COSC 2P13:

SPOOLer Simulation Project

Submitted To: Vlad Wojcik

Submitted By: Owen Brown

Submitted On: April 6th, 2015

**Table of Contents**

**1 Introduction** 1.1 - The Problem  
 1.2 - The Queues  
 1.3 - Simulation Approach

**2 Simulation Results** 2.1 - FIFO  
 2.2 -SPJF  
 2.3 -PAPQ  
 2.4 -LBAQ

**3 Conclusions** 3.1 - Advantages and Disadvantages  
 3.2 - Libraries Used , Setup & Further Reading

1 - Introduction

1.1 - The Problem

The printer is - at the basic level - simply handles scarce resources (printing time) and manages requests for those resources (print jobs) accordingly. This behaviour mirrors the resource management of operating systems, where the scarce resources (processors) are managed and distributed to requests for those resources (processes and threads).  
 The SPOOL (or simultaneous peripheral operation on-line) handles the reading in of such requests and storing for processing at a more convenient time. The SPOOL can implement a number of strategies for the organization of requests, ensuring a particular kind of request is always given priority.  
 In our case, the printer in question can print at a rate of 40 pages per minute (or 2/3 pages per second). The average print job is 4 pages long, so it follows that the printer shouldn't encounter any exorbitant turnaround times as long as the request frequency is under 40 / 4 = 10 requests per minute.  
 We will conduct a simulation over four queues in order to determine which type is most advantageous in specific situations. For our purposes, we will define the following values:  
  
 : Time it takes for a request to print after being selected. Let q be the number of pages and z be the rate at which a printer can print pages (in our case, 2/3 pages per second). Then , .

: Amount of time (in seconds) the request has spent waiting to be selected as the next job to be printed.

: Time it takes for a print request to finish printing from the time it arrived in the queue.

1.2 - The Queues

The simulation will be conducted using four types of queues: FIFO, SPJF, PAPQ and LBAQ. We will define each queue, outlining its approach to managing requests.  
  
**FIFO** (First In First Out) - Print requests are serviced based on their time of arrival.  
**SPJF**(Shortest Print Job First) - Print requests are serviced based on the number of pages to be printed.  
**PAPQ**(Priority-Aged Print Queue) - Each print request arrives with a priority of *A*, and has its priority increase at a rate of *B* based on its *T* value. So we can define the priority to be :

**LBAQ**(Length-Based Aged Queue) - Each print request arrives with a priority of *A*, and has its priority increase at a rate of *B* based in its *T* value. This final value is then divided by the number of pages *L* of the request to return the final priority value. So we can define the priority to be:

Each queue is expected to behave differently given their distinct approach to handling the print requests.

1.3 - Simulation Approach

The design of the simulation is based on two facts: the number of pages for each request follows Poisson distribution with a λ of 4, the time between arrivals follows exponential distribution.  
 The first of these properties is achieved through use of a Poisson Generator, an object with uses the Math.random() uniform distribution as a parameter and a user defined λ. The specific library used in this case is the math uncommons1, which has a built in Poisson Generator with the aforementioned attributes. The number generated is added to the print request with the following code:

for (int i = 0; i<numReq-1; i++){

j = pg.nextValue();

while (j == 0){

j = pg.nextValue();

}

}

Random rand;

PoissonGenerator pg;

rand = new Random();

pg = new PoissonGenerator(4,rand);

where pg is the Poisson distribution number generator and pg.nextValue() returns an appropriate value. The value 0 is avoided through use of the while loop, which (in the case of a 0) runs nextValue() again until a non-zero value is assigned. This presents us with a slightly biased distribution known as Zero-Truncated Poisson Distribution. This is a necessary step however, as a print request of 0 pages is illogical for our purposes.

The second property, exponentially distributed difference in arrival times, is achieved through the following code:

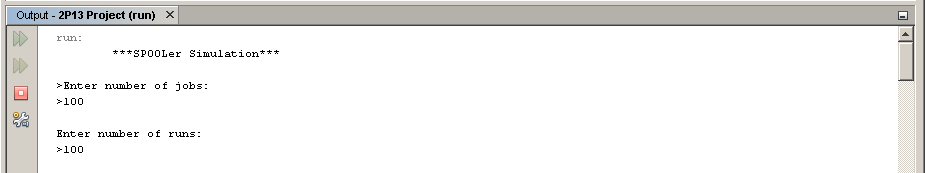
double arrivetime = 0;  
job = new Request((Math.log(1-Math.random()))/(-(freq/60.0))+arrivetime,j);

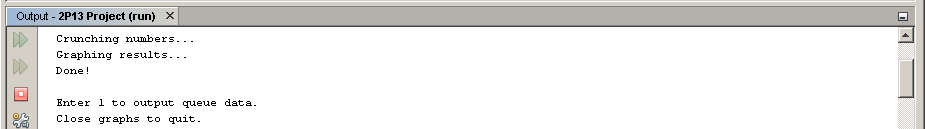
this code makes use of the formula for exponential distribution, specifically :  
C:\Users\Owen\Desktop\expDist.png  
(Equation source: http://en.wikipedia.org/wiki/Exponential\_distribution)

this ensures inter-arrival times will follow exponential distribution, where p is a value returned by Math.random() and = printing rate. To get the rate in terms of seconds, we simply divide our number (1, 2,...9) by 60.

