details of how this is implemented are mentioned in section 4 of this report.

4. Design and Implementation

4.1 Monotonically Decreasing Priorities:

Approach:

We begin by creating a class Task (see Task in figure a) with attributes taskName, period, maxExecTime, priority, index (index in the task sequence) and present (indicates the presence or absence of the task in the task set).

We then create another class Schedule_Prefix (refer to Schedule_Prefix in figure a) with attributes taskName, startTime and finishTime. Whenever an instance of a task gets executed, an object of this class is created. So a list these objects represents the entire schedule of the task sequence within a certain time period.

The execution of the program starts in the main class Delay (refer Delay in figure a). Initially, the event handler, action is invoked and then the input file is accepted. This file has the task names of the tasks present in the task sequence first. They are stored in an ArrayList. The input file then has a set of tasks with task names, periods, maximum execution times and priorities. All the information is read and a task object is created by adding the additional information about index, present, etc.

We proceed by placing all the task objects into a priority queue (refer Priority queues in figure a) ordered by priorities. Since Java has an inbuilt priority queue for which we can provide any comparator class, I have used that data structure for this implementation.

Next, we sort the task objects by periods and find out the Maximum period and also the Minimum period and store this information (see Delay in figure a). All the above things are performed in the action method.

Now, we begin scheduling the tasks as follows. For the time period from 0 to P (Maximum period), we schedule each task for 1 time unit according to priorities. I have a logical global timer (see timer in figure a) set up for this purpose. This is done by just removing the task objects from priority queue and increasing the timer. If tasks are of equal priority, we give preference to those in the task set. This is taken care of in the comparator class. This is accomplished by making a call to the schedulePrefixFirst method in the Delay class.

When the above procedure is being executed, we check for the availability of tasks at each Integral multiple of the shortest period. If any task becomes available we place it in the priority queue. This is done by making a call to the checkTaskAvailability method in the Delay class.

After completion of building the schedule till P, we continue as follows. For the time period P to 2P, we schedule each task for a time of its maximum Execution time according to priorities. If tasks are of equal priority, we give preference to those not in the task set. And we stop if any instance of the last task in the task set has finished execution. All this is implemented in the same way as the first part of the schedule prefix that has already been built with the necessary changes. This is accomplished by making a call to the schedulePrefixSecond method in the Delay class.

Now that we have the schedule prefix, from the computed scheduled prefix, we compute the Last-Last Information Flow to P + R(Tn) – 1 by scanning the schedule prefix once backward.

We store the information in an ArrayList. This is done by making a call to the lastLastDelay method in the Delay class.

From the above computed Last-Last Information Flow, we compute the First-First Information Flow by scanning the schedule prefix once forward and we store the information in an ArrayList. Now, the difference of first and last entry in this flow plus 1 gives us the DFF_T. This is accomplished by making a call to the firstFirstInfoDelay method in the Delay class.

There is also a formal proof that this is the worst possible delay in [3].

Challenges:

In the implementation each task could execute multiple instances. So keeping track of the finish times of each individual task instance and taking care of preemption of tasks was the main challenge.

The entire implementation and mainly building the schedule prefix and computing the delays were to be done efficiently.

Since the two parts of the schedule prefix were very similar to each other, reusing the code so that the two parts of the schedule prefix are computed without duplication of work and increase of complexity was necessary.

Deciding the appropriate data structures to represent schedule prefix, Information flow and other things was important since it could impact the efficiency.

Preparing a comprehensive set of test cases that covers rate monotonic and other forms of scheduling was another challenge.

Finally, providing an easy to use interface so that the end user can test this tool against his own set of test cases was required. This is done using Java applets.

Currently there is no efficient algorithm known for finding the worst case Last-Last Information Propagation Delay for this kind of task sequences in [3].

4.2 Monotonically Increasing Priorities

Approach:

We start by creating the same classes and data structures that were used in the implementation of previous algorithm. The only difference is that the class Task has an additional attribute indexInSequence to indicate the index of the task in the task sequence. It is set as -1 if the task is not present in the sequence. This indexInSequence is also included in Schedule_Prefix class, but called as Index in it (refer figure b).

The execution of the program starts in the main class Delay (refer Delay in figure b). Initially, the event handler, action is invoked and then the input file is accepted and we perform the same read and store information operations as before. The only difference is we assign the indexInSequence attribute its value.

We then store the task objects (see Task in figure b) in the priority queue (see Priority queues in figure b) and sort the tasks according to periods and find out the maximum and minimum periods in the same way as above.

Now, we construct a schedule as follows. For the time period from 0 to P (max period), we schedule each task for a time unit of 1 according to priorities. If tasks are of equal priority, we give preference to those in the task set. This is implemented exactly as we have done before. This is accomplished by making a call to the schedulePrefixFirst method in the Delay class.

From the above computed schedule prefix, we find out all the time values where an instance of first task in the task set finishes. We store them in an ArrayList. These are the possible values for the start of the second part of the schedule prefix. This is done by making a call to the possibletValues method in the Delay class.

From the above computed list, for each **value** from the list, we compute the information flows and information propagation delays as follows

We schedule each task for their maximum execution times from time **value** plus **1** to P using the same techniques as before. We store this information in a list. And we do the same from time P to 2P. Notice that this schedule is cyclic with period P. So we store this information in a separate list and also add this information to the other list. Here if tasks are of equal priority, we give preference to those that are not present in the task set. This is accomplished by making a call to the schedulePrefixSecond method in the Delay class.

Now, for each finish time no later than 2P of a task T_i, where $1 \leq i < n$, we compute the finish time of the next instance of T_{i+1}. These finish times are stored in an array F[1..2P] so that if T_i finishes at time t, then F[t] gives the finish time of the next instance of T_{i+1}. To do this, we have a temporary list of ArrayLists indexed by the tasks in the task sequence.

We initialize these ArrayLists with nulls. We then proceed by scanning the schedule prefix forwards. For each finish time **f** of a task instance, if its indexInSequence (let us call it j for now) is not the last task index, we add **f** to the ArrayList at index j of the temporary list. Now we check if j is greater than 0 and then retrieve the j-1 entry in the temporary list. And for all the values **V** in that ArrayList, if **f** is greater than **V**, we write F[**V**] = **f**. We continue this way until we reach the end of schedule prefix. Now if any of the values **V** do not have an entry F[**V**], then making use of the cyclic schedule prefix we wrap around it and add the maximum period to its values to fill in the missing values using the same technique above. This is accomplished by making a call to the finishTimesArray method in the Delay class.

We compute the First-First Information flow by traversing through the array **F**[1..2P]. Notice that we might need information beyond 2P in which case we make use of the cyclic property of the schedule after 2P and efficiently find out the necessary value by wrapping around the array **F**[1...2P] and adding the maximum period to its values. From this information flow we have the First-First-Information propagation delay which is the difference of last and first entries in the flow plus 1. This is done by making a call to the firstFirstInfoDelay method in the Delay class.

We now calculate the finish time of an instance of T_n that finishes before the last entry in the First-First Information Flow (let us call the last entry as L). This is done by first finding out all the finish times of instances of T_n in the time period **value** to 2P and we store them in a list. Since the schedule is cyclic after 2P, L can be reduced to be within P and 2P. Hence a single scan of the above computed list gives us the greatest value less than the reduced L. This value can be elevated to be in the range of the original L value by adding the maximum period necessary number of times (let us call it N). Now, N minus **value** minus 1 gives us the Last-Last Information Propagation Delay.

We notice that there might not necessarily be a Last-Last Information Flow to N, because it is not guaranteed that all the tasks in the task sequence from $T_2 \ldots T_{n-1}$ have executed at least one of their instances between **value** and N. However it does not affect the overall worst case delay because, the N value computed is not even the worst case delay for this particular schedule, so it cannot be the worst case delay computed over all schedules. Hence we simply ignore it. All of this is accomplished by making a call to the lastLastInfoDelay method in the Delay class.

After computing both the First-First Information Propagation Delay and Last-Last Information Propagation Delay for all the **values.** We take the maximum for both sets which gives us DFF_T and DLL_T respectively.

There is also a formal proof that this is the worst possible delay in [3].

Challenges:

One of the main challenges while implementing this algorithm was to compute several schedule prefixes efficiently.

Since the information flow might exceed 2P, the other challenge was to figure out an efficient way to compute the entire flow without extending the schedule prefix beyond 2P, since it could be very large and the run time could increase drastically. This is done just using arrays and lists.

The last-last delay could be computed using the same information that is used to calculate first-first delay. So using the same functions and components for computing both these delays was another challenge.

Also, preparing a comprehensive set of test cases that covers rate monotonic and other forms of scheduling for both delays was important.

