Low Cost Job Scheduling

Chaz Lambrechtsen | 45426317

Introduction

This project will focus on a job scheduling algorithm that is different to baseline algorithms (First Fit, Best Fit and Worst Fit). The goal of the algorithm should implement at least one of the following optimisations:

- Minimisation of average turnaround time
- Maximisation of average resource utilisation
- Minimisation of total server rental cost

Problem definition

The baseline algorithms FF, WF, BF are limited and only useful for some situations. The indicators to consider in this problem are, wait time, cost, simulation end time, execution time, and turnaround time. None of the baseline algorithms reduce the total cost. I have created an algorithm to do exactly this, the performance objective of this algorithm will be to minimise the total cost of server rental potentially at the cost of other performance metrics.

Algorithm description

After some initial research and investigating the source code of the ds-server [2] I can see how some of these performance metrics are calculated. Using this information we can already see that lowering the cost will affect other metrics as some jobs may not be scheduled to the most optimal server, potentially increasing wait time and turnaround time.

Cost:
for each server in each server type
    server uptime * (server cost/TimeInSeconds)
Wait time:
Sum of total time spent on each job

Turnaround time:
Avg job finish time – job start time

Purpose

This algorithm's purpose is to decrease the total cost of server rental. This is implemented in the algorithm by prioritising idle or active servers as the highest priority because cost is increased by servers being active regardless of how many jobs are being scheduled.

Design

Calculations based on a single set of servers that can be reproduced for multiple sets:

Server Type	**Rate**	**Seconds**	**Gradient**
Tiny	0.1	3600	2.77778E-05
Small	0.2	3600	5.55556E-05
Medium	0.4	3600	0.000111111
Large	0.8	3600	0.000222222
xLarge	1.6	3600	0.000444444
2xLarge	3.2	3600	0.000888889
3xLarge	6.4	3600	0.001777778
4xLarge	12.8	3600	0.003555556

From this table we can determine that the cost of a server is directly proportional to that of the server's uptime. To decrease the total cost of each server we must make sure that each server that is active or idle, gets another job as soon as possible to reduce the time spent not processing any jobs as this increases the uptime without working on jobs.
Additionally, we should consider how much of the server's resources are being used when active, if the server has the potential to complete another job simultaneously then it should do so as it would decrease the uptime by not having the job run later.This means that it is preferable to have a job scheduled on the same server at another job at the same time over scheduling it with the best fitting server.

Example

Let j = current job,

Set lowestValue and minAvail to very large number
Read data from system.xml and find smallest possible server to handle ji
Obtain server state information

```
For each server type i, si, in the order provided to you
        For each server j, si, j of server type si, from 0 to limit -1
                Calculate the fitnessValue of si, j
                If server si, j is (active or idle) and (fitnessValue < lowestValue or
                (fitnessValue is equal to LowestValue and the available time < minAvail)
                        8. Set lowestValue to fitnessValue
                End if
                If server si, j is inactive and the server type is equal to the smallest possible
                Server
                        Set smallestServer = si, j
                End if
        End for
End for
If lowestValue is found then
        Return server with lowestValue3
Else if smallestServer is found then
        Return smallestServer
Else
        Request the next job
End if
```

Implementation

Implementing this algorithm came with some refactoring as I had also implemented two new classes with this file. These classes are the Job class and the Server class.

The Job class is a simple class that has seven private fields of data, all of these fields related to the respective Strings from ds-server when the “REDY” request is received meaning the client is ready to receive the next job. The following is what each field is called and a short explanation of each field:

- Int startTime: The current job’s start time.
- Int id: The ID of the current job.
- Int estimatedRunTime: The estimated time that it will take for a job to be completed.
- Int cpu: The number of CPU cores required to complete the job.
- Int memory: The amount of memory required to complete the job.
- Int disk: The amount of disk space required to complete the job.