For the simulation itself, the process is as follows:

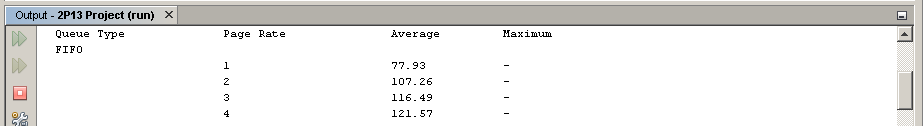
* 1. Prompt user for number of print jobs x.
  2. Prompt user for number of runs y.
  3. Conduct simulation of x jobs using each queue with a print frequency of 1,2,..,9 for a total of y runs.
  4. Graph results, print relative data at user's request.

This is a sample output of the simulation run:  
User inputs 100 for number of jobs and 100 for number of runs: 



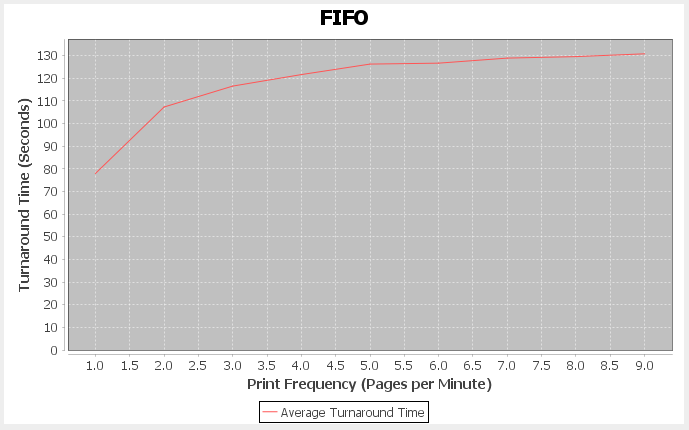
The user should have a set of graphs pop up in a window at this point:  
These graphs will include 1 FIFO, 1 SPJF, 12 PAPQ for each AB combination and 12 LBAQ for each AB combination.  
  
\*User may zoom in by clicking and dragging top-left to bottom-right.  
Zoom out by clicking and dragging from bottom-left to top-right.  
  
The user also has the option to enter '1' to see the numerical representation of the graphs.

This output can be seen below:

  
Finally, the user simply has to click the 'x' on the graph window to stop the program from running.

2 - Simulation Results  
\*The following results were obtained with the input of 100 jobs for 100 runs:

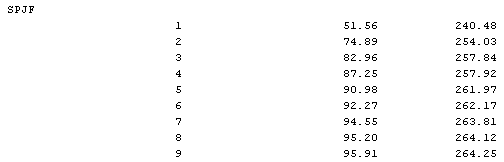
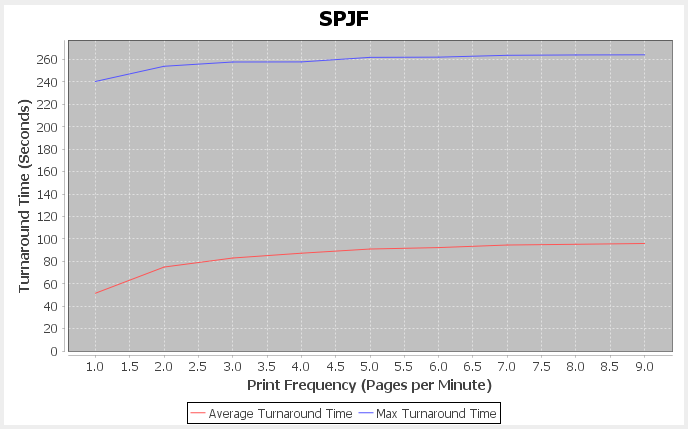
2.1 - FIFO



Based on the graph and the numerical results, we may draw the following conclusions of FIFO:

* FIFO has a sharp increase in *U* when the Print Frequency increases from 1 ppm (pages per minute) to 2ppm.
* Each subsequent jump in page frequency leads to a smaller increase in *U*, as it approaches its maximum printing capacity of 10 ppm.
* FIFO's fair approach to processing print requests appears to come at a cost of a higher average *U* when compared to other algorithms.