Figure a: Monotonically Decreasing Priorities

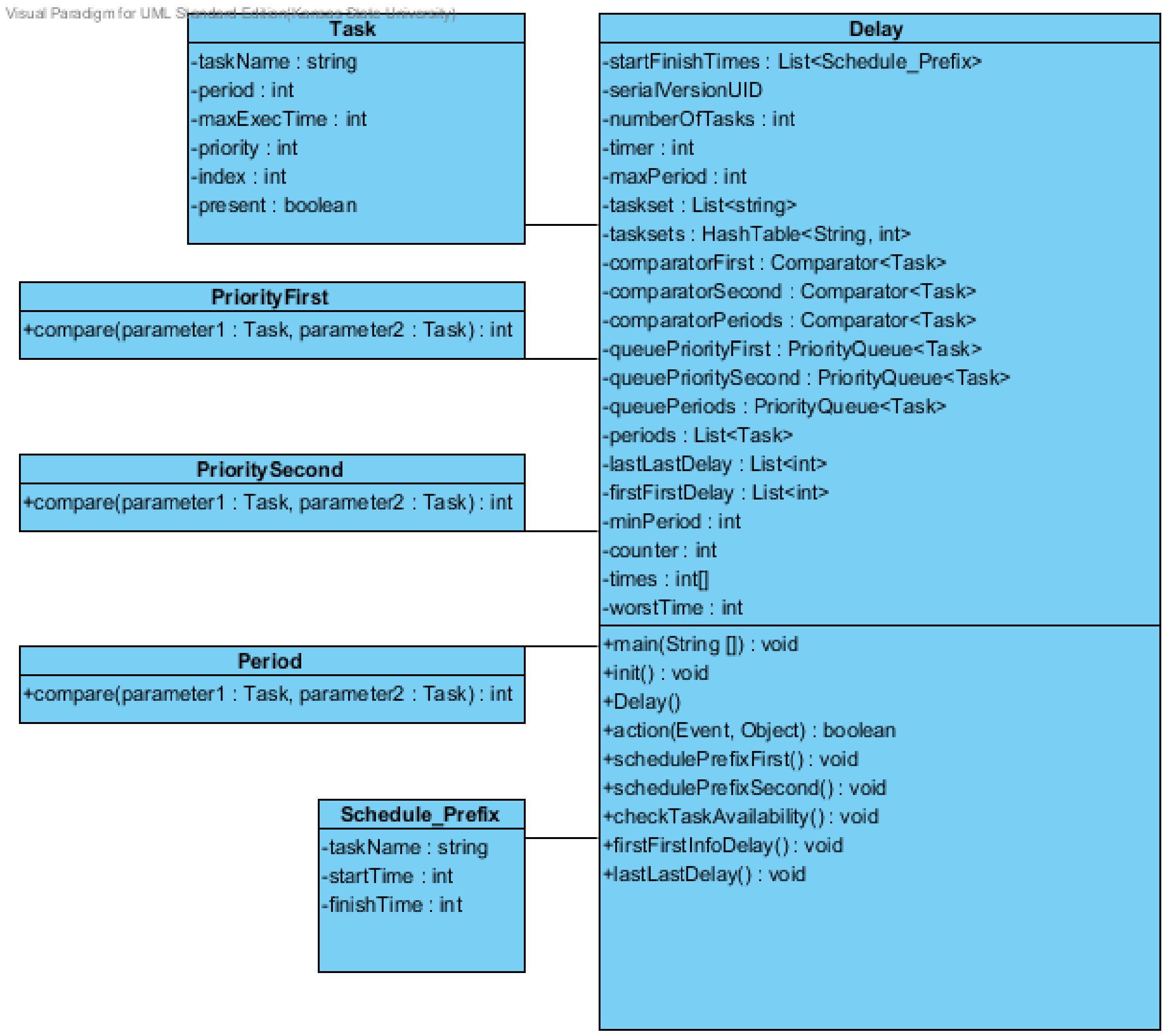
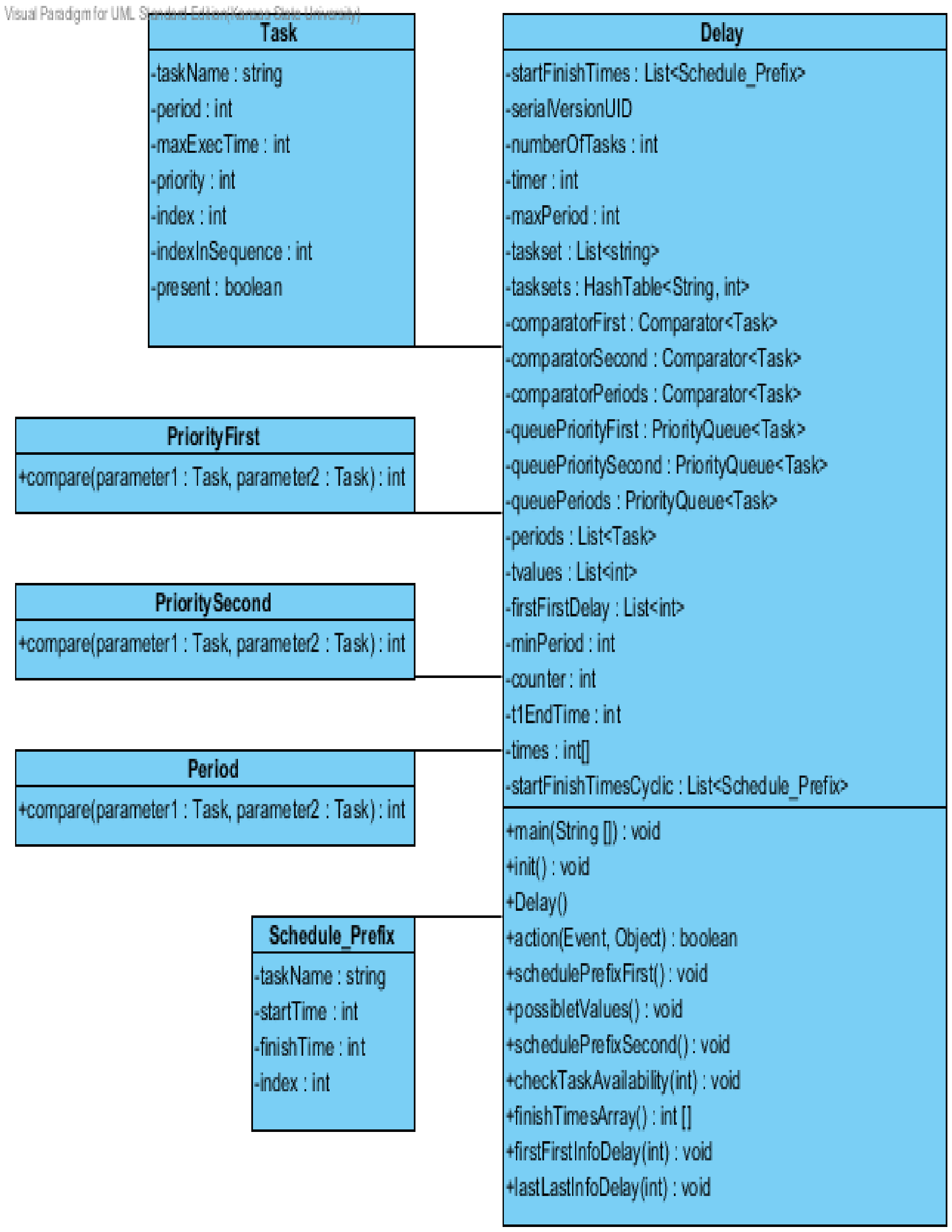

Visual Paradigm for UML Standard Edition(Kansas State University)
Task
-taskName : string
-period : int
-maxExecTime : int
-priority : int
-index : int
-present : boolean
PriorityFirst
+compare(parameter1 : Task, parameter2 : Task) : int
PrioritySecond
+compare(parameter1 : Task, parameter2 : Task) : int
Period
+compare(parameter1 : Task, parameter2 : Task) : int
Schedule_Prefix
-taskName : string
-startTime : int
-finishTime : int
Delay
-startFinishTimes : List<Schedule_Prefix>
-serialVersionUID
-numberOfTasks : int
-timer : int
-maxPeriod : int
-taskset : List<string>
-tasksets : HashTable<String, int>
-comparatorFirst : Comparator<Task>
-comparatorSecond : Comparator<Task>
-comparatorPeriods : Comparator<Task>
-queuePriorityFirst : PriorityQueue<Task>
-queuePrioritySecond : PriorityQueue<Task>
-queuePeriods : PriorityQueue<Task>
-periods : List<Task>
-lastLastDelay : List<int>
-firstFirstDelay : List<int>
-minPeriod : int
-counter : int
-times : int[]
-worstTime : int
+main(String []) : void
+init() : void
+Delay()
+action(Event, Object) : boolean
+schedulePrefixFirst() : void
+schedulePrefixSecond() : void
+checkTaskAvailability() : void
+firstFirstInfoDelay() : void
+lastLastDelay() : void

Figure b: Monotonically Increasing Priorities


Visual Paradigm for UML Standard Edition(Kansas State University)
Task
-taskName : string
-period : int
-maxExecTime : int
-priority : int
-index : int
-indexInSequence : int
-present : boolean
PriorityFirst
+compare(parameter1 : Task, parameter2 : Task) : int
PrioritySecond
+compare(parameter1 : Task, parameter2 : Task) : int
Period
+compare(parameter1 : Task, parameter2 : Task) : int
Schedule_Prefix
-taskName : string
-startTime : int
-finishTime : int
-index : int
Delay
-startFinishTimes : List<Schedule_Prefix>
-serialVersionUID
-numberOfTasks : int
-timer : int
-maxPeriod : int
-taskset : List<string>
-tasksets : HashTable<String, int>
-comparatorFirst : Comparator<Task>
-comparatorSecond : Comparator<Task>
-comparatorPeriods : Comparator<Task>
-queuePriorityFirst : PriorityQueue<Task>
-queuePrioritySecond : PriorityQueue<Task>
-queuePeriods : PriorityQueue<Task>
-periods : List<Task>
-tvalues : List<int>
-firstFirstDelay : List<int>
-minPeriod : int
-counter : int
-t1EndTime : int
-times : int[]
-startFinishTimesCyclic : List<Schedule_Prefix>
+main(String []) : void
+init() : void
+Delay()
+action(Event, Object) : boolean
+schedulePrefixFirst() : void
+possibletValues() : void
+schedulePrefixSecond() : void
+checkTaskAvailability(int) : void
+finishTimesArray() : int []
+firstFirstInfoDelay(int) : void
+lastLastInfoDelay(int) : void

5. Testing and Results

This implementation requires testing on various kinds of task sequences. I have listed some of the test cases and types of sequences considered for both the algorithms. Also the results are analyzed and interpreted.

User Interface:

The user interface is very simple. A Java Applet is launched upon running the application (you can also run it as a Java application) and has an *OPEN* and *CLOSE* button. The user can open a text file by clicking on the OPEN button and immediately the output is displayed on the applet viewer. The user can continue to open other input text files and each time the output is displayed on the same screen. When finished testing, you can exit the application by clicking the CLOSE button.

Input Format:

A text file containing a Task set indicated by Task names, followed by the Task sequence. Each Task has a Task name, Period, Maximum Execution time and Priority in this order. First "END" in the file indicates the end of task set and second "END" indicates the end of input. The program assumes that each input file is correct and does not check for obvious errors like spelling mistakes, incorrect order of input, values that contradict our assumptions etc. Every task sequence should have a feasible schedule.

Output Format:

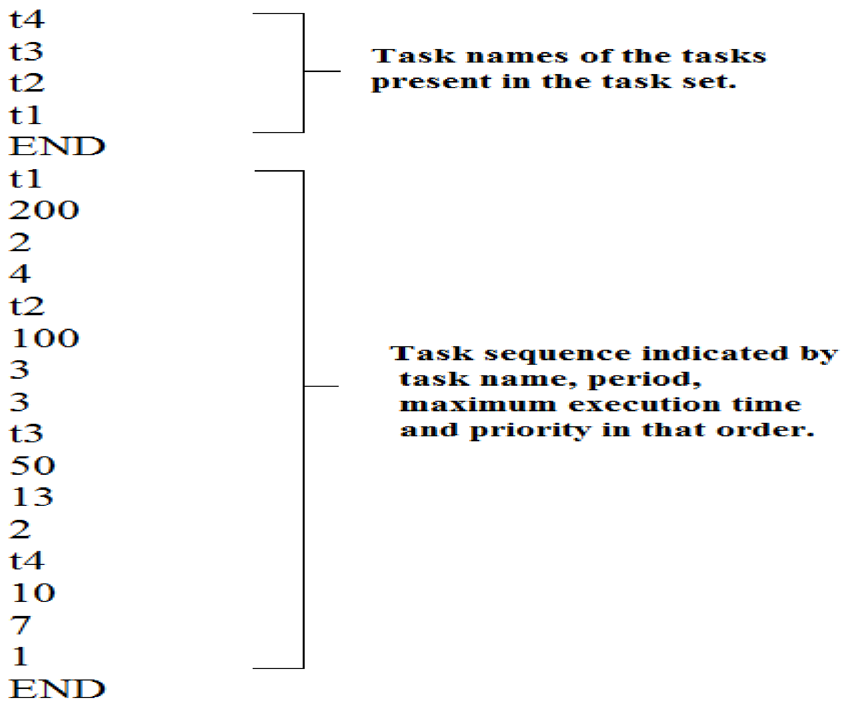
The computed Information flows and Information propagation delays are printed on the Java Applet. Also it might display some additional information about the tasks.

