

CS297 Report

ASH - A Scheduler for HOAs

Qian Li

(sabrinaqq2002@yahoo.com)

Advisor: Dr. Chris Pollett

December 2004

Table of Contents

1. Introduction.....	4
2. Define HOAs scheduling problem.....	5
3. Deadline consideration.....	5
3.1 Greedy Task-scheduling Algorithm.....	5
3.2 Greedy-task Scheduling method.....	6
4. Average flow time and average stretch consideration.....	8
4.1 Semi-clairvoyant R algorithm (Sc-R), FIFO and SPTF.....	8
4.2 Strategy of Sc-R, FIFO and SPTF algorithms.....	8
4.2.1 Semi-clairvoyant R algorithm method and implementation.....	8
4.2.2 First In First Out method and improvements.....	9
4.2.3 Shortest Process Time First Method and flaws.....	10
4.3 Scheduling results and comparisons.....	10
5. Conclusion and Future works.....	11
References.....	13
Appendix A: Source codes for Deliverable One.....	14
Appendix B: Source code for Greedy task-scheduling Algorithm.....	16
Appendix C: Source code for Sc-R, FIFO and SPTF algorithms.....	21

List of Tables

Table 1: Schedule result for no job late for their deadlines	7
Table 2: Schedule results for some jobs late for deadlines and got penalties.....	7
Table 3: One example of the combinational scheduler scheduling results.....	10
Table 4: Enumeration of some results on different number of jobs	11

1. Introduction

Job scheduling is simply defined as optimal allocation of limited resources to jobs over time. An optimal allocation will result in an efficient scheduling of a set of activities on a set of resources. This problem is a basic problem either in computer science research or in real application research. This is also a difficult problem, and hence, most of the research in this field has incorporated with some potentially inapplicable assumptions. The goal of this project will be to develop an applicable job scheduler, which is capable of aiding the homeowners' associations (HOAs) in achieving an optimal scheduling results, not only to have customers' requests fulfilled in a timely manner, but also to be given immediate feedback on a timeframe in which activities will be made.

Different scheduling algorithms are concerned on different specific fields. In simple words, the scheduling algorithms are to assign the start and end times to a set of jobs that are in waiting queue to be scheduled, subject to certain constraints to optimize certain objective functions. Those constraints are typically either time constraints (before the deadline) or resource constraints (some jobs compete for the same resource to run). Therefore, to develop an optimal scheduling algorithm especially for HOAs, we must have the well-defined model that specify the all of their particular considerations, constrains and requirements. In my project, the first step is to define the HOAs scheduling problem refer to the standard notation defined as $\alpha|\beta|\gamma$ [GKLL79].

Based on the defined HOAs model, the first scheduling algorithm considered in this project is Greedy task-scheduling algorithm [CLR90]. This solution can optimally schedule those unit-time jobs on a single processor proven by the combinatorial structure known matroid that is useful in determining when the greedy method yields optimal solutions.

The other scheduling algorithms considered in my project are Semi-clairvoyant R algorithm [BMLP04], First In First Out (FIFO) and Shortest Processing Time First (SPTF) scheduling algorithms. These algorithms are differentiated in their methods, and their scheduling results. Each of them has their own pros and cons, so it is necessary to come up with certain new scheduling algorithms that can efficiently trade off between the cons and pros; as a result, they are suitable for different requirements and configurable for different needs. This is also the reason why this project raises to the Master students' projects level. The algorithm is achievable mostly depends on substantial research and mathematical maturity to understand and compare the algorithms involved, not just implement them.

There are three deliverables and some research slides turned in. Because this project's final goal is to build up a configurable HOA web-site job-scheduler, it will require some fundamental knowledge of dynamic web design and database management. The deliverable one is subject to achieving the basic skills in building up web application by using MySQL, Apache and PHP. The deliverable two focus on improving Greedy task-scheduling algorithm, and the deliverable three subject to testing variants in Semi-clairvoyant R algorithm, FIFO and SPTF algorithms.

2. Define HOAs scheduling problem

In general, scheduling problems can be denoted as $\alpha|\beta|\gamma$, where α stands for the machine environment, β stands for the various side constraints and characteristics, and γ stands for an optimality criterion. For the HOAs situation, the α defined as “1” (one machine), such that there is only one job can be processes at a time.

At the beginning of each scheduling problem, we have the set J of jobs, which numbered from 1 to n . Each job J_i has a release time r_i , a processing time p_i , a deadline d_i and weight w_i , then the job J_i can be denoted as tuple (r_i, p_i, d_i, w_i) . If each job must be finished once it starts, it is a non-preemptive scheduling environment; otherwise, a job in process can be interrupted and continued at a later point of time, it is a preemptive environment. The HOAs' machine environment belongs to non-preemptive environment.

Then the focus turns to β , side constraints and characteristics, which can specify the machine environment. There are lots of possible side constraints and characteristics, such as logical dependent or independent among jobs, and the jobs arrival internals. As usual, the job release time is always before its actual processing time and finish time. In other words, this is the order constraint. No matter what side constraints and characteristics taken into consideration, the final goal of scheduling is to produce a good schedule for certain objects. In HOAs' problem, matching deadline, timely feedback, certain amount of budget limit and so on are all to be considered during the scheduling.