The Server class has a similar structure to the Job class. It also contains seven private variables. The following is what each field is called and a short explanation of each field:

- String type: This type of server it is (e.g. small, medium, large, etc).
- Int ID: The server's ID.
- Int state: The current state of the server (0 = inactive, 1 = booting, 2 = idle, 3 =
- active, 4 = unavailable).
- Int availTime: The time until the server is available (time is unknown if equals to 1)
- Int cpu: The amount of CPU cores the server currently has available.
- Int memory: The amount of memory the server currently has available.
- Int disk: the amount of disk space the server currently has available.
- Int runTime: Run time for this server
- Int waitTime: waiting time for this server

All of these fields are private because there is no need to change them once they are already set, and so to avoid any accidents, they have been made private with a set of getters to get the values of each of them.

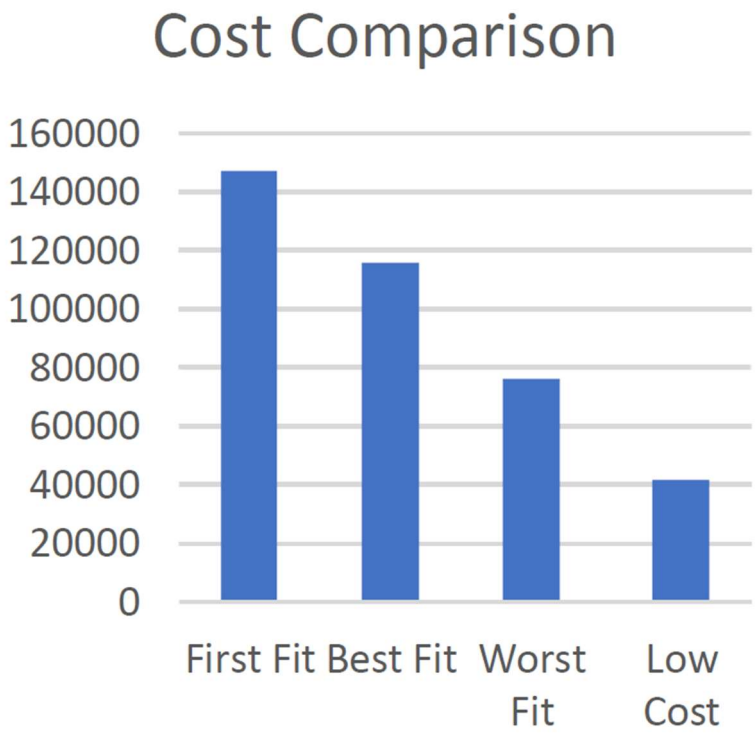
Implementing the Server class was also simple. I just added all new variables needed for stage 2 such as state , waitTime and runTime

Evaluation

This table represents the results from running baseline algorithms and Low Cost algorithm.

	Low Cost	First Fit	Worst Fit	Best Fit
Simulation End Time	6683944	2574535	5437397	2573860
Cost	41876.51	147028.8	76338.17	115684.33
Total Servers Used	4	62	10	42
Wait Time	2600754	631	1761109	191372
Execution Time	15617	14764	15524	15047
Turn Around Time	2616371	15395	1776634	206420

As expected the Low Cost algorithm is very cheap to use in terms of cost, only utilising 4 servers but this is a double edged sword because we can see a significant increase in turn around time, wait time and simulation end time.



Conclusion

In conclusion, the low cost algorithm successfully reduces the total cost while also being more efficient than First Fit which was the closest runner up in terms of reducing cost. There is most likely some room for improvement with this algorithm and given more time for testing and trialing different implementations I could have made this more efficient but this is the best implementation of this algorithm so far as it completes its objective of reducing total cost server rental costs very well. All in all this algorithm is great when there are not many jobs because it is cheap to run, however if there are too many jobs then it can take a long time and in a realistic scenario there may be a constant amount of jobs and the jobs could take an unlimited amount of time to complete which is why balance is important however was not the goal of this project.

References

[1] https://github.com/CazDev/Low-Cost-Job-Scheduling
[2] https://github.com/distsys-MQ/ds-sim