5.1 Types of test cases for Monotonically Decreasing Priorities:

a. Rate Monotonic Task sequences with all the tasks in the sequence also present in the task set:

Here, the scheduling algorithm used is rate monotonic scheduling. And the task set contains all the tasks from the sequence. In this case the worst case First-First Information Flow starts from 0 and ends at P(maximum period) + R(T_N) and Worst Case First-First Information Propagation Delay is always P + R(T_N).

Sample Input File:

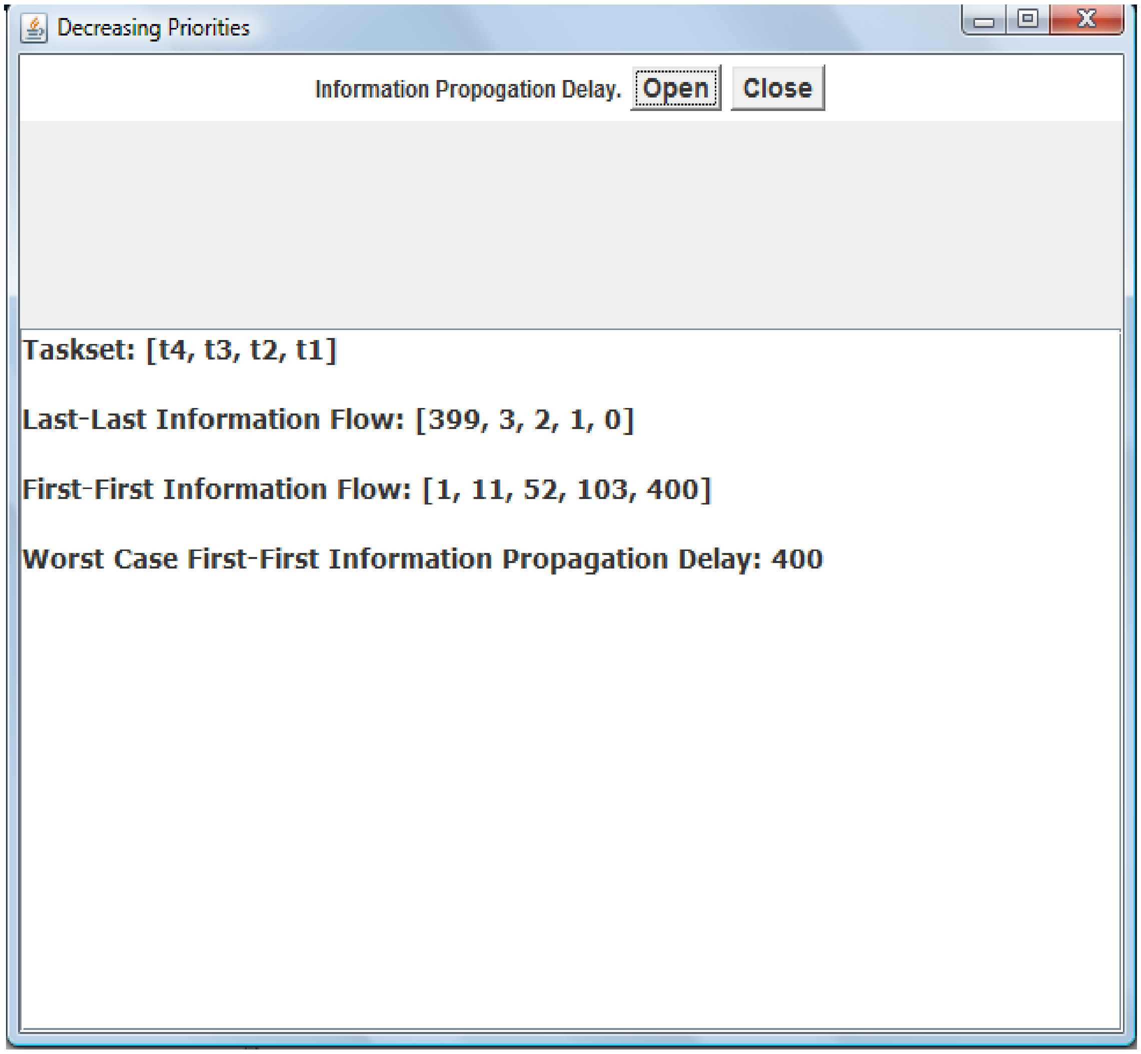


```
t4
t3
t2
t1
END
t1
200
2
4
t2
100
3
3
t3
50
13
2
t4
10
7
1
END
```

Task names of the tasks present in the task set.

Task sequence indicated by task name, period, maximum execution time and priority in that order.

Output for the above Input File:



```
Taskset: [t4, t3, t2, t1]

Last-Last Information Flow: [399, 3, 2, 1, 0]

First-First Information Flow: [1, 11, 52, 103, 400]

Worst Case First-First Information Propagation Delay: 400
```

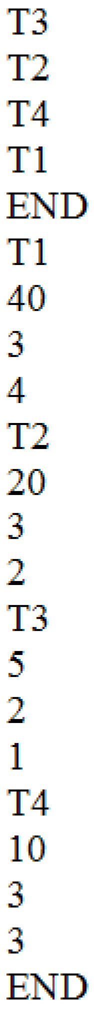
Analysis and Interpretation:

From the above output screen shot we can see that for the given task sequence, the Worst case First-First Information Propagation Delay from Task T_4 to Task T_1 in the sequence is 400, which is exactly as we predicted since P= 200 and $R(T_N)$=200. Hence the task T_1 might take up to 400ms to produces an output after the task T_4 has detected a change in the Input and propagated this information to T_1.

b. Non Rate Monotonic Task sequences with all the tasks in the sequence also present in the task set:

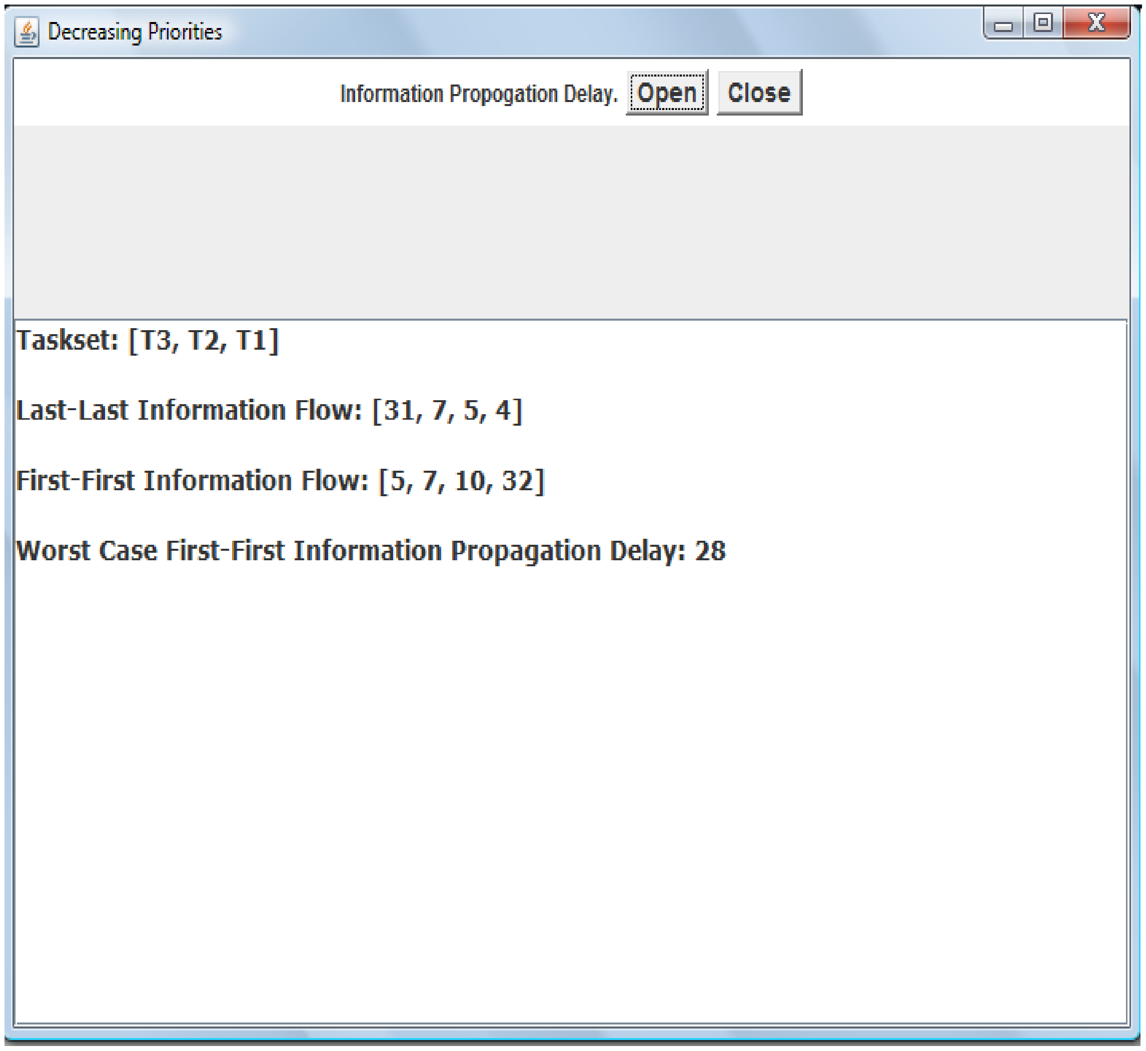
Here, the scheduling algorithm used is non rate monotonic scheduling, so the task priorities do not depend on the periods. And the task set contains all the tasks from the sequence. In this case also the worst case First-First Information Flow starts from 0 and ends at P(maximum period) + R(T_N) and Worst Case First-First Information Propagation Delay is always P + R(T_N).

Sample Input File:



```
T3
T2
T4
T1
END
T1
40
3
4
T2
20
3
2
T3
5
2
1
T4
10
3
3
END
```

Output for the above Input File:



```
Taskset: [T3, T2, T1]

Last-Last Information Flow: [31, 7, 5, 4]

First-First Information Flow: [5, 7, 10, 32]

Worst Case First-First Information Propagation Delay: 28
```

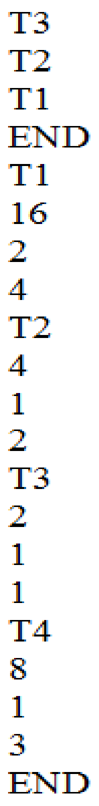
Analysis and Interpretation:

From the above output screen shot we can see that for the given task sequence, the Worst case First-First Information Propagation Delay from Task T_3 to Task T_1 in the sequence is 60, which is exactly as we predicted since P= 40 and $R(T_N)=20$. Hence the task T_1 might take up to 60ms to produces an output after the task T_3 has detected a change in the Input and propagated this information to T_1.

c. Non Rate Monotonic Task sequences with task set containing only some of the tasks from the task sequence.