However, there are no consistent rules to define what kind of scheduling algorithm is good because it is application dependable. Sometime the goal is to minimize the completion time, and sometime the goal is to maximize the total weights gained from matching the deadline of each job. Formally, the goal is the optimality criterion γ , the purpose of scheduling algorithm is to construct the algorithm optimize this criterion. In HOAs, γ is to minimize the penalties got from delaying after the assigned deadlines, and maximize the average flow time and stretch based on scheduled jobs.

3. Deadline consideration

Matching deadlines and minimizing the penalties caused by postponing the due jobs is a critical issue in HOA's scheduling problem. The implemented Greedy task-scheduling algorithm intentionally achieves this goal.

3.1 Greedy Task-scheduling Algorithm

The greedy algorithm always makes the choice that looks the best at that time. That is, it tries to make a locally optimal choice that could leads to the final global optimal solution in the hope. However, the greedy algorithms rarely find the globally optimal solution consistently as expected, since they usually don't operate exhaustively on all the data. The dynamic programming algorithms (or backtracking algorithms) always produce the optimal global solution, but its performance is unfavorable, especially for a large number of jobs to be scheduled Therefore, Greedy task-scheduling algorithm outstands because it

is useful in real applications and it is quick to think up and often come up with good approximations to the optimum. Of course, the main benefit of greedy algorithms lies in both their conceptual simplicity and their computational efficiency. If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the actual method of choice. The combinatorial structures, known as matroids, [CLR90] are useful in determining when greedy method yields optimal solutions.

3.2 Greedy-task Scheduling method

The job set J is a set of unit-time tasks with deadlines, and each J_i ($i? 1,2,...n$) is independent from the other jobs, the (J, J_i) system is a matroid [CLR90]. Also, $M=(J, J_i)$ is a weighted matroid with weight function w , so the Greedy-task scheduling (M, w) returns an optimal subset. By this theorem, this algorithm can be used to find a maximum weight independent job set. This method is an efficient algorithm for scheduling unit-time tasks with deadline and penalties for a single processor $\alpha=1$. Its running time is $O(n^2)$ no matter what kind of sorting algorithm applied in the process of scheduling.

The strategy of the Greedy task-scheduling algorithm:

Initiate n time slots empty

Sort the n waiting for scheduling jobs by their weights

For (time=1; time<=n; time++)

Schedule the highest weighted job J_i ($i? 1,2,...n$) in the currently waiting queue

If (J_i deadline time slot is empty)

then assign J_i at that time slot

else if (before J_i deadline time slots available)

then assign the latest available time slot in deadline to J_i

Otherwise pick the most end time slot to J_i .

3.3 Test results and analysis

For testing purpose, the jobs are generated randomly by program. And each job has its own description parameters, such as the job number, weights, processing time and deadline. At the beginning the jobs generated in the waiting queue, and the sorter-class will sort all the active jobs depends on their weights. And the sorted jobs will be in decreased weight order. And the Greedy scheduler schedule them according to the above describe strategy.

The following tables display two of the test results:

Jobno	Weight	Deadline		Time	Job no	The schedule results:
1	59	18		1	3	<i>The number of late tasks = 0</i>
2	43	6		2	18	
3	23	1		3	9	
4	29	15		4	8	
5	71	16		5	19	
6	51	16		6	2	
7	99	16		7	15	


Jobno	Weight	Deadline	Greedy Task scheduling 	Time	Job no	The schedule results: <i>The total penalties = 0</i>
8	14	6		8	12	
9	11	16		9	11	
10	30	13		10	4	
11	21	15		11	10	
12	75	8		12	6	
13	97	17		13	14	
14	55	16		14	16	
15	38	7		15	5	
16	56	16		16	7	
17	29	19		17	13	
18	75	2		18	1	
19	39	6		19	17	

Table 1: Schedule result for no job late for their deadlines


Jobno	Weight	Deadline	Greedy Task scheduling 	Time	Job no	The schedule results: <i>The number of late tasks = 4</i> <i>The total penalties = 94 (8+12+37+37)</i>
1	46	7		1	3	
2	50	23		2	23	
3	44	1		3	22	
4	84	9		4	14	
5	29	15		5	16	
6	25	12		6	13	
7	94	13		7	1	
8	8	9		8	9	
9	65	9		9	4	
10	80	10		10	10	
11	37	4		11	20	
12	69	19		12	6	
13	43	11		13	7	
14	40	4		14	8	
15	46	23		15	5	
16	15	12		16	17	
17	81	16		17	18	
18	12	9		18	11	
19	37	22		19	12	
20	49	11		20	21	
21	37	1		21	19	
22	66	3		22	15	
23	97	2	23	2		

Table 2: Schedule results for some jobs late for deadlines and got penalties

(Note: the red highlighted the job numbers are the jobs that are late for their deadlines and have to pay their own weights as penalties.)

The scheduling results can show us the optimal scheduling in respect to minimizing the penalties and it can really benefit the HOAs scheduling problem in case of the budget consideration. However, it has its limitations, such as the unit processing time, non-release time. It assumes all the jobs can be done in unit-time; actually, each job should have its particular processing time. And the coming jobs will not arrive at the same time, each of them has its specific release time. To schedule the waiting jobs can refer to the

online algorithm. To improve this greedy algorithm by getting rid of these pointed out limitations, I will apply the online algorithm into the on going scheduling with respect to W_i/P_i for CS298.