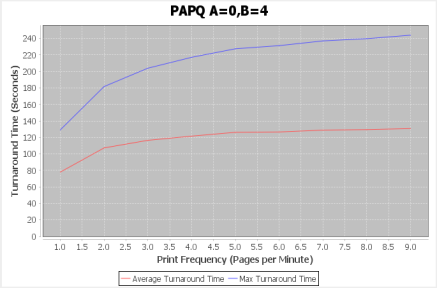
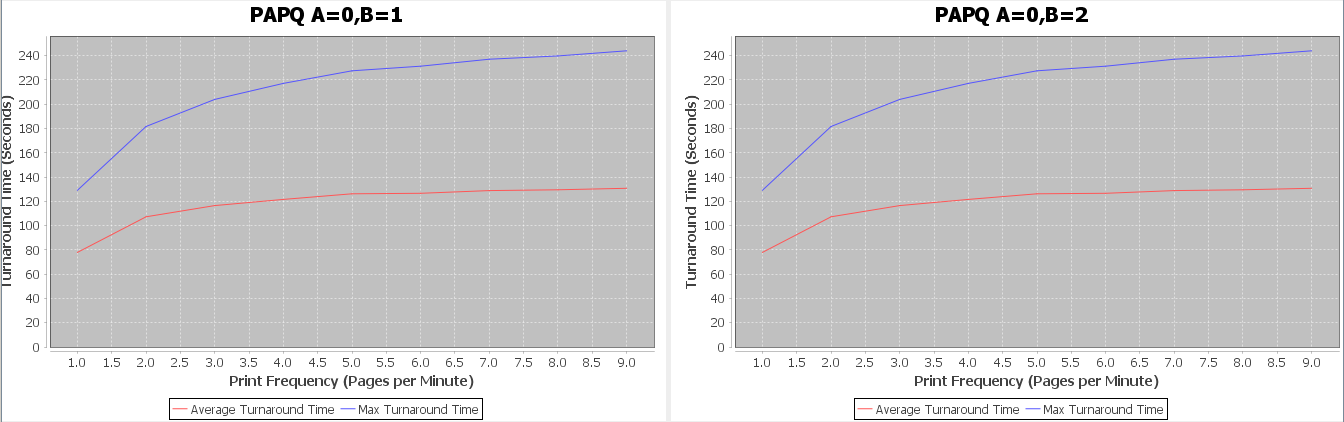
2.2 - SPJF

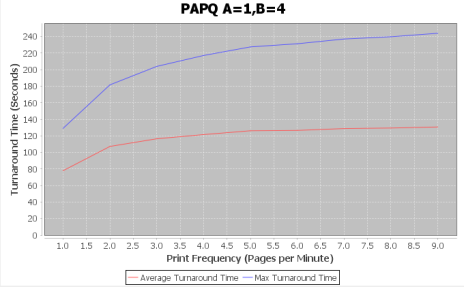
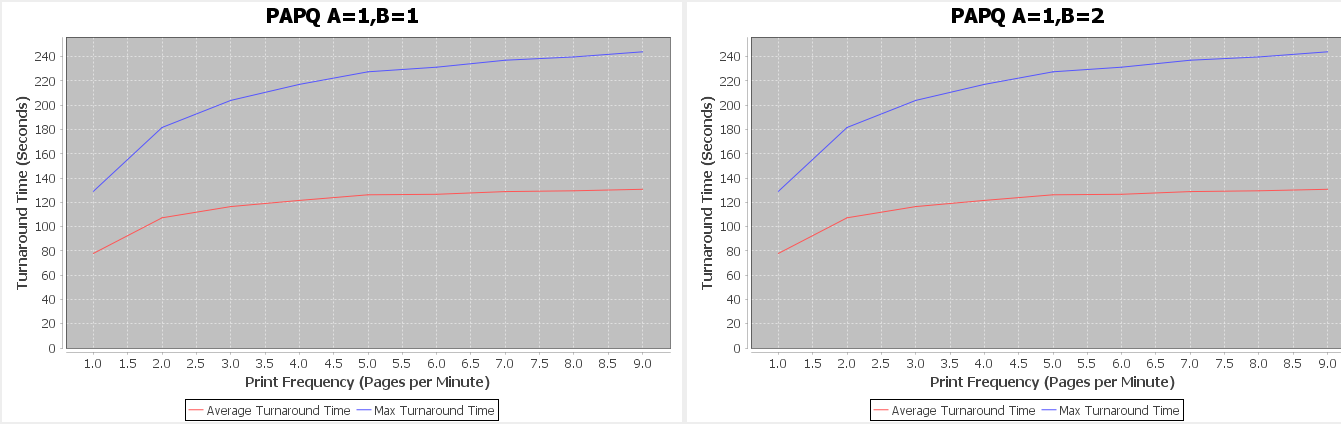