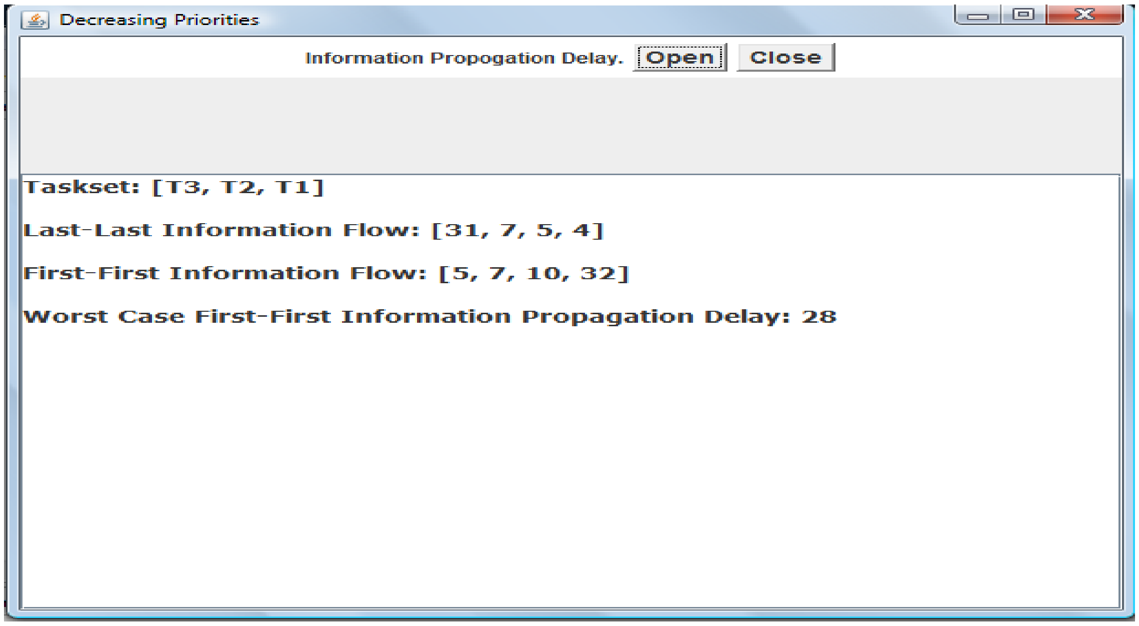
Here, the scheduling algorithm used is non rate monotonic scheduling, so the task priorities do not depend on the periods. And the task set does not contain all the tasks from the sequence. In this case the worst case First-First Information Flow need not start from 0 and still ends at P(maximum period) + R(T_N) and Worst Case First-First Information Propagation Delay is always less than P + R(T_N).

Sample Input File:



```
T3
T2
T1
END
T1
16
2
4
T2
4
1
2
T3
2
1
1
T4
8
1
3
END
```

Output for the above Input File:



```
Taskset: [T3, T2, T1]

Last-Last Information Flow: [31, 7, 5, 4]

First-First Information Flow: [5, 7, 10, 32]

Worst Case First-First Information Propagation Delay: 28
```

Analysis and Interpretation:

From the above output screen shot we can see that for the given task sequence, the Worst case First-First Information Propagation Delay from Task T_3 to Task T_1 in the sequence is 28, which is exactly less than 32, since P = 16 and $R(T_N)$ = 16. Hence the task T_1 might take up to 28ms to produces an output after the task T_3 has detected a change in the Input and propagated this information to T_1.

5.2 Types of test cases for Monotonically Increasing Priorities:

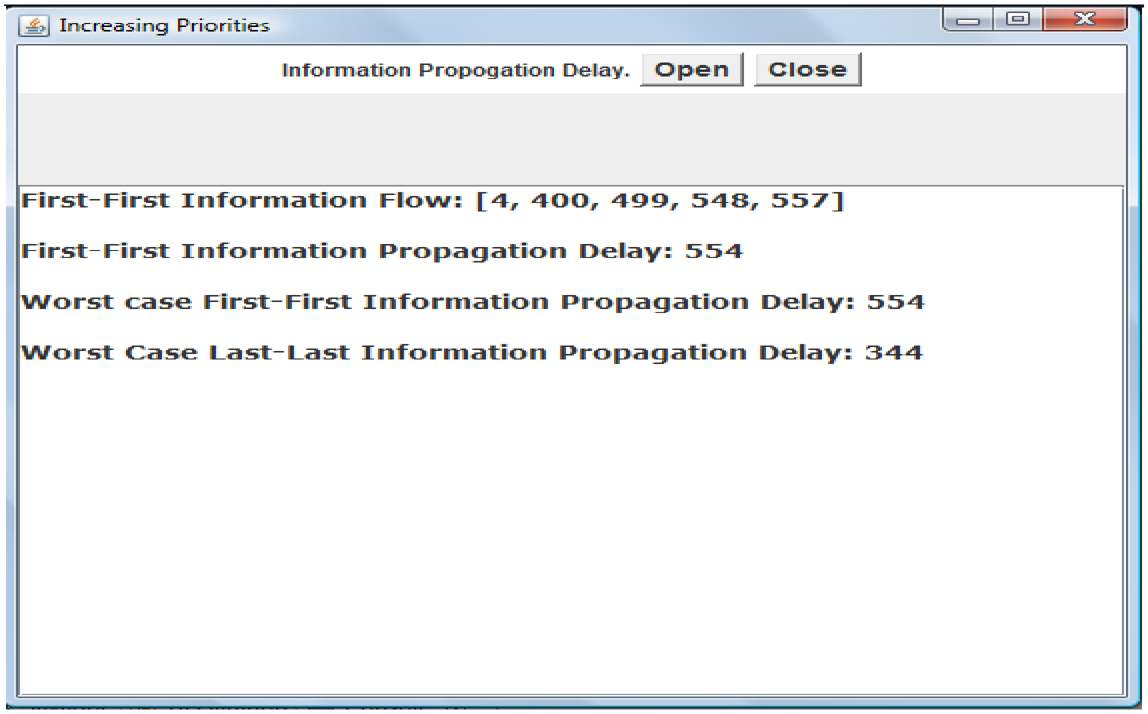
a. Rate Monotonic Task sequences with all the tasks in the sequence also present in the task set:

Here we need to compute only one schedule since there is only one possible t_0 value. In this case Worst Case First-First Information Propagation Delay and Worst Case Last-Last Information Propagation Delay are always less than sum of the periods plus period of the first task in the sequence.

Sample Input File:

```
T1
T2
T3
T4
END
T1
200
2
4
T2
100
3
3
T3
50
13
2
T4
10
7
1
END
```

Output for the above Input File:



```
First-First Information Flow: [4, 400, 499, 548, 557]

First-First Information Propagation Delay: 554

Worst case First-First Information Propagation Delay: 554

Worst Case Last-Last Information Propagation Delay: 344
```

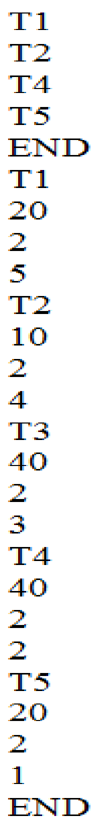
Analysis and Interpretation:

From the above output screen shot we can see that for the given task sequence, the Worst case First-First Information Propagation Delay from Task T_1 to Task T_4 in the sequence is 554 and Worst case Last-Last Information Propagation Delay is 344. Both these are less than 560(sum of the periods, 360 + 200). Hence the task T_1 might take up to 28ms to produces an output after the task T_3 has detected a change in the Input and propagated this information to T_1.

b. Non Rate Monotonic Task sequences with task set containing only some of the tasks from the task sequence.

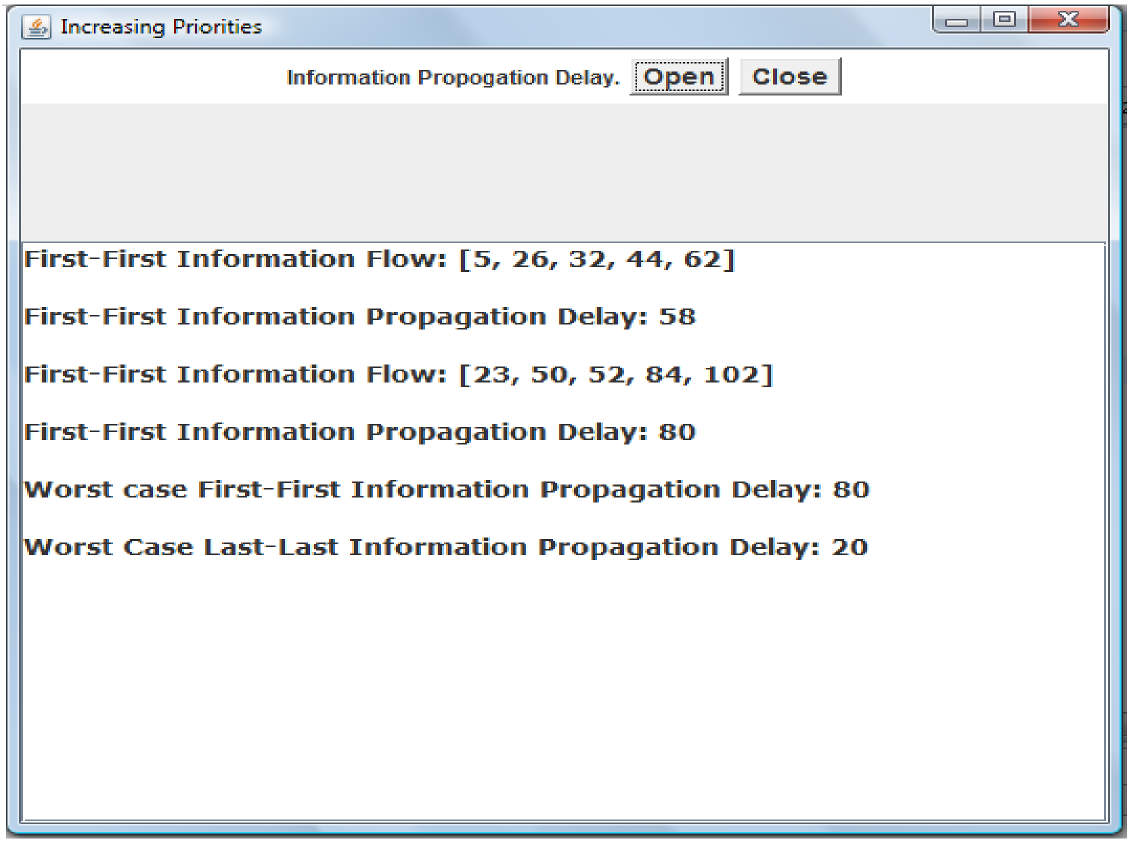
Here we need to compute several schedules since there are many possible t_0 values. In this case also the Worst Case First-First Information Propagation Delay and Worst Case Last-Last Information Propagation Delay are always less than sum of the periods plus period of the first task in the sequence.