4. Average flow time and average stretch consideration

Besides the deadlines and penalties, both the jobs' average flow time and average stretch are important measurements for the job schedulers. For a set of jobs J_i which $i = 1, 2, \dots, n$, each of them has release time r_i (non-negative integer), and processing time p_i (non-negative integer). After scheduling, each job has its own completion time c_i , and the average flow time is $1/n \sum_{i=1}^n (c_i - r_i + I)$ and the average stretch is equal to $1/n \sum_{i=1}^n \{(c_i - r_i + I) / p_i\}$. During the scheduling, those active jobs can be divided into two categories: *partial jobs and total jobs* [BLMP04]. Partial jobs have already been executed in the past by the scheduler, while the total jobs have never been executed by the scheduler. All above parameters are calculated and compared in this project.

4.1 Semi-clairvoyant R algorithm (Sc-R), FIFO and SPTF

HOAs job scheduler is an online job scheduler. Study on similar example - web servers, we found that currently most of the web servers apply FIFO scheduling strategy instead of the SPTF scheduling strategy even though SPTF can obtain minimized average flow time. Why? The most important reasons are: a) SPTF is 2-competitive with respect to average stretch [SRRJ97] and b) web servers use FIFO is the fear of starvation. However, here is another strong argument made for unfounded fear of starvation, in other words, the web servers should take SPTF as another scheduling choice.

It was shown that the semi-clairvoyant R algorithm is $O(1)$ -competitive with respect to average flow time on a single machine, and $O(1)$ -competitive with respect to average stretch on a single machine, but it can not be simultaneously $O(1)$ -competitive with respect to both average flow time and average stretch [BLMP04].

4.2 Strategy of Sc-R, FIFO and SPTF algorithms

4.2.1 Semi-clairvoyant R algorithm method and implementation

The Sc-R algorithm can guarantee that at all times each queue of jobs has at most one partial job; this fact is also useful for the algorithm analysis. The enqueueer will put the released jobs into different queues. The queue number k from 1 to n , the job belongs to queue k ? ($1, 2, \dots, n$), it means that $p_i \in [2^k, 2^{k+1})$. The scheduling from the lowest numbered queue:

- o If at any certain time, all the jobs in the queue k , then pick the partial job to execute if there exist one, otherwise, pick any one of the total jobs to run;
- o If at any certain time, there are two lowest numbered queue m, n ($m < n$) and both of them have jobs in them:
 - o If m has exactly one total job J_i , and n has exactly one partial job J_j , then run the partial job J_j ;

- Under any other situations, run the partial jobs in m if there has one, otherwise run a total job in m .

And this method implemented in my code as following:

```

Class Rscheduler extends Scheduler{
    public Rscheduler(){
        int i;
        jobsClass = new Queue [4];
        for(i=0; i<4; i++)
        {
            jobsClass[i] = new Queue();
        }
    }
    public Job run(int time)
    {
        ...
        for(i=0; i<jobsClass.length; i++)
        {
            count = jobsClass[i].getItemNum();
            if( count == 0)
            {
                continue;
            }
            else
            {
                break;
            }
        }
        ... return (firstJob);
    } ...
}

```

These jobs executed in preemptive environment. In the next semester, it will be modified as non-preemptive scheduling according to the real application requirement in CS298. Also, for locating the jobs into the queue will not only take the first request process time into account, also the deadline and other description parameters such as cost and weight per processing time will put into consideration. Because the current enqueueer only locates the jobs once, at the beginning, actually the jobs coming over time one by one, it is necessary to relocate the jobs into the different queues, then scheduling can be much more efficient than the original one.

4.2.2 First In First Out method and improvements

The FIFO scheduling strategy assigns priority to the jobs in the order in which they request for processing. The priority of each job is computed by the enqueueer by time stamping all incoming jobs. Actually, this project did not check and apply the time-stamping technique. It took use of the automatically generated increased Job ID, another

parameter to describe each job's coming order, to judge its priority in the queue. Every time the new jobs come, the enqueuer will add the job to the tail of the queue with a unique Job ID, and the dispatcher will remove the jobs from the head of the queue.

This is non-preemptive scheduling environment. The most benefit of this algorithm is easy to implement, no fear about starvation at all. Although this is not totally capable of scheduling the HOAs problem, it can be combined with Sc-R algorithm and apply into the highest number queue that holds bunch of jobs with long processing time requests. This idea inspired by multiple-level queue scheduling algorithm in operating system. On the other hand, this is really good reference for comparing the scheduling algorithms.

4.2.3 Shortest Process Time First Method and flaws

And the SPTF scheduling algorithm always chooses the job that requiring minimum processing time to run first. It surely guarantee that the scheduler can obtain the best average flow time, however it might penalize jobs with long service time requests. If the ready queue is saturated, then the jobs with long service time request ten to be left in the ready queue while those jobs with short process time request receive service. In some extreme case, where the schedule system has little idle time, jobs with large processing time request will never be served. As a result, the total starvation of those jobs that have large processing time requests may be a serious liability of the scheduling algorithm.