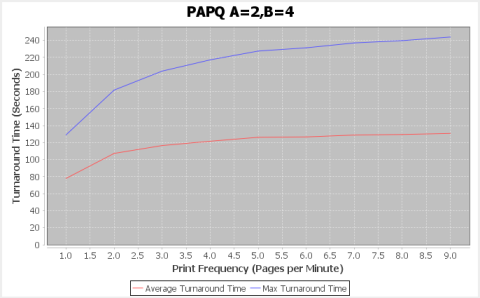
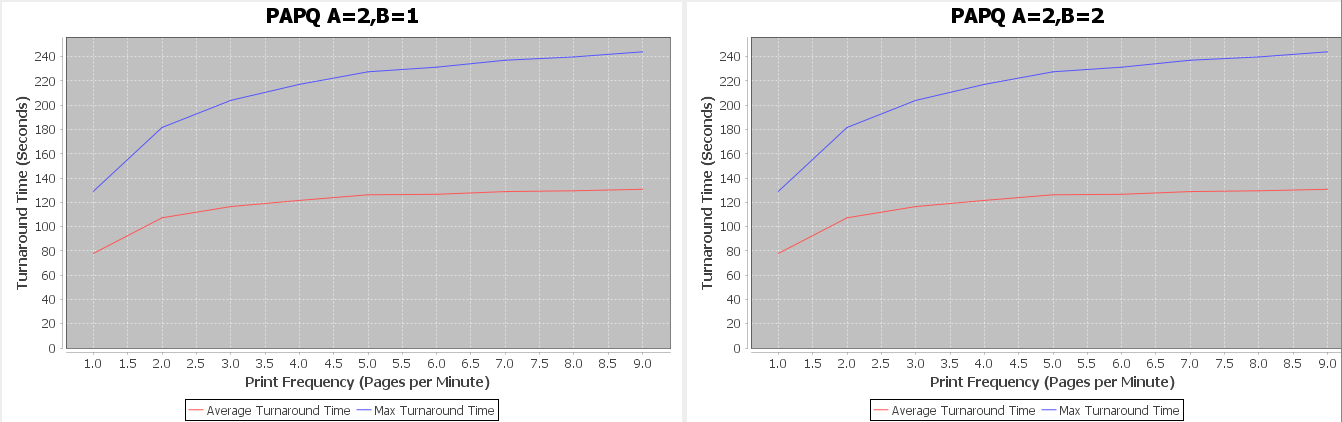
Based on the graph and the numerical results, we may draw the following conclusions of SPJF:

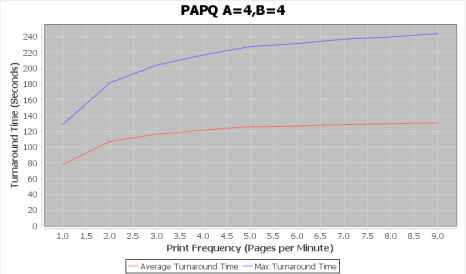
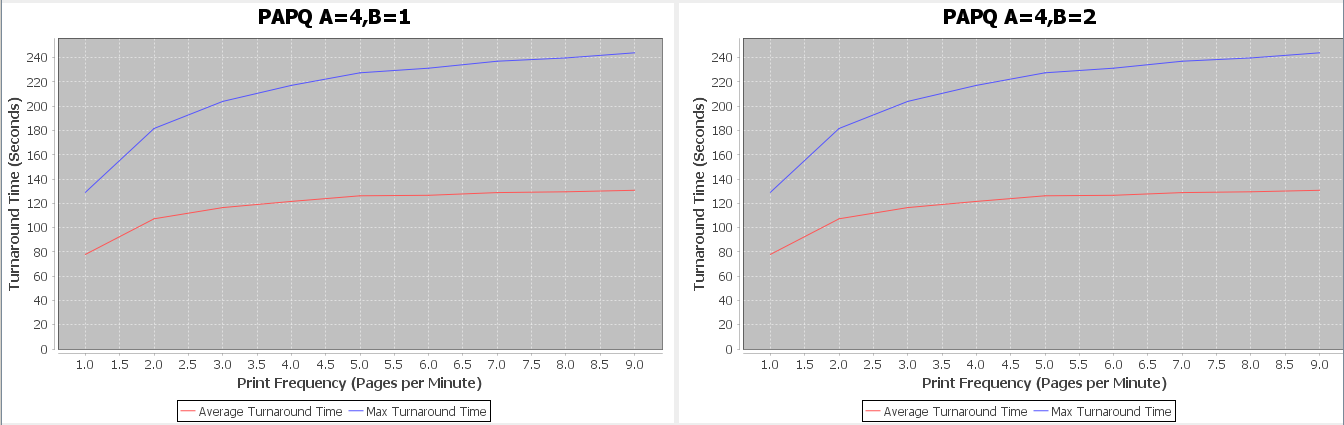
* SPJF has a sharp increase in *U* when the Print Frequency increases from 1 ppm (pages per minute) to 2ppm.
* Average *U* appears to asymptotically approach number of jobs. This performance is an improvement over FIFO. However the maximum wait time for each frequency appears to be high as long pages are stuck in the queue.