Sample Input File:



```
T1
T2
T4
T5
END
T1
20
2
5
T2
10
2
4
T3
40
2
3
T4
40
2
2
T5
20
2
1
END
```

Output for the above Input File:



```
First-First Information Flow: [5, 26, 32, 44, 62]

First-First Information Propagation Delay: 58

First-First Information Flow: [23, 50, 52, 84, 102]

First-First Information Propagation Delay: 80

Worst case First-First Information Propagation Delay: 80

Worst Case Last-Last Information Propagation Delay: 20
```

Analysis and Interpretation:

From the above output screen shot we can see that for the given task sequence, the Worst case First-First Information Propagation Delay from Task T_1 to Task T_4 in the sequence is 80 and the Worst case Last-Last Information Propagation Delay is 20. Both these are less than 170 (sum of the periods 130 +40). Hence the task T_1 might take up to 28ms to produces an output after the task T_3 has detected a change in the Input and propagated this information to T_1.

The application has been thoroughly tested against various other task sequences and the output is exactly as we expected.

6. Performance Analysis

The performance of this application is measured in terms of time complexity rather than the response time since the response time is usually very good for the test cases we used.

6.1 Run Time Analysis for Algorithm for Monotonically Decreasing Priorities:

For constructing the first part of the scheduled prefix that is from 0 to P(maximum period in the task set), if P_k is the largest period in the task sequence, then there can be at most P_k task instances that can arrive. So adding and removing the tasks into the priority queue can take place that many times. Hence this can take $O(P_k \log m)$ time, where m is the number of tasks in the task set.

Now for constructing the second part of the scheduled prefix that is from P to $R(T_N)$, where T_N is the last task in the task set. If P_n is its period, then it can take $O(P_n \log m)$ time. Since P_n is less than or equal to P_k, the run time for this is also in $O(P_k \log m)$.

And for computing the Last-Last Information flow and First-First Information flow it is sufficient to scan the schedule prefix once. So it is in $O(P_k)$. Hence the total run time is in $O(P_k \log m)$.

6.2 Run Time Analysis for Algorithm for Monotonically Increasing Priorities:

For constructing the scheduled prefix that is from P(maximum period in the task set) to 2P, from the above analysis the run time is in O(P log m), where m is the number of tasks in the task set

If P_1 is the period of the first task in the task sequence, then there can only be P/ P_1 possible values for the start time of the schedule prefix we need to build. Hence these schedules can be build in $O(P^2 \log m / P_1)$.

Given the schedule prefix, the array to store the finish times can be computed by scanning the schedule once. Hence it is in O(P). And from this array, the first-first information flow and last-last information flow can be computed in O(n) time, where n is the number of tasks in the task sequence. Hence the total run time is in $O(P^2 \log m / P_1)$.

7. Scope, Conclusion and Future Work

7.1 Scope

The application clearly demonstrates the ideas and algorithms presented in [1] and [3]. Using it the user can test with their own set of test cases and analyze how bad these worst case propagation delays can be. They can also try and adjust the priority assignments or periods or change other constraints and see if the delays are reduced.

It avoids the manual computation of these delays which saves a lot of time and effort. Although currently we are operating on a limited set of tasks that match our assumptions, it can be easily extended to include many other kinds of task sequences and also other constraints can be added.

7.2 Conclusion

We have shown that the worst case first-first Information propagation delays for any task sequence that match our assumptions are within the bounds mentioned in [1]. Also we have shown several examples where the delays approach the bounds. We have also shown that for the commonly occurring case in which the periods within the sequence are monotonically decreasing and the priorities are rate monotonic, the delay is always less than three times the longest period, though it can come arbitrarily close to this bound.

Also the worst case last-last Information propagation delays are analyzed for the case where task sequences are monotonically increasing. The results have been shown using which we can have a good idea about the bounds.

The whole process of development has been a great learning experience where concepts of real time scheduling along with concepts of Information propagation delays have been covered. Also programming using the Java language has allowed me to understand the power of its built in data types and the advantages of object oriented programming. The whole process of developing this project was like being involved in a work from learning concepts to implementing them and every aspect of it has been very informative.

7.3 Future Work

Implementing the case where task sequences have mixed priorities that are neither monotonically increasing nor decreasing. There is an algorithm for these kinds of task sequences in [3].

We can extend the application to operate on task sequences whose periods need not be harmonically related. It can be done by making very few changes to the current implementation.

It can also be extended to include task sequences that follow different priority schemes apart from fixed priority scheduling.

Finally, if the application can work as an analysis tool which adjusts the priority assignments or periods so that the delays are reduced, it would be very useful.

References

[1] Rodney R. Howell and Masaaki Mizuno, “Propagation Delays in Fixed-Priority Scheduling of Periodic Tasks,” in 22nd Euromicro Conference on Real-Time Systems, pp. 219-228, Brussels, Belgium, 2010.

[2] N. Feiertag *et al.*, “A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics,” in *Proceedings of the IEEE Real-Time System Symposium (RTSS), Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, Barcelona, Spain, 2008.

[3] Rodney R. Howell and Masaaki Mizuno, “Computing Information Propagation Delays Through Sequences of Fixed-Priority Periodic Tasks,” Department of Computing and Information Sciences, Kansas State University (In Preparation).

[4] Java SE 6 Documentation, http://download.oracle.com/javase/6/docs.

[5] C. Liu and J. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *J. ACM*, vol. 20, pp. 46–61, 1973.

[6] Y. Yamamoto, “Sampled-data control theory: The past and the future,” *Information and Control*, vol. 40, no. 1, pp. 76–81, 2001.