4.3 Scheduling results and comparisons

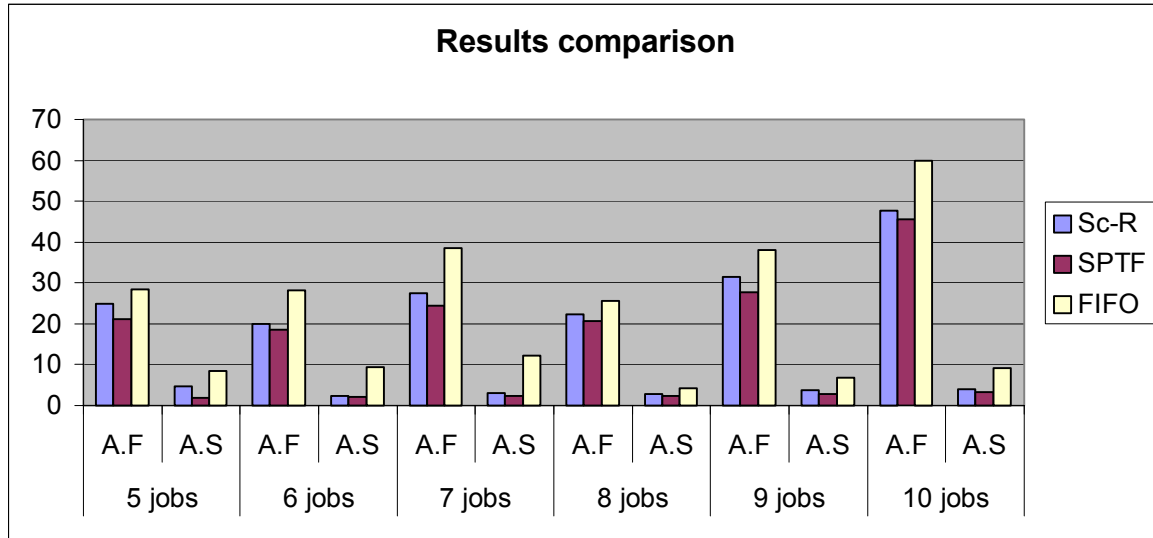
In this project, I build up a combinational scheduler that can schedule the same randomly generated jobs by applying different scheduling algorithms; they are Sc-R algorithm, FIFO and SPTF respectively. The following are listed two of the test cases:

ID	r_i	p_i	Sc-R algorithm			SPTF algorithm			FIFO algorithm		
			s_i	c_i	f_i	s_i	c_i	f_i	s_i	c_i	f_i
1	1	4	1	7	7	1	4	4	1	4	4
2	2	14	16	29	28	69	82	81	5	18	17
3	4	1	4	4	1	5	5	2	19	19	16
4	4	2	5	6	3	6	7	4	20	21	18
5	5	3	8	10	6	8	10	6	22	24	20
6	5	13	30	42	38	43	55	51	25	37	33
7	6	9	43	51	46	16	24	19	38	46	41
8	7	5	11	15	9	11	15	9	47	51	45
9	8	13	52	64	57	56	68	61	52	64	57
10	8	9	65	73	66	25	33	26	65	73	66
11	9	9	74	82	74	34	42	34	74	82	74
Average Flow time			30.45			27.0			35.55		
Average Stretch			3.46			2.9			6.36		

Table 3: One example of the combinational scheduler scheduling results

No. Alg.	5 jobs		6 jobs		7 jobs		8 jobs		9 jobs		10 jobs	
	A.F	A.S	A.F	A.S	A.F	A.S	A.F	A.S	A.F	A.S	A.F	A.S
Sc-R	25.0	4.65	20.0	2.38	27.43	3.15	22.25	2.82	31.4	3.74	47.7	3.877
SPTF	21.2	1.92	18.67	2.11	24.43	2.33	20.75	2.33	27.78	2.89	45.5	3.36
FIFO	28.4	8.41	28.16	9.49	38.42	12.25	25.63	4.31	38.0	6.83	59.9	9.13

Table 4: Enumeration of some results on different number of jobs



*Table 4 –1: Visualized results comparison of table 4
(Note: A.F stands for Average Flow time, A.S stands for Average Stretch)*

The above listed table just shown the random results of different number of jobs' scheduling, and the calculation on average flow time and average stretch. This is not a statistics table, and here is meaningless to make statistics analysis. Because the scheduling performance mostly depends on the randomly generated jobs, it is no way to make real statistical comparison. However, this table and figure can present the performance of different algorithms. Obviously, the SPTF scheduler outstands from the other two with respect to average flow time and average stretch. But this algorithm still has some fatal flaw, such as long process time request job gets into starvation.

5. Conclusion and Future works

These researched and implemented scheduling algorithms can partly satisfy the really application in HOAs scheduling problem, such as Greedy task-scheduling focus on meeting deadlines and minimize the penalties, Semi-clairvoyant R algorithm focus on achieving better average flow time, FIFO focus on obtaining the easy implementation and fairness among jobs, SPTF just focus on reaching the shortest waiting time in the queue and average turnaround time. Obviously, no one of them can really come up the required solution for the HOAs scheduling problem. To get a real way out, it can try to apply those

algorithm methods and combine them with kind of trade off to come up a combinational HOAs scheduling algorithm.