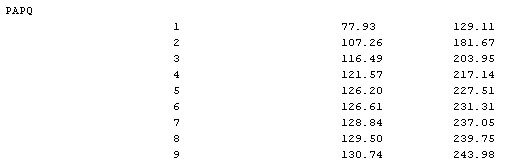
2.3 - PAPQ







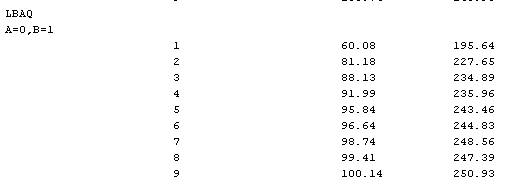
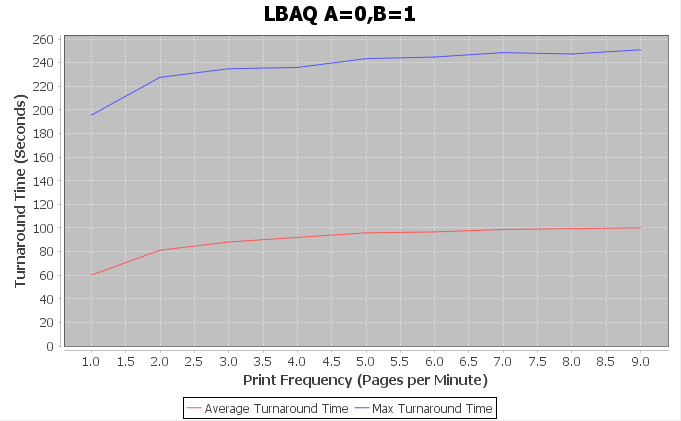
  
Every PAPQ variant produces the exact same data, the data for these graphs is displayed below:

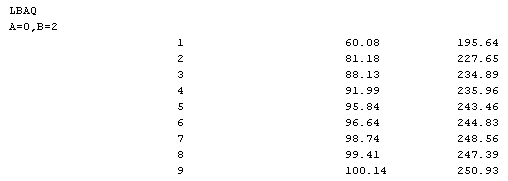
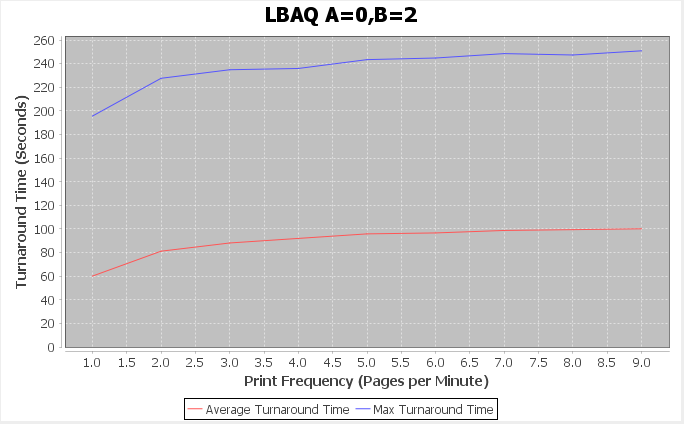
It becomes obvious, when comparing this data with FIFO, that PAPQ achieves the exact effect of FIFO.

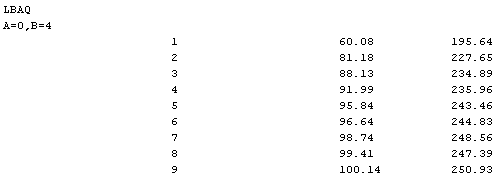
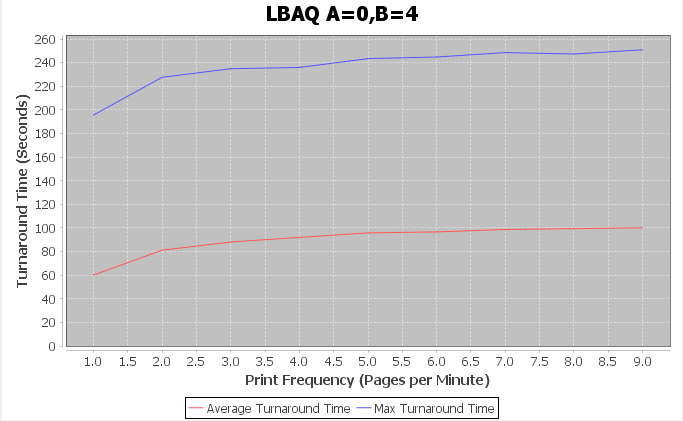
Based on the graph and the numerical results, we may draw the following conclusions of PAPQ:

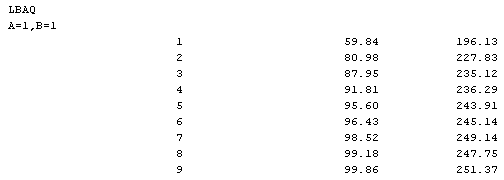
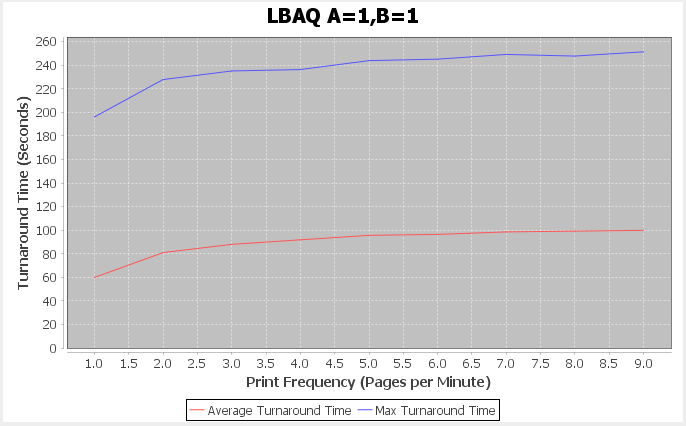
* PAPQ's performance mirrors FIFO, since the constants change in linear fashion with wait time *T*.
* PAPQ has a lower max *U* than SPJF as jobs cannot be perpetually delayed by new arrivals.

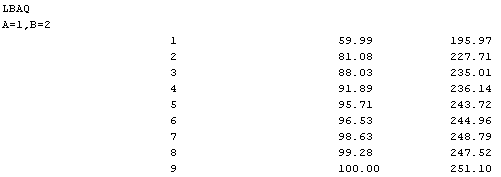
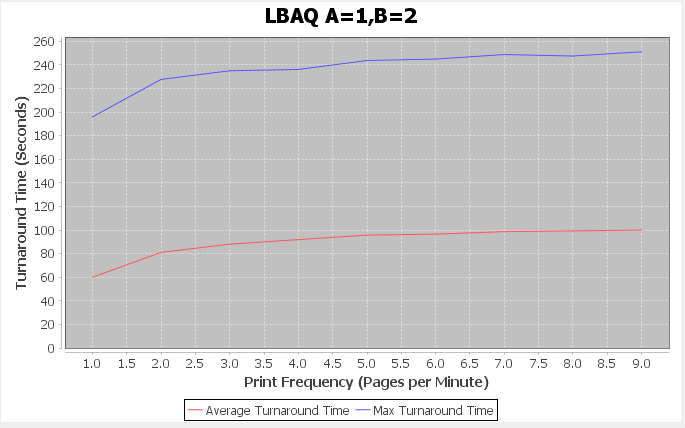
2.4 - LBAQ

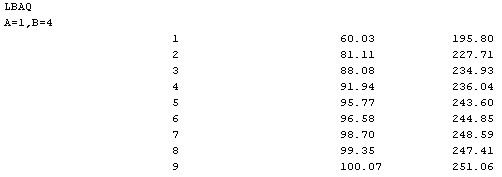
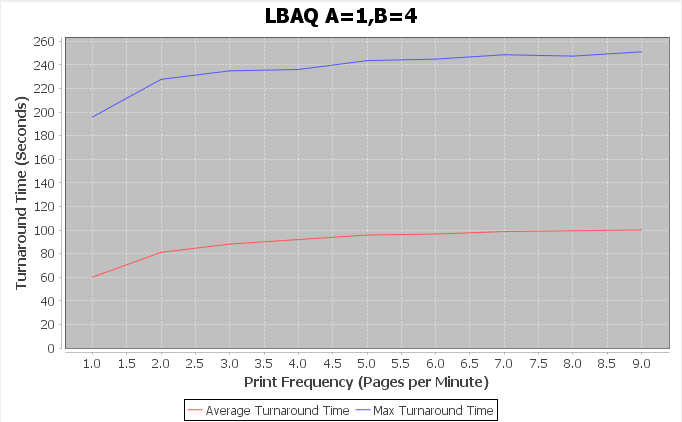


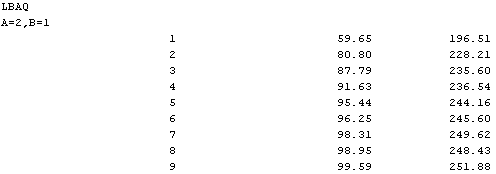
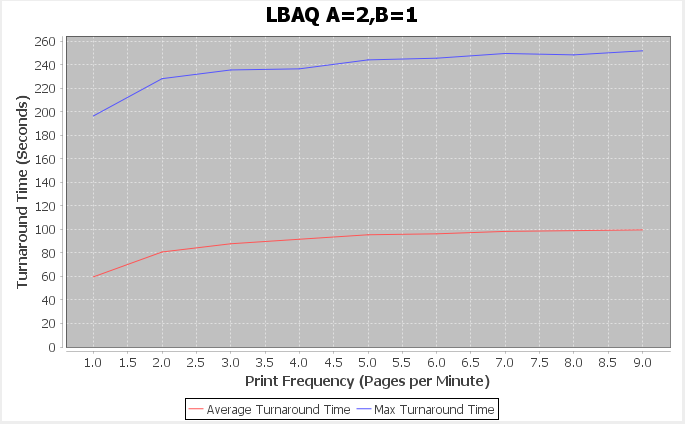