Appendix A - Code for Algorithm 1

Delay.java

```
/** Program for computing First to First Information
 *  propagation Delay in a Periodic task sequence with
 *  decreasing Priorities.

 *  Input : A text file containing the Task set indicated by Task names,
 *  followed by the Task sequence. Each Task has a Task name, Period, Maximum
 *  Execution time and Priority in this order. First "END" indicates the end of
 *  Task set and second "END" indicates the end of input. The program assumes
 *  each input file is correct and does not check for errors. Every task
 *  sequence has a feasible schedule.

 *  Output : The First to First Information Flow is printed on the
 *  java Applet.

 * @author Vineet Tadakamalla
 */

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;
import java.util.List;
import java.awt.*;
import javax.swing.*;

/**
 * This class represents the task structure. All the entries are found from
 * input except for Index which is the just order of tasks.
 */
class Task {
	String taskName; // Name of the task
	Integer period; // Period of the task
	Integer maxExecTime; // Maximum execution time of the task
	Integer priority; // Priority of the task
	Integer index; // Index which is just the sequence in which they are entered
	Boolean present; // To indicate whether the task is in task set or not
}

/**
```

```
 * This class represents the schedule prefix constructed which helps in
 * calculating the first to first information flow.
 */
class Schedule_Prefix {
	String taskName; // Name of the task
	Integer startTime; // start time of each instance of a task
	Integer finishTime; // Finish time of each instance of a task
}

public class Delay extends JApplet {
	private static final long serialVersionUID = 1L;
	/**
	 * number of tasks in the sequence timer to keep track of schedule Maximum
	 * period in the sequence
	 */
	private int numberOfTasks = 0, timer = 0, maxPeriod = 0;
	/**
	 * task set for which the delay has to be calculated. Obtained from input
	 * file
	 */
	private List<String> taskset = new ArrayList<String>();
	/**
	 * used to temporarily store the task names present in the task set
	 */
	private Hashtable<String, String> tasksets = new Hashtable<String, String>();
	/**
	 * Declare a comparator instance for Priority queue First
	 */
	private Comparator<Task> comparatorFirst = new PriorityFirst();
	/**
	 * Declare a comparator instance for Priority queue Second
	 */
	private Comparator<Task> comparatorSecond = new PrioritySecond();
	/**
	 * Declare a comparator instance for Period queue
	 */
	private Comparator<Task> comparatorPeriods = new Period();
	/**
	 * Declare Priority queue to store tasks in first part of Schedule prefix
	 */
	private PriorityQueue<Task> queuePriorityFirst = new PriorityQueue<Task>(
			10, comparatorFirst);
	/**
	 * Declare Priority queue to store tasks in second part of Schedule prefix
	 */
	private PriorityQueue<Task> queuePrioritySecond = new PriorityQueue<Task>(
                    10, comparatorSecond);
/**
 * Declare Priority queue to sort periods and also store tasks
 */
private PriorityQueue<Task> queuePeriods = new PriorityQueue<Task>(10,
                    comparatorPeriods);
/**
 * used to temporarily store the task set
 */
private List<Task> periods = new ArrayList<Task>();
/**
 * last to last delay from which we calculate the actual first to first
 * delay
 */
private List<Integer> lastLastDelay = new ArrayList<Integer>();
/**
 * The first to first information propagation delay
 */
private List<Integer> firstFirstDelay = new ArrayList<Integer>();
/**
 * Minimum period of the sequence Worst_time helps in calculating the
 * information flow. counter to keep track of each task instance execution
 * time
 */
private Integer minPeriod = 0, worstTime = 0, counter = 0;
/**
 * stores start time of current task instance reference by task index.
 */
private Integer[] times = new Integer[10];
/**
 * The schedule prefix which contains the task name, start and finish times.
 */
private List<Schedule_Prefix> startFinishTimes = new ArrayList<Schedule_Prefix>();
private File ffile; // Input File
private JLabel title; // Title of the Output
private JTextArea display; // Display Area
private JPanel p; // Panel to hold the text area

/**
 * the main method for the program. Creates a Java Applet and a JFrame,
 *
 * @param args
 *          The command-line arguments. Not used in this program, but
 *          required by Java
 */
```

```
public static void main(String[] args) {
        JApplet theApplet = new Delay(); // Create an instance of Delay class.
        theApplet.init(); // Initialize the Applet
        JFrame window = new JFrame("Decreasing Priorities"); // Title
        window.setContentPane(theApplet);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.pack();
        window.setVisible(true);
}

/**
 * Initializes Applet with dimensions 500*500
 *
 * @return Does not return anything.
 */
public void init() {
        setSize(500, 500);
}

/**
 * Constructor for the Class Delay used to set up the user Interface Sets
 * the Panel and display area.
 */
public Delay() {
        p = new JPanel(); // Create a Panel
        title = new JLabel(); // Create a Label
        Font f; // Declare a font
        // Create a Text Area with dimensions 20*30
        display = new JTextArea(20, 30);
        // Initialize the font to "Verdana", with Bold and size 14.
        f = new Font("Verdana", Font.BOLD, 14);
        display.setFont(f); // Set the font for the display area
        display.setEditable(false); // Make it non editable
        // Set where it should be displayed. Here it is South
        getContentPane().add(new JScrollPane(display), BorderLayout.SOUTH);
        p.add(title, 0); // Add title to the Panel
        title.setText("Information Propogation Delay.\n"); // Set title
        p.setFont(f); // Set Font for panel
        p.add(new Button("Open"), -1); // Add open button to open the input file
        p.add(new Button("Close")); // Add close button to close the applet
        // Set background color to white
        p.setBackground(new Color(255, 255, 255));
        add("North", p); // Add the panel to the north of the applet.

        /* End of UI */
}
```

```
/**
 * action is the Implemented method of the Japlpet interface
 *
 * @param A
 *          mouse click event which is either open or close.
 * @return Boolean to indicate success
 */
public boolean action(Event event, Object arg) {
        if (arg.equals("Close")) {
                System.exit(0); // If the user clicks close button, exit the applet
        }
        if (arg.equals("Open")) { // If the user clicks open
                Frame parent = new Frame(); // Add a frame
                FileDialog fd = new FileDialog(parent, "Please choose a file:",
                                FileDialog.LOAD); // Pop the file dialog box to open the
                // input file
                fd.setVisible(true);
                String selectedItem = fd.getFile();
                if (selectedItem == null) {
                        // Handle File not found here.
                } else {
                        ffile = new File(fd.getDirectory() + File.separator
                                        + fd.getFile()); // accept the input file into ffile
                        // Declare a buffered reader br to read the file
                        BufferedReader br;
                        String str = null; // Temporary variable to store each entry
                        try {
                                // Store the input file in br
                                br = new BufferedReader(new FileReader(ffile));
                                str = br.readLine(); // start reading line by line
                                while (!str.contains("END")) { // Stop if line is "END"
                                        taskset.add(str); // Add the string into task set list
                                        tasksets.put(str, str); // Also make an entry in the
                                                                // Hashtable
                                        str = br.readLine(); // Read next line
                                }
                                str = br.readLine(); // Continue Reading the file

                                while (!str.contains("END")) { // Stop if the line is "END"
                                Task task = new Task(); // create a Task instance task
                                numberOfTasks++; // Increase the variable for each task
                                task.taskName = str; // first entry is the task name
                                str = br.readLine();
                                task.period = new Integer(str); // second is task period
                                str = br.readLine();
// third is maximum execution time
task.maxExecTime = new Integer(str);
str = br.readLine();
task.priority = new Integer(str); // fourth is priority
task.index = numberOfTasks - 1; // Add the index
        /**
         * If Hashtable contains an entry then the task is
         * present in the task set.
         */
        if (tasksets.get(task.taskName) != null)
                task.present = true;
        else
                task.present = false;
        // Store the tasks in the priority Queue
        queuePriorityFirst.add(task);
        // Also store it in the Second priority Queue
        queuePeriods.add(task);
        str = br.readLine();
}

/* sorting the periods */
while (queuePeriods.size() != 0) {
        periods.add(queuePeriods.remove());
}
Task per = new Task();
per = periods.get(numberOfTasks - 1);
// Set the maximum period which is the last entry of periods
// list
maxPeriod = new Integer(per.period);
per = periods.get(0);
// Set the minimum period which is the first entry of
// periods list
minPeriod = new Integer(per.period);
// Set all current task instance start time to zero
for (int i = 0; i < times.length; i++)
        times[i] = 0;
// call the below methods
schedulePrefixFirst();
schedulePrefixSecond();
lastLastDelay();
firstFirstInfoDelay();
taskset.clear();
periods.clear();
lastLastDelay.clear();
firstFirstDelay.clear();
numberOfTasks = 0;
```

```
                    timer = 0;
                    startFinishTimes.clear();
                    queuePeriods.clear();
                    queuePriorityFirst.clear();
                    queuePrioritySecond.clear();
                }

                catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        return true; // return true if everything is good
    }

    /**
     * Compute the schedule prefix from 0 to maxPeriod
     *
     * @return Does not return anything
     */
    public void schedulePrefixFirst() {
        counter = minPeriod; // set counter to minimum period
        queuePriorityFirst.clear(); // Remove all entries from priority queue
        while (timer < maxPeriod) { // Process Upto max period
            // call this method to check if any task instance is ready for
            // execution
            checkTaskAvailability();
            /**
             * If there is any instance available in the queue, Remove the head
             * of the priority queue, schedule it, add it to the schedule prefix
             * and increase the timer by 1 and decrease the counter.
             */

            if (queuePriorityFirst.size() > 0) {
                Schedule_Prefix temp = new Schedule_Prefix();
                temp.taskName = (queuePriorityFirst.remove().taskName);
                temp.startTime = timer;
                temp.finishTime = (timer + 1);
                startFinishTimes.add(temp);
                timer++;
                counter--;
            } else {
                /**
                 * Processor is idle since there is no instance ready for
                 * execution Just increase the timer and decrease the counter
                 */
				timer++;
				counter--;
			}
		}
	}

	/**
	 * Compute the schedule prefix from maxPeriod to time where the first
	 * instance of last task in the task set is scheduled to completion
	 *
	 * @return Does not return anything
	 */
	public void schedulePrefixSecond() {
		counter = minPeriod; // Reset the counter
		// Maximum possible time needed to process
		while (timer < (2 * maxPeriod)) {
			// call this method to check if any task is available
			checkTaskAvailability();
			Schedule_Prefix temp = new Schedule_Prefix();
			/**
			 * Check the head of the queue and schedule the task to its maximum
			 * execution time or the value of counter whichever is minimum.
			 * Increase the timer and decrease the counter accordingly.
			 */
			if (queuePrioritySecond.size() > 0) {
				Task temp1 = new Task();
				temp1 = queuePrioritySecond.remove();
				String taskName = temp1.taskName;
				Integer execution_time = temp1.maxExecTime;
				/* Task instance can be scheduled to completion */
				if (execution_time <= counter) {
					temp.taskName = taskName;
					// check if the task instance has already started but was
					// preempted
					if (times[temp1.index] > 0) {
						// Add this start time to temp
						temp.startTime = times[temp1.index];
						// reset the Current instance start time to 0
						times[temp1.index] = 0;
					} else
						/**
						 * Add the start time as the current time since there
						 * was no start time earlier
						 */
						temp.startTime = timer;
					// Add the finish time
```

```
                    temp.finishTime = (timer + execution_time);
                    // Add this to the scheduled prefix
                    startFinishTimes.add(temp);
                    timer = timer + execution_time;
                    counter = counter - execution_time;
                    /**
                     * Check if this is the last task in the taskset, if yes
                     * stop
                     */
                    if (taskset.get(taskset.size() - 1).equals(taskName)) {
                        worstTime = timer - 1;
                        lastLastDelay.add(worstTime);
                        break;
                    }
                }
                /**
                 * The task instance cannot be scheduled to completion because a
                 * higher priority task instance becomes available before this
                 * can be completed. So schedule it till counter
                 */
                else {
                    temp1.maxExecTime = (execution_time - counter);
                    queuePrioritySecond.add(temp1);
                    /**
                     * If there is no entry for the start time of this instance
                     */
                    if (times[temp1.index] == 0)
                        times[temp1.index] = timer;// make an entry
                    timer = timer + counter;
                    counter = 0; // Reset the counter
                }
            } else
                break; // If there is no task instance ready just stop
        }
    }

    /**
     * Check if there is any task instance ready for execution
     *
     * @return Does not return anything
     */
    public void checkTaskAvailability() {
        if (counter == 0 || counter == minPeriod) {
            /**
             * Check for the arrival of tasks starting from the minimum period
             * task until no task arrives and add them to the priority queue.
```

```
                         */
                        for (int i = 0; i < numberOfTasks; i++) {
                                Task task_ = new Task();
                                task_.period = periods.get(i).period;
                                if (timer % (task_.period) == 0) {
                                        task_.maxExecTime = periods.get(i).maxExecTime;
                                        task_.priority = periods.get(i).priority;
                                        task_.taskName = periods.get(i).taskName;
                                        task_.index = periods.get(i).index;
                                        task_.present = periods.get(i).present;
                                        if (timer < maxPeriod)
                                                queuePriorityFirst.add(task_);
                                        else
                                                queuePrioritySecond.add(task_);
                                } else
                                        break;
                        }
                        counter = minPeriod; // reset counter
                }
        }

        /**
         * Compute the last-last propagation delay by scanning the schedule prefix
         * once backwards
         *
         * @return Does not return anything
         */

        public void lastLastDelay() {
                display.append("Taskset: " + taskset + "\n\n"); // Display the task set
                int j = startFinishTimes.size(); // j is the index of last entry in
                                                 // Schedule Prefix
                for (int i=taskset.size()-1;i>=0;i--) {
                        String taskName = taskset.get(i); // retrieve each task
                        while (true) {
                                // scan till we reach the first entry
                                Schedule_Prefix temp = new Schedule_Prefix();
                                temp = startFinishTimes.get(--j);
                                String taskName1 = temp.taskName;
                                if (taskName.equals(taskName1)) { // if this is the task we want
                                        Integer finishtime = temp.finishTime;
                                        /**
                                         * check if the finish time is less than the start time of
                                         * next task in the sequence
                                         */
                                        if (finishtime <= worstTime) {
```

```
                                        // set it to the current finish time
                                        worstTime = finishtime;
                                        // add the start time to the lastLastDelay list
                                        lastLastDelay.add(temp.startTime);
                                        break;
                                }
                        }
                }
        }
        // display the information flow
        display.append("Last-Last Information Flow: " + lastLastDelay + "\n\n");
}

/**
 * Compute the first-first propagation delay by scanning the schedule prefix
 * once forwards
 *
 * @return Does not return anything
 */

public void firstFirstInfoDelay() {
        /**
         * declare the start time to be 1 plus the last entry of last to last
         * flow
         */

        Integer starttime = lastLastDelay.get(lastLastDelay.size() - 1) + 1;
        firstFirstDelay.add(starttime);
        int j = 0; // declare j to be the index of first entry in Schedule
                                // Prefix
        for (int i = 0;i < taskset.size();i++) {
                // scan until the last task in the set
                String taskName = taskset.get(i);
                while (true) {
                        Schedule_Prefix temp = new Schedule_Prefix();
                        temp = startFinishTimes.get(++j);
                        String taskName1 = temp.taskName;
                        // if this is the task we are looking for
                        if (taskName.equals(taskName1)) {
                                Integer startTime = temp.startTime;
                                /**
                                 * if start time is greater than the finish time of last
                                 * task instance
                                 */
                                if (startTime >= starttime) {
                                        // add this to the first to first flow list
                                        firstFirstDelay.add(temp.finishTime);
                                        // set start time to be the finish time of the task
                                        // instance
                                        starttime = temp.finishTime;
                                        break;
                                    }
                               }
                          }
                     }
                     // Display the Information flow
                     display.append("First-First Inormation Flow: " + firstFirstDelay + "\n\n");
                     display.append("Worst Case First-First Inforamtion Propagation Delay: "
                               + ((firstFirstDelay.get(firstFirstDelay.size()-1))-
                                        (firstFirstDelay.get(0)-1)) + "\n\n");
          }
}
```

Priority_Queue.java

```
import java.util.Comparator;

/**
 * Store the tasks in the decreasing order of priorities for the first part of
 * schedule prefix
 */
class PriorityFirst implements Comparator<Task> {
          @Override
          public int compare(Task x, Task y) {
                     if ((x.priority) < (y.priority))
                               return -1;
                     if ((x.priority) > (y.priority))
                               return 1;
                     if ((x.priority).equals((y.priority))) {
                               if (x.present)
                                        return -1;
                               if (y.present)
                                        return 1;
                               return 0;
                     }
                     return 0;
          }
}

/**
```

```
 * Store the tasks in the decreasing order of priorities for the second part of
 * schedule prefix
 */
class PrioritySecond implements Comparator<Task> {
	@Override
	public int compare(Task x, Task y) {
		if ((x.priority) < (y.priority))
			return -1;
		if ((x.priority) > (y.priority))
			return 1;
		if ((x.priority).equals((y.priority))) {
			if (x.present)
				return 1;
			if (y.present)
				return -1;
		}
		return 0;
	}
}

/**
 * This class stores the tasks in the increasing order of periods
 */
class Period implements Comparator<Task> {
	@Override
	public int compare(Task x, Task y) {
		if ((x.period) < (y.period))
			return -1;
		if ((x.period) > (y.period))
			return 1;
		return 0;
	}
}
```