In the next semester, I will design an ASH scheduling algorithm that can use the Semi-clairvoyant R algorithm to assure there is only one partial job during the process of scheduling, and apply Greedy task-scheduling algorithm, FIFO and SPTF into different queues. And this try will aim at the actual Rules and Regulations of HOAs. Moreover, there is an important issue that is not taken into consideration throughout the semester — budget and cost per job. In other words, there still has some job to be done about the detail description of those jobs. How to name it, how to count it, and how to deploy it in the ASH algorithm and ect will have a elaborate problem description done at the beginning of CS298.

Another limitation for the tested algorithms is the assumptions: Greedy assumes each job can be done in unit time, FIFO assumes each job's ID number is its real timestamp, Semi-clairvoyant R algorithm assumes there always has enough budget to make the job done, and SPTF assumes that each job has no deadline limitation. So, in the next semester, all the assumptions should be taken away, and develop an applicable ASH for jobs' scheduling.

In HOAs, ASH receives and processes the coming jobs in partial amounts. In serving each request the ASH has a choice of several alternatives, each with a particular cost. The alternative chosen at a step may influence the costs of alternatives on future requests. This characteristic exactly matches the property of the online algorithms. Therefore, more study and research about online algorithms will be done in CS298.

Based on my literature research in the field of probability, statistics and queueing theory, I found that the method of the imbedded Markov chain is the traditional approach to M/M/1 in much of the literature; also it makes analysis of such system straightforward [MR95]. The reason is the memorylessness property of Markov chain – the future behavior of a Markov chain depends on its current state, and not on how it arrived at the present state. This means that the transition probabilities depend only on the current state. And this property fits the HOAs scheduling problem well, I will try to embed the Markov chains into my CS298 project in order to develop an applicable, predictable and configurable ASH.

References

- [AI90] A. Allen. Probability, Statistics, and Queueing Theory with Computer Science Applications. Academic Press, inc. 1990.
- [AL99] Online algorithm, Susanne Albers, Stefano Leonardi, Volume 31, Issue 3es (September 1999) Article No. 4, Year of Publication: 1999 ISSN:0360-0300
- [ATCH] Algorithms and Theory of Computation Handbook, page 10-17, Copyright © 1999 by CRC Press LLC. Appearing in the Dictionary of Computer Science, Engineering and Technology, Copyright © 2000 CRC Press LLC.
- [BLMP04] B. Luca, L. Stefano, M. Alberto, P. Kirk, Semi-clairvoyant scheduling. Theoretical Computer Science 324(2004) 325 –335.
- [BNR02] (Incremental) Priority Algorithms, Allan Borodin, Morten N. Nielsen, Charles Rackoff, Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms Pages: 752 – 761, Year of Publication: 2002, ISBN:0-89871-513-X
- [CLR90] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Introduction to algorithms. MIT Press. 1990.
- [GKLL79] R.L.Graham, A.H.G. Rinnooy Kan., E.L. Lawler, J. K. Lenstra. Optimization and approximation in deterministic sequencing and scheduling: survey. Annals of Discrete Mathematics, 5:287-326, 1979.
- [KSW] David Karger, Massachusetts Institute of Technology, Cliff Stein, Dartmouth college, Joel Wein, Polytechnic University. Scheduling Algorithms. Lecture notes.
- [MR95] M. Rajeev, R. Prabhakar, Randomized Algorithms, Cambridge University Press, 1995
- [NG02] Nutt G. Operating Systems – A Modern Perspective 2nd Edition LAB UPDATE
- [RP04] Mechanism design for online real-time scheduling, Ryan Porter, Pages: 61 - 70 Year of Publication: 2004 ISBN: 1-58113-711-0
- [SRRJ97] S. Muthukrishnan, R. Rajaraman, R. Shaheen, J. Gehrke, Online scheduling to minimize average stretch, IEEE Symp. Foundations of Computer Science, 1997, pp 433 - 442

Appendix A: Source codes for Deliverable One

<!--Description: This PHP program can be used to store job site web pages, and it can also perform a search engine based on the users' query to grab the results from the MySQL database.-- >

```
<?php

// set some short variable names
$host="localhost";
$user="guest";
$password="qq99";
$searchterm=$_POST['searchterm'];

$searchterm=trim($searchterm);

if(!$searchterm)
{
    echo 'You have not entered the correct search terms. Please go
back and try again.';
    exit();
}

$searchterm=addslashes($searchterm);

@$db_link=mysql_connect($host,$user,$password);

if(!$db_link)
{
    echo 'It could not connect to the assigned database. Please go
back and try again.';
    exit();
}

mysql_select_db('job_site');
$query="select title,url,description from job_list where title like
'${$searchterm}%'";
$result=mysql_query($query);

if($result == 0)
{
    echo "<b>Error ".mysql_errno().": ".mysql_error().
"</b>";
}
elseif (@mysql_num_rows($result) == 0)
{
    echo("<b>Query completed. No results returned.
</b><br>");
}
else
{
    $numrow_result=mysql_num_rows($result);
    echo '<p><b>Number of similar records found:
'."{$numrow_result}".$</b></p>';
}

for ($i=0; $i<$numrow_result;$i++)
{
    $row=mysql_fetch_array($result);
```