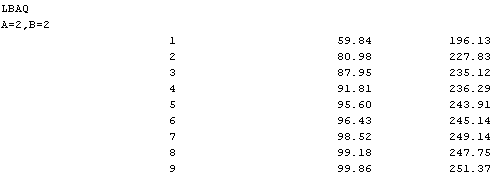
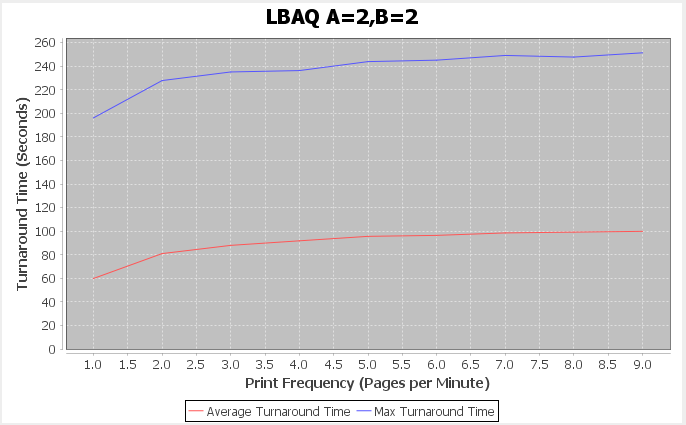


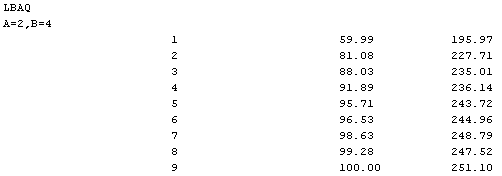
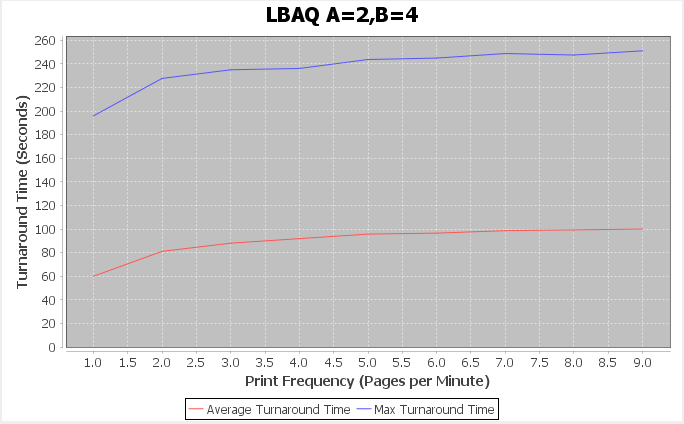


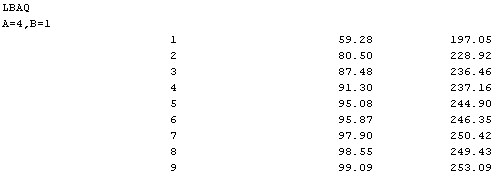
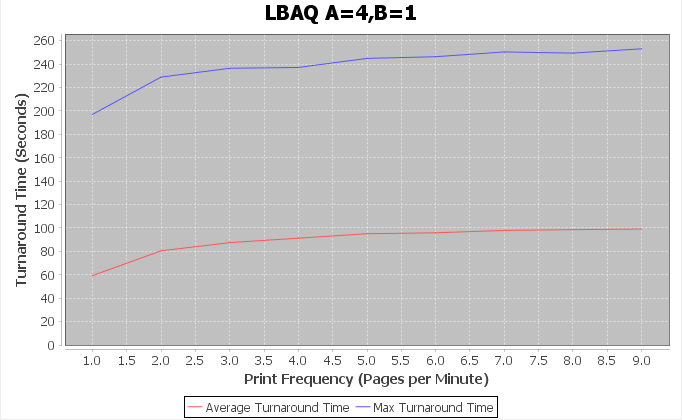


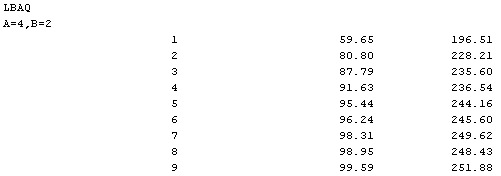
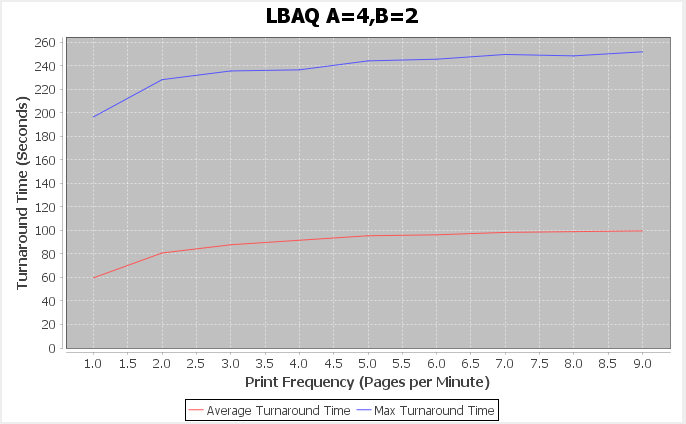