Appendix B - Code for Algorithm 2

Delay.java

```
/** Program for computing First to First Information
 *  propagation Delay in a Periodic task sequence with
 *  increasing Priorities.

 *  Input : A text file containing the Task set indicated by Task names,
 *  followed by the Task sequence. Each Task has a Task name, Period, Maximum
 *  Execution time and Priority in this order. First "END" indicates the end of
 *  Task set and second "END" indicates the end of input. The programs assumes
 *  each input file is correct and does not check for errors. Every task
 *  sequence has a feasible schedule.

 *  Output : The First to First Information Flow is printed on the
 *  java Applet.

 * @author Vineet Tadakamalla
 */

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;
import java.util.List;
import java.awt.*;
import javax.swing.*;

/**
 * This class represents the task structure. All the entries are found from
 * input except for Index which is the just order of tasks.
 */
class Task {
	String taskName; // Name of the task
	Integer period; // Period of the task
	Integer maxExecTime; // Maximum execution time of the task
	Integer priority; // Priority of the task
	Integer index; // Index which is just the sequence in which they are entered
	Integer indexInSequence; // Index of the task in the task set, -1 if not
	// present
	Boolean present; // To indicate whether the task is in task set or not
}
```

```
/**
 * This class represents the schedule prefix constructed which helps in
 * calculating the first to first information flow.
 */
class Schedule_Prefix {
	String taskName; // Name of the task
	Integer startTime; // start time of each instance of a task
	Integer finishTime; // Finish time of each instance of a task
	Integer index; // Index of the task in the task set, -1 if not present
}

public class Delay extends JApplet {
	private static final long serialVersionUID = 1L;
	/**
	 * number of tasks in the sequence timer to keep track of schedule Maximum
	 * period in the sequence
	 */
	private int numberOfTasks = 0, timer = 0, maxPeriod = 0;
	/**
	 * task set for which the delay has to be calculated. Obtained from input
	 * file
	 */
	private List<String> taskset = new ArrayList<String>();
	/**
	 * used to temporarily store the task names present in the task set
	 */
	private Hashtable<String, Integer> tasksets = new Hashtable<String, Integer>();
	/**
	 * Declare a comparator instance for Priority queue First
	 */
	private Comparator<Task> comparatorFirst = new PriorityFirst();
	/**
	 * Declare a comparator instance for Priority queue Second
	 */
	private Comparator<Task> comparatorSecond = new PrioritySecond();
	/**
	 * Declare a comparator instance for Period queue
	 */
	private Comparator<Task> comparatorPeriods = new Period();
	/**
	 * Declare Priority queue to store tasks in first part of Schedule prefix
	 */
	private PriorityQueue<Task> queuePriorityFirst = new PriorityQueue<Task>(
			10, comparatorFirst);
	/**
	 * Declare Priority queue to store tasks in second part of Schedule prefix
 */
private PriorityQueue<Task> queuePrioritySecond = new PriorityQueue<Task>(
                    10, comparatorSecond);
/**
 * Declare Priority queue to sort periods and also store tasks
 */
private PriorityQueue<Task> queuePeriods = new PriorityQueue<Task>(10,
                    comparatorPeriods);
/**
 * used to temporarily store the task set
 */
private List<Task> periods = new ArrayList<Task>();
/**
 * List of possible start time values.
 */
private List<Integer> tvalues = new ArrayList<Integer>();
/**
 * The first to first information propagation delay
 */
private List<Integer> firstFirstDelay = new ArrayList<Integer>();
/**
 * Minimum period of the sequence Worst_time helps in calculating the
 * information flow. counter to keep track of each task instance execution
 * time. t1EndTime to know the first instance where t1 finishes after start
 * time.
 */
private Integer minPeriod = 0, counter = 0, t1EndTime = 0, worstFirstDelay=0,
worstLastDelay=0;
/**
 * stores start time of current task instance reference by task index.
 */
private Integer[] times = new Integer[10];
/**
 * The schedule prefix which contains the task name, start and finish times.
 */
private List<Schedule_Prefix> startFinishTimes = new ArrayList<Schedule_Prefix>();
/**
 * The schedule prefix which contains the task name, start and finish times
 * from Maxperiod to 2 * Maxperiod, which is cyclic.
 */
private List<Schedule_Prefix> startFinishTimesCyclic = new
ArrayList<Schedule_Prefix>();
private File ffile; // Input File
private JLabel title; // Title of the Output
private JTextArea display; // Display Area
private JPanel p; // Panel to hold the text area
```

```
/**
 * the main method for the program. Creates a Java Applet and a JFrame,
 *
 * @param args
 *          The command-line arguments. Not used in this program, but
 *          required by Java
 */

public static void main(String[] args) {
        JApplet theApplet = new Delay(); // Create an instance of Delay class.
        theApplet.init(); // Initialize the Applet
        JFrame window = new JFrame("Increasing Priorities"); // Title
        window.setContentPane(theApplet);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.pack();
        window.setVisible(true);
}

/**
 * Initializes Applet with dimensions 500*500
 *
 * @return Does not return anything.
 */
public void init() {
        setSize(500, 500);
}

/**
 * Constructor for the Class Delay used to set up the user Interface Sets
 * the Panel and display area.
 */
public Delay() {
        p = new JPanel(); // Create a Panel
        title = new JLabel(); // Create a Label
        Font f; // Declare a font
        // Create a Text Area with dimensions 20*30
        display = new JTextArea(20, 30);
        // Initialize the font to "Verdana", with Bold and size 14.
        f = new Font("Verdana", Font.BOLD, 14);
        display.setFont(f); // Set the font for the display area
        display.setEditable(false); // Make it non editable
        // Set where it should be displayed. Here it is South
        getContentPane().add(new JScrollPane(display), BorderLayout.SOUTH);
        p.add(title, 0); // Add title to the Panel
        title.setText("Information Propogation Delay.\n"); // Set title
```

```
		p.setFont(f); // Set Font for panel
		p.add(new Button("Open"), -1); // Add open button to open the input file
		p.add(new Button("Close")); // Add close button to close the applet
		// Set background color to white
		p.setBackground(new Color(255, 255, 255));
		add("North", p); // Add the panel to the north of the applet.

		/* End of UI */
	}

	/**
	 * action is the Implemented method of the Japlpet interface
	 *
	 * @param A
	 *            mouse click event which is either open or close.
	 * @return Boolean to indicate success
	 */
	public boolean action(Event event, Object arg) {
		if (arg.equals("Close")) {
			System.exit(0); // If the user clicks close button, exit the applet
		}
		if (arg.equals("Open")) { // If the user clicks open
			Frame parent = new Frame(); // Add a frame
			FileDialog fd = new FileDialog(parent, "Please choose a file:",
					FileDialog.LOAD); // Pop the file dialog box to open the
			// input file
			fd.setVisible(true);
			String selectedItem = fd.getFile();
			if (selectedItem == null) {
				// Handle File not found here.
			} else {
				ffile = new File(fd.getDirectory() + File.separator
						+ fd.getFile()); // accept the input file into ffile
				// Declare a buffered reader br to read the file
				BufferedReader br;
				String str = null; // Temporary variable to store each entry
				try {
					// Store the input file in br
					br = new BufferedReader(new FileReader(ffile));
					str = br.readLine(); // start reading line by line
					while (!str.contains("END")) { // Stop if line is "END"
						taskset.add(str); // Add the string into taskset list
						tasksets.put(str, taskset.size() - 1); // Also make an
						// entry in the
						// Hashtable
						str = br.readLine(); // Read next line
```

```
            }
            str = br.readLine(); // Continue Reading the file

        while (!str.contains("END")) { // Stop if the line is "END"
            Task task = new Task(); // create a Task instance task
            numberOfTasks++; // Increase the variable for each task
            task.taskName = str; // first entry is the task name
            str = br.readLine();
            task.period = new Integer(str); // second is task period
            str = br.readLine();
            // third is maximum execution time
            task.maxExecTime = new Integer(str);
            str = br.readLine();
            task.priority = new Integer(str); // fourth is priority
            task.index = numberOfTasks - 1; // Add the index
            /**
             * If Hashtable contains an entry then the task is
             * present in the task set.
             */
            if (tasksets.get(task.taskName) != null) {
                task.present = true;
                task.indexInSequence =tasksets.get(task.taskName);
                } else {
                    task.present = false;
                    task.indexInSequence = -1;
                }
                // Store the tasks in the priority Queue
                queuePriorityFirst.add(task);
                // Also store it in the Second priority Queue
                queuePeriods.add(task);
                str = br.readLine();
            }

            /* sorting the periods */
            while (queuePeriods.size() != 0) {
                periods.add(queuePeriods.remove());
            }
            Task per = new Task();
            per = periods.get(numberOfTasks - 1);
            // Set the maximum period which is the last entry of periods
            // list
            maxPeriod = new Integer(per.period);
            per = periods.get(0);
            // Set the minimum period which is the first entry of
            // periods list
            minPeriod = new Integer(per.period);
```

```
                    // Set all current task instance start time to zero
                    for (int i = 0; i < times.length; i++)
                        times[i] = 0;
                    // call the below methods
                    schedulePrefixFirst();
                    possibletValues();
                    startFinishTimes.clear();
                    System.out.println(tvalues);
                    /**
                     * For each pivot value compute the first first delay
                     */
                    for (int i = 0; i < tvalues.size(); i++) {
                        timer = tvalues.get(i); // retrieve pivot value and set
                        // it as timer
                        while ((timer + counter) % minPeriod != 0)
                            // set counter
                            counter++;
                        // call the below methods
                        schedulePrefixSecond();
                        firstFirstInfoDelay(tvalues.get(i));
                        LastLastInfoDelay(tvalues.get(i));
                        firstFirstDelay.clear();
                        startFinishTimes.clear();
                    }
            display.append("Worst case First-First Information Propagation Delay: "
                            + worstFirstDelay+ "\n\n");
            display.append("Worst Case Last-Last Information Propagation Delay: "
                            + worstLastDelay+ "\n\n");
                    taskset.clear();
                    tvalues.clear();
                    periods.clear();
                    numberOfTasks = 0;
                    timer = 0;
                    worstLastDelay=0;
                    worstFirstDelay=0;
                    startFinishTimesCyclic.clear();
                }

                catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        return true; // return true if everything is good
    }
```

```
/**
 * Compute the schedule prefix from 0 to maxPeriod
 *
 * @return Does not return anything
 */
public void schedulePrefixFirst() {
        counter = minPeriod; // set counter to minimum period
        queuePriorityFirst.clear(); // Remove all entries from priority queue
        while (timer < maxPeriod) { // Process Upto max period
                // call this method to check if any task instance is ready for
                // execution
                checkTaskAvailability(0);
                /**
                 * If there is any instance available in the queue, Remove the head
                 * of the priority queue, schedule it, add it to the schedule prefix
                 * and increase the timer by 1 and decrease the counter.
                 */

                if (queuePriorityFirst.size() > 0) {
                        Schedule_Prefix temp = new Schedule_Prefix();
                        temp.taskName = (queuePriorityFirst.remove().taskName);
                        temp.startTime = timer;
                        temp.finishTime = (timer + 1);
                        startFinishTimes.add(temp);
                        timer++;
                        counter--;
                } else {
                        /**
                         * Processor is idle since there is no instance ready for
                         * execution Just increase the timer and decrease the counter
                         */
                        timer++;
                        counter--;
                }
        }
}

/**
 * Compute the possible pivot times of the schedule
 *
 * @return does not return anything
 */
public void possibletValues() {
        for (int i = 0; i < startFinishTimes.size(); i++) {
                Schedule_Prefix temp = startFinishTimes.get(i);
                if ((temp.taskName).equals(taskset.get(0))) {
```

```
					tvalues.add(temp.startTime + 1);
				}
		}
}

/**
 * Compute the schedule prefix from pivot time to 2 * MaxPeriod.
 *
 * @return Does not return anything
 */
public void schedulePrefixSecond() {
		queuePrioritySecond.clear();
		// Maximum possible time needed to process
		while (timer < 2 * maxPeriod) {
			// call this method to check if any task is available
			checkTaskAvailability(1);
			Schedule_Prefix temp = new Schedule_Prefix();
			/**
			 * Check the head of the queue and schedule the task to its maximum
			 * execution time or the value of counter whichever is minimum.
			 * Increase the timer and decrease the counter accordingly.
			 */
			if (queuePrioritySecond.size() > 0) {
				Task temp1 = new Task();
				temp1 = queuePrioritySecond.remove();
				String taskName = temp1.taskName;
				Integer execution_time = temp1.maxExecTime;
				/* Task instance can be scheduled to completion */
				if (execution_time <= counter) {
					temp.taskName = taskName;
					// check if the task instance has already started but was
					// preempted
					if (times[temp1.index] > 0) {
						// Add this start time to temp
						temp.startTime = times[temp1.index];
						// reset the Current instance start time to 0
						times[temp1.index] = 0;
					} else
						/**
						 * Add the start time as the current time since there
						 * was no start time earlier
						 */
						temp.startTime = timer;
					// Add the finish time
					temp.finishTime = (timer + execution_time);
					temp.index = temp1.indexInSequence;
```

```
                    // Add this to the scheduled prefix
                    startFinishTimes.add(temp);
                    if (temp.startTime >= maxPeriod)
                        startFinishTimesCyclic.add(temp);
                    timer = timer + execution_time;
                    counter = counter - execution_time;
                }
                /**
                 * The task instance cannot be scheduled to completion because a
                 * higher priority task instance becomes available before this
                 * can be completed. So schedule it till counter
                 */
                else {
                    temp1.maxExecTime = (execution_time - counter);
                    queuePrioritySecond.add(temp1);
                    /**
                     * If there is no entry for the start time of this instance
                     */
                    if (times[temp1.index] == 0)
                        times[temp1.index] = timer;// make an entry
                    timer = timer + counter;
                    counter = 0; // Reset the counter
                }
            } else {
                timer++; // If there is no task instance ready just stop
                counter--;
            }

        }
}

/**
 * Check if there is any task instance ready for execution
 *
 * @return Does not return anything
 */
public void checkTaskAvailability(int num) {
        if ((counter == 0) || (counter == minPeriod)) {
            /**
             * Check for the arrival of tasks starting from the minimum period
             * task until no task arrives and add them to the priority queue.
             */
            for (int i = 0; i < numberOfTasks; i++) {
                Task task_ = new Task();
                task_.period = periods.get(i).period;
                if (timer % (task_.period) == 0) {
```

```
                                        task_.maxExecTime = periods.get(i).maxExecTime;
                                        task_.priority = periods.get(i).priority;
                                        task_.taskName = periods.get(i).taskName;
                                        task_.index = periods.get(i).index;
                                        task_.indexInSequence = periods.get(i).indexInSequence;
                                        task_.present = periods.get(i).present;
                                        if (num == 0)
                                                queuePriorityFirst.add(task_);
                                        else
                                                queuePrioritySecond.add(task_);
                                } else
                                        break;
                        }
                        counter = minPeriod; // reset counter
                }
        }

        /**
         * Constructs an array that for each finish time no later that 2*Maxperiod
         * of a task T(i), 1<=i<=(taskset.size), the finish time of the next
         * instance of T(i+1).
         *
         * @return Returns the computed array
         */
        public Integer[] finishTimesArray() {
                /**
                 * Declare a list of linked lists to hold the temporary finish time
                 * values for each task in the task set.
                 */
                List<List<Integer>> finishtimes = new ArrayList<List<Integer>>();
                /**
                 * Declare the array to store the finish times.
                 */
                Integer[] finishtimes1 = new Integer[(2 * maxPeriod) + 1];
                Boolean b = true;
                /**
                 * Store finish time as 0 for each task in the temporary list.
                 */
                for (int i = 0; i < taskset.size() - 1; i++) {
                        List<Integer> temp = new ArrayList<Integer>();
                        finishtimes.add(temp);
                }

                /**
                 * Scan through the schedule prefix once and for each finish time record
                 * its next task instance finish time.
```

```
 */
for (int i = 0; i < startFinishTimes.size(); i++) {
        Schedule_Prefix temp = startFinishTimes.get(i);
        Integer currentFinishTime = temp.finishTime;
        Integer index = temp.index;
        if ((index == 0) && b) {
                t1EndTime = currentFinishTime;
                b = false;
        }
        if (index < (taskset.size() - 1) & index > -1)
                finishtimes.get(index).add(currentFinishTime);
        if (index > 0) {
                List<Integer> temp1 = finishtimes.get(index - 1);
                while (temp1.size() > 0) {
                        if (temp1.get(0) < currentFinishTime)
                        finishtimes1[temp1.remove(0)] = currentFinishTime;
                        else
                                break;
                }
        }
}
/**
 * Declare a variable to keep track of how many tasks remain whose
 * instances finish times does not have a value.
 */
int count = 0;
/**
 * Scan through the prefix again, this time by adding MaxPeriod to the
 * finish times since it is a cyclic schedule and record the missing
 * values.
 */
for (int j = 0; j < startFinishTimesCyclic.size(); j++) {
        if (count == finishtimes.size())
                break;
        Schedule_Prefix temp1 = startFinishTimesCyclic.get(j);
        if (temp1.index > 0) {
                List<Integer> temp = finishtimes.get(temp1.index - 1);
                while (temp.size() > 0) {
                finishtimes1[temp.remove(0)] = temp1.finishTime + maxPeriod;
                        if (temp.size() == 0)
                                count++;
                }
        }
}
return finishtimes1;
```

```
}

/**
 * Compute the first-first propagation delay by going through the finish
 * times array.
 *
 * @return Does not return anything
 */
public void firstFirstInfoDelay(Integer t0value) {
	firstFirstDelay.add(t0value);
	Integer finishtimes[] = finishTimesArray();
	firstFirstDelay.add(t1EndTime);
	Integer currentIndex = t1EndTime;
	for (int i = 1; i < taskset.size(); i++) {
		if (currentIndex <= maxPeriod) {
			firstFirstDelay.add(finishtimes[currentIndex]);
			currentIndex = finishtimes[currentIndex];
		} else {
			Integer tempIndex = 0;
			if (currentIndex % maxPeriod == 0)
				tempIndex = 2 * maxPeriod;
			else
				tempIndex = (currentIndex % maxPeriod) + maxPeriod;
			Integer tempFinishIndex = finishtimes[tempIndex];
			int x = 0;
			if (currentIndex > tempFinishIndex)
			x = (int) Math
				.floor(((currentIndex - tempFinishIndex) / maxPeriod) + 1);
			tempFinishIndex += x * maxPeriod;
			firstFirstDelay.add(tempFinishIndex);
			currentIndex = tempFinishIndex;
		}
	}
	display.append("First-First Information Flow: " + firstFirstDelay
			+ "\n\n");
	display.append("First-First Information Propagation Delay" +
			": " + (currentIndex-(t0value-1))+ "\n\n");
	if(worstFirstDelay<(currentIndex-(t0value-1)))
		worstFirstDelay=(currentIndex-(t0value-1));
}

/**
 * Compute the last-last propagation delay by computing the last task
 * instance finish time less than the last entry in first-first info flow
 *
 * @return Does not return anything
```

```
	 */
	public void LastLastInfoDelay(Integer t0value) {
		int lastFinishTime = 0, tnFinishTime, temptnFinishTime;
		List<Integer> lastInstanceFinishTimes = new ArrayList<Integer>();
		for (int i = startFinishTimesCyclic.size() - 1; i >= 0; i--) {
			if (startFinishTimesCyclic.get(i).index == (taskset.size() - 1))
				lastInstanceFinishTimes
						.add(startFinishTimesCyclic.get(i).finishTime);
		}
		tnFinishTime = firstFirstDelay.get(firstFirstDelay.size() - 1);
		if (tnFinishTime % maxPeriod == 0)
			temptnFinishTime = 2 * maxPeriod;
		else
			temptnFinishTime = (tnFinishTime % maxPeriod) + maxPeriod;
		for(int j=0;j < lastInstanceFinishTimes.size();j++) {
			if (lastInstanceFinishTimes.get(j) < temptnFinishTime) {
				lastFinishTime = lastInstanceFinishTimes.get(j);
				break;
			}
		}
		if (lastFinishTime == 0)
			lastFinishTime = temptnFinishTime;
		int x = (int) Math
				.ceil(((tnFinishTime - lastFinishTime) / maxPeriod) - 1);
		lastFinishTime += x * maxPeriod;
		if(worstLastDelay<(lastFinishTime - (t0value - 1)))
			worstLastDelay=(lastFinishTime - (t0value - 1));
	}
}
```