Appendix B: Source code for Greedy task-scheduling Algorithm

```
/******  
* Project:      Testing the variants in scheduling algorithms  
* File:        Greedy.java  
* Description:  A random number of jobs are generated with random weights. Jobs are  
*              sorted according to job weights, and then scheduled per Greedy algorithm.  
*  
*****/  
class Greedy {  
  
    Job[] jobList;  
    Job[] sortedJobList;  
    int[] schedule;  
  
    public Greedy()  
    /*-----  
    PURPOSE: constructor, creat jobs and add to job list  
    RECEIVES: nothing  
    RETURNS: nothing  
    -----*/  
    {  
  
        int jobCounts;  
        int wipi, processsTime, deadLine;  
  
        jobCounts = getRandom(30);  
  
        jobList = new Job[jobCounts];  
        sortedJobList = new Job[jobCounts];  
        schedule = new int[jobCounts];  
  
        for(int i=0; i<jobCounts; i++)  
        {  
            wipi = getRandom(100);  
            //processsTime = getRandom(10);  
            processsTime = 1;  
            deadLine = getRandom(jobCounts); // ??? algorithm limit: dead line  
            can not over job count.  
            //jobList[i] = new Job(i+1, wipi, processsTime, deadLine);  
            jobList[i] = new Job(i+1, wipi, processsTime, deadLine);  
        }  
  
    }  
  
    /*-----
```



```

PURPOSE: display all jobs in job list.
RECEIVES: nothing
RETURNS: nothing
-----*/
private void displayData()
{
    System.out.println("\r\n\r\nJob List:");
    System.out.println("\r\njobno\tWi/Pi\tprocessTime\tdeadLine\tweights");
    System.out.println("-----");
");
    for(int i=0; i<jobList.length; i++)
    {
        System.out.println( " " + jobList[i].getJobNumber() + "\t "
                               + jobList[i].getWeight() + "\t
"
                               +
jobList[i].getProcessTime() + "\t\t "
                               + jobList[i].getDeadLine() +
"\t\t "
                               + (jobList[i].getWeight() *
jobList[i].getProcessTime())
                               );
    }
}

/*-----
PURPOSE: generate a random integer under a max number.
RECEIVES: nothing
RETURNS: nothing
-----*/
private int getRandom(int Max)
{
    return((int)(Math.random() * (double)Max) + 1);
}

/*-----
PURPOSE: Display sorted job list for debugging.
RECEIVES: nothing
RETURNS: nothing
-----*/
private void displaySortedData()
{
    System.out.println("\r\n\r\nSorted Job List:");
    System.out.println("\r\njobno\tWi/Pi\tprocessTime\tdeadLine\tweights");
    System.out.println("-----");
");

```

```

        for(int i=0; i<sortedJobList.length; i++)
        {
            System.out.println( " " + sortedJobList[i].getJobNumber() + "\t "
                                +
sortedJobList[i].getWeight() + "\t "
                                +
sortedJobList[i].getProcessTime() + "\t\t "
                                +
sortedJobList[i].getDeadLine() + "\t\t "
                                +
(sortedJobList[i].getWeight() * sortedJobList[i].getProcessTime())
                                );
        }
    }

    /*-----
PURPOSE: Sort jobs according to job weights with quick sort algorithm.
RECEIVES: nothing
RETURNS: nothing
-----*/
    public void sort()
    {
        Sorter sortJob = new QuickSorter();
        sortJob.sort(jobList, sortedJobList);
    }

    /*-----
PURPOSE: make schedule per Greedy algorithm.
RECEIVES: nothing
RETURNS: nothing
-----*/
    private void makeSchedule()
    {
        scheduleMaker scheduleJob = new GreedyScheduleMaker();
        scheduleJob.makeSchedule(sortedJobList, schedule);
    }

    /*-----
PURPOSE: Display ssheduled job list and statistics.
RECEIVES: nothing
RETURNS: nothing
-----*/
    private void displayResult()
    {
        System.out.println("\r\n\r\nSchedule:");
        System.out.println("\r\ntime\tjobno");
    }

```

```

System.out.println("-----");
for(int i=0; i<jobList.length; i++)
{
    System.out.println(" " + (i+1) + "\t" + schedule[i]);
}

int penalties = 0;
int penalty_count = 0;
for(int k=0; k<jobList.length; k++)
{
    if( (k+1) > jobList[schedule[k]-1].getDeadLine() )
    {
        penalty_count++;
        penalties += jobList[schedule[k]-1].getWeight();
    }
}
System.out.println("\r\nThe number of late tasks = " + penalty_count);
System.out.println("\r\nThe total penalties = " + penalties);
}

public static void main(String args[])
/*-----
PURPOSE: creates instance of class Greedy. Sort, make schedule and
display scheduled job list.
RECEIVES: nothing
RETURNS: nothing
-----*/
{
    System.out.println("Starting Greedy job scheduler ...");
    Greedy testObj = new Greedy();
    testObj.displayData();
    testObj.sort();
    testObj.displaySortedData();

    /*testObj.sortWeights()*/
    testObj.makeSchedule();
    testObj.displayResult();
    System.out.println("\r\n");
}

}
class GreedyScheduleMaker extends scheduleMaker{

    private Job [] sortedjobList;
    private int [] schedule;