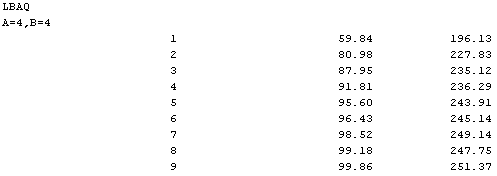
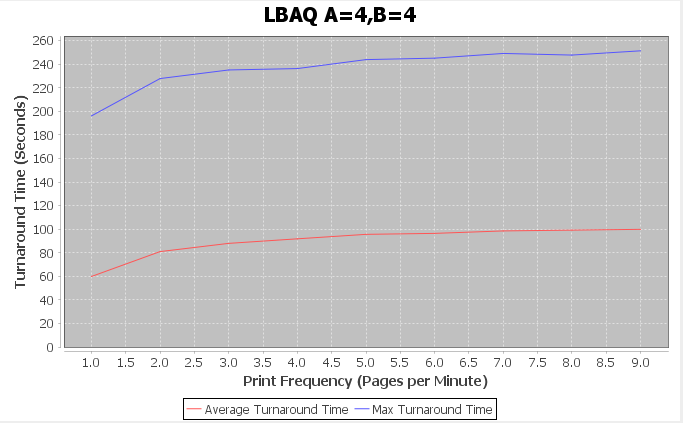












Based on the graph and the numerical results, we may draw the following conclusions of LBAQ:

* Average *U* lower than FIFO and PAPQ at all frequencies, but performs worse in terms of average *U* than SPJF.
* Max *U* lower than SPJF at all frequencies, but higher than PAPQ and FIFO.
* Across inputs, we observe a slight change in performance. As A increases, we see an slight overall decrease in average *U* and a slight overall increase in max *U*.
* Similar to HRRN (Highest response ratio next).

3 - Conclusions

3.1 - Advantages and Disadvantages

The exponentially distributed arrival times seem to affect our Y intercept of Turnaround Time. This means as the number of jobs increase, we can expect an increase in the initial Turnaround Time of each queue.

In terms of Average Turnaround Time, the algorithms performed accordingly:

1. SPJF
2. LBAQ (Overall)
3. FIFO & PAPQ
4. FIFO & PAPQ

The clear winner was SPJF. Over multiple reasonable data sets, SPJF outperformed all other queue's in terms of *U*. This makes SPJF the ideal print job management system to employ in our SPOOL assuming steady printer use over time.

In terms of Maximum Turnaround Time, the algorithms performed accordingly:

1. FIFO & PAPQ
2. FIFO & PAPQ
3. LBAQ
4. SPJF

We can observe that the performance of maximum turnaround time is inversely correlated to the relative average turnaround times. This makes FIFO and PAPQ the most ideal queue's for avoiding exorbitant max queue times. This would make them the optimal queuing system when attempting to keep a cap on wait times.

3.2 - Libraries Used, Setup and Further Reading

The following libraries were used in the creation of the simulation:

[1] http://maths.uncommons.org

* Poisson Distribution number generator, used to assign page numbers.

[2] www.jfree.org

* Graphing library, used to display results of simulation.

All appropriate library files should be included in the final project submission, specifically:

* uncommons-maths-1.2.3.jar
* jfreechart-1.0.19 library suite (multiple jar files)

***Setup:***

All relative JAR files are included in a folder called Libraries. If the compiler complains about missing libraries, simply add the JAR files into the project before running.

*NetBeans*:

* Navigate to the Project and open in Netbeans
* In the Projects pane, under Source directory there should be a Libraries directory
* Right-click and select 'Add JAR/Folder'
* Navigate to the Libraries folder included in submission
* Select all JARS and click 'Open'

***Further Reading...***

A number of alternative queue's exist that may further improve on the performance of print jobs (lower turnaround time) in the SPOOL. Implementation of the following may improve our understanding on how the four queue's proposed may be optimized:

* Round Robin Scheduling (could be implemented to print small jobs while a large job is being processed - not strictly RRS but a variation).
* Deadline Scheduling (could be achieved by providing a deadline with each job request submission, this would assume users wouldn't exploit).
* Multilevel Queue Scheduling (could be achieved by choosing some arbitrary cut off point for 'small' and 'large' jobs).