Priority_Queue.java

```
import java.util.Comparator;

/**
 * Store the tasks in the decreasing order of priorities for the first part of
 * schedule prefix
 */
class PriorityFirst implements Comparator<Task> {
	@Override
	public int compare(Task x, Task y) {
		if ((x.priority) < (y.priority))
			return -1;
		if ((x.priority) > (y.priority))
			return 1;
		if ((x.priority).equals((y.priority))) {
			if (x.present)
				return -1;
			if (y.present)
				return 1;
			return 0;
		}
		return 0;
	}
}

/**
 * Store the tasks in the decreasing order of priorities for the second part of
 * schedule prefix
 */
class PrioritySecond implements Comparator<Task> {
	@Override
	public int compare(Task x, Task y) {
		if ((x.priority) < (y.priority))
			return -1;
		if ((x.priority) > (y.priority))
			return 1;
		if ((x.priority).equals((y.priority))) {
			if (x.present)
				return 1;
			if (y.present)
				return -1;
		}
		return 0;
	}
}

/**
 * This class stores the tasks in the increasing order of periods
 */
class Period implements Comparator<Task> {
	@Override
	public int compare(Task x, Task y) {
		if ((x.period) < (y.period))
			return -1;
		if ((x.period) > (y.period))
			return 1;
		return 0;
	}
}
```