```

```

public GreedyScheduleMaker() {}

void makeSchedule(Job [] sortedList, int [] sch)
{
    sortedjobList = sortedList;
    schedule = sch;
    for(int i=0; i<schedule.length; i++)
    {
        schedule[i] = 0;
    }
    makeSchedule();
}

private void makeSchedule()
{
    for(int i=0; i<sortedjobList.length; i++)
    {
        int deadline = sortedjobList[i].getDeadLine();

        for(int j=0; j<sortedjobList.length; j++)
        {
            if(schedule[deadline-1] == 0)
            {
                schedule[deadline-1] =
sortedjobList[i].getJobNumber();
                break;
            }
            else
            {
                deadline--;
                if(deadline == 0)
                    deadline = sortedjobList.length;
            }
        }
    }
}
}

```

Appendix C: Source code for Sc-R, FIFO and SPTF algorithms

```
/*
*****
* Project:      Testing the variants in scheduling algorithms
* File:        SchedulerCompare.java
* Description:  For every second, a random number is generated for the number of
*              job released at this time. If this number is not zero, the exact
*              number of random jobs are created and added into R,S,F scheduler
*              queue according to respective algorithm. Only the first ten second can
*              release jobs.
*              The job at the top of scheduler queue run one second as the scheduler
*              algorithm defined. The program terminated after all scheduler queues
*              are empty and statistics of the schedules calculated at the end.
*****
*/
```

```
class SchedulerCompare {

    public SchedulerCompare()
    /*-----
    PURPOSE: constructor, creat scheduler queues..
    RECEIVES: nothing
    RETURNS: nothing
    -----*/
    {

        R_execQueue = new Queue();
        R_generateQueue = new Queue();
        S_execQueue = new Queue();
        S_generateQueue = new Queue();
        F_execQueue = new Queue();
        F_generateQueue = new Queue();
    }

    public int ReleaseJobsCount(int time)
    /*-----
    PURPOSE: generate a random number for jobs released at the time.
    RECEIVES: an integer of the time
    RETURNS: an integer of released jobs at the second
    REMARKS: first second at least one job is going to be released..
    -----*/
    {

        int count;

        count = (int)(Math.random() * 10);

        if(count < 3)
        {
```

```

        count = 0;
    }
    else if(count < 8)
    {
        count = 1;
    }
    else
    {
        count = 2;
    }

    if((time == 1) && (count == 0))
    {
        count = 1;
    }

    return(count);
}

```

```
public void run()
```

```
/*-----*/
```

into
run
and statistics

PURPOSE: Loop for every second. Jobs are generated in every second and added
R. S. F. working queues. The job at the top of each working queue
for one second. The loop end when all working queues are empty
are displayed for each scheduler algorithm.

RECEIVES: nothing.

RETURNS: nothing.

REMARKS: only first ten second can generate jobs.

```
-----*/
```

```

{
    int i, count, R_DONE, S_DONE, F_DONE;
    int time = 1;
    Job currJob;

    Scheduler R_Sched = new Rscheduler();
    Scheduler F_Sched = new FifoScheduler();
    Scheduler S_Sched = new Sscheduler();

    count = ReleaseJobsCount(time);
    for(i=0; i<count; i++) //move to queue
    {
        currJob = new Job(time);
        R_Sched.addJob(currJob);
    }
}

```

```

R_generateQueue.enqueue(currJob);

currJob = new Job(currJob);
S_Sched.addJob(currJob);
S_generateQueue.enqueue(currJob);

currJob = new Job(currJob);
F_Sched.addJob(currJob);
F_generateQueue.enqueue(currJob);
}

R_DONE = 0;
S_DONE = 0;
F_DONE = 0;
while((F_DONE == 0) || (S_DONE == 0) || (R_DONE == 0))
{
    if((R_DONE == 0) && ((currJob = R_Sched.run(time)) != null))
    {
        R_execQueue.enqueue(currJob);
    }
    else
    {
        R_DONE = 1;
    }

    if((S_DONE == 0) && ((currJob = S_Sched.run(time)) != null))
    {
        S_execQueue.enqueue(currJob);
    }
    else
    {
        S_DONE = 1;
    }

    if((F_DONE == 0) && ((currJob = F_Sched.run(time)) != null))
    {
        F_execQueue.enqueue(currJob);
    }
    else
    {
        F_DONE = 1;
    }

    time++;
    if(time < 10)
    {

```

```

        count = ReleaseJobsCount(time);
        for(i=0; i<count; i++) //move to queue
        {
            currJob = new Job(time);
            R_Sched.addJob(currJob);
            R_generateQueue.enqueue(currJob);

            currJob = new Job(currJob);
            S_Sched.addJob(currJob);
            S_generateQueue.enqueue(currJob);

            currJob = new Job(currJob);
            F_Sched.addJob(currJob);
            F_generateQueue.enqueue(currJob);
        }
    }

    System.out.println("\r\n\r\nSemi-clairvoyant R Scheduler:\r\n");
    R_generateQueue.listJobs();
    R_generateQueue.calculate();
    R_execQueue.listExec();

    System.out.println("\r\n\r\nShortest Processing time First Scheduler:\r\n");
    S_generateQueue.listJobs();
    S_generateQueue.calculate();
    S_execQueue.listExec();

    System.out.println("\r\n\r\nFirst In First Out Scheduler:\r\n");
    F_generateQueue.listJobs();
    F_generateQueue.calculate();
    F_execQueue.listExec();
}

public static void main(String args[])
/*-----
PURPOSE: creates new instance of class SchedulerCompare, and run scheduler
comparation.
RECEIVES: nothing
RETURNS: nothing
-----*/
{

    System.out.println("Starting Scheduler Compare...\r\n");
    //Job.displayListTitle();

```



```

        SchedulerCompare mainFrame = new SchedulerCompare();
        mainFrame.run();
    }

    private Queue R_execQueue;
    private Queue S_execQueue;
    private Queue F_execQueue;
    private Queue R_generateQueue;
    private Queue S_generateQueue;
    private Queue F_generateQueue;
}

class Rscheduler extends Scheduler{

    public Rscheduler(){
        int i;

        jobsClass = new Queue [4];
        for(i=0; i<4; i++)
        {
            jobsClass[i] = new Queue();
        }
    }

    public Job run(int time)
    {
        int i, j;
        int count=0;
        Job firstJob, secondJob;

        for(i=0; i<jobsClass.length; i++)
        {
            count = jobsClass[i].getItemNum();
            if( count == 0)
            {
                continue;
            }
            else
            {
                break;
            }
        }

        if(i >= jobsClass.length)

```

```

    {
        return(null);
    }

    firstJob = jobsClass[i].getFirstItem();
    if((count == 1) && (firstJob.bIsTotalJob() == 1))
    {
        for(j=i+1; j<jobsClass.length; j++)
        {
            count = jobsClass[j].getItemNum();
            if( count == 0)
            {
                continue;
            }
            else
            {
                break;
            }
        }

        if(j < jobsClass.length)
        {
            secondJob = jobsClass[j].getFirstItem();
            if(secondJob.bIsTotalJob() == 0)
            {
                if(secondJob.run(time) == 0)
                {
                    jobsClass[j].deQueue();
                }
                return(secondJob);
            }
        }
    }

    if(firstJob.run(time) == 0)
    {
        jobsClass[i].deQueue();
    }
    return(firstJob);
}

public void addJob(Job currJob)
{
    if(currJob.getProcessTime() < 2)
    {

```

```

        jobsClass[0].enqueue(currJob);
    }
    else if(currJob.getProcessTime() < 4)
    {
        jobsClass[1].enqueue(currJob);
    }
    else if(currJob.getProcessTime() < 8)
    {
        jobsClass[2].enqueue(currJob);
    }
    else
    {
        jobsClass[3].enqueue(currJob);
    }
}

private Queue [] jobsClass;
}
class Rscheduler extends Scheduler{

    public Rscheduler(){
        int i;

        jobsClass = new Queue [4];
        for(i=0; i<4; i++)
        {
            jobsClass[i] = new Queue();
        }
    }

    public Job run(int time)
    {
        int i, j;
        int count=0;
        Job firstJob, secondJob;

        for(i=0; i<jobsClass.length; i++)
        {
            count = jobsClass[i].getItemNum();
            if( count == 0)
            {
                continue;
            }
            else
            {

```

```

        break;
    }
}

if(i >= jobsClass.length)
{
    return(null);
}

firstJob = jobsClass[i].getFirstItem();
if((count == 1) && (firstJob.bIsTotalJob() == 1))
{
    for(j=i+1; j<jobsClass.length; j++)
    {
        count = jobsClass[j].getItemNum();
        if( count == 0)
        {
            continue;
        }
        else
        {
            break;
        }
    }

    if(j < jobsClass.length)
    {
        secondJob = jobsClass[j].getFirstItem();
        if(secondJob.bIsTotalJob() == 0)
        {
            if(secondJob.run(time) == 0)
            {
                jobsClass[j].deQueue();
            }
            return(secondJob);
        }
    }
}

if(firstJob.run(time) == 0)
{
    jobsClass[i].deQueue();
}
return(firstJob);
}

```

```

public void addJob(Job currJob)
{
    if(currJob.getProcessTime() < 2)
    {
        jobsClass[0].enqueue(currJob);
    }
    else if(currJob.getProcessTime() < 4)
    {
        jobsClass[1].enqueue(currJob);
    }
    else if(currJob.getProcessTime() < 8)
    {
        jobsClass[2].enqueue(currJob);
    }
    else
    {
        jobsClass[3].enqueue(currJob);
    }
}

private Queue [] jobsClass;
}
class Sscheduler extends Scheduler{

    public Sscheduler(){
        workinigQueue = new Queue();
    }

    public Job run(int time)
    {
        int i, j;
        Job firstJob;

        if(workinigQueue.getItemNum() == 0)
        {
            return(null);
        }

        firstJob = workinigQueue.getFirstItem();

        if(firstJob.run(time) == 0)
        {
            workinigQueue.dequeue();
        }
    }
}

```

```

        }
        return(firstJob);
    }

    public void addJob(Job currJob)
    {
        workingQueue.insertAsendProcessTime(currJob);
    }

    private Queue workingQueue;
}
class FifoScheduler extends Scheduler{

    public FifoScheduler(){
        workingQueue = new Queue();
    }

    public Job run(int time)
    {
        int i, j;
        Job firstJob;

        if(workingQueue.getItemNum() == 0)
        {
            return(null);
        }

        firstJob = workingQueue.getFirstItem();

        if(firstJob.run(time) == 0)
        {
            workingQueue.dequeue();
        }
        return(firstJob);
    }

    public void addJob(Job currJob)
    {
        workingQueue.enqueue(currJob);
    }

    private Queue workingQueue;
}

```