Cost efficient Job Scheduler for Distributed Systems

Michael Parker, 45663068.

Introduction:

Throughout the process of creating the three base-line algorithms for our job scheduler, it is evident that these algorithms are not optimal, and while they do a semi decent job at scheduling jobs, they are not the most efficient algorithms to be running if this project were to be pushed to a real environment. A post analysis of the three-baseline algorithm clearly shows that while the avg waiting time, avg execution time and turnover time are semi decent, the cost of running these servers using these algorithms would be detrimental to someone actually using this software. In order to fix this problem, a new scheduling algorithm will be created that will improve mainly upon the total cost when running these jobs.

Problem definition:

This new algorithm's objective is to be the new baseline algorithm with the goal of being an entry level algorithm for those creating a start-up or companies without the financial investment to be renting hundreds of servers, and spending thousands on jobs that have the capability to run much cheaper. In comparison to the three baseline algorithms that were created, it's noticeable that while each algorithm has a use in a particular scenario, none focus on limiting server usage alongside cost. Effectively this new algorithm, would be the cost-efficient algorithm in which to run as it would use the least number of servers than any other algorithm and it would also run far cheaper than the three baseline algorithms.

Algorithm description:

The way in which this new cost-efficient algorithm differs from the three baseline algorithms in terms of code, is that while the three baseline algorithms check if the 'current' server has the available resources to run the 'current' job , the new algorithm will skip this step and will in turn, calculate the fitness level and assign jobs to servers that originally held the closest fitness level to 0. This means that if a server originally had the exact resources to run a job, it would run that job and while this job was running another job came long which also had the exact same resources as the previously mentioned server, this new job would also be assigned to said server. If by some chance something was missed and there was no server to run the job, the algorithm would go to its backup algorithm in which it will check all servers that could originally run the job, and get the server which has a closest fitness to 0.

Pseudo code: cost efficient algorithm

For a given job ji,

1. set currentFit = to a very high number (INT_MAX)
2. obtain server information from system.xml or available resc commands
    3. For each server type t, st, in the same order as system.xml
        4. For each server i, st,i of server type st, until limit
        5. Calculate the fitness level fst,I,ji – defined as current server cores – current job cores
                6. if fitness level == 0 || fitness < currentFit
                        7. set currentFit to fitness level
                        8. return st,i
                9. end if
        10. end For
    11. end For
    12. if cost efficient match is found
        13. return the server with the best cost efficiency
    14. else
        15. set minTime to a very low number (MIN_VALUE)
        16. set minAvail to a very low number (MIN_VALUE)
                17. For each server i, si until limit
                        18. calculate bad fitness level fsi,ji
                        19. set minTime = badfitness
                        20. set minAvail = servers availtime
                        21. set cost efficient server = si
22. return cost efficient server

Comparisons on ds-config-s3-1.xml

CEA

Job ID	Job Requirements	Server no	Server Resources	Job Start	Server Resources	Job End
0	1 200 1200	#0 small	84 1 4000 16000	24	-1 0 3800 14800	19407
1	1 600 700	#0 small	-1 0 3800 14800	19407	-1 0 3400 15300	20502
2	1 600 800	#0 small	-1 0 3800 14800	20502	-1 0 3400 15200	22229
3	2 2100 1300	#0 medium	144 2 16000 64000	84	-1 0 13900 62700	1377
4	2 1100 2400	#0 medium	-1 0 13900 62700	1377	-1 0 14900 61600	2688
5	1 100 1600	#0 small	-1 0 3800 14800	22229	-1 0 3900 15400	23789
6	1 600 1500	#0 small	-1 0 3800 14800	23789	-1 0 3400 15500	23800
7	2 500 400	#0 medium	-1 0 13900 62700	2688	-1 0 15500 64600	3709
8	2 500 2800	#0 medium	-1 0 13900 62700	3709	-1 0 15500 61200	49136
9	1 600 200	#0 small	-1 0 3800 14800	23800	-1 0 3400 15800	25497

FF

Job ID	Job Requirements	Server no	Server Resources	Job Start	Server Resources	Job End
0	1 200 1200	#0 small	84 1 4000 16000	24	-1 0 3800 15800	19407
1	1 600 700	#1 small	137 1 4000 16000	77	-1 0 3400 15300	1232
2	1 600 800	#0 medium	140 2 16000 64000	80	-1 1 15400 64200	1867
3	2 2100 1300	#1 medium	144 2 16000 64000	84	-1 0 12900 63700	1377
4	2 1100 2400	#0 medium	-1 2 16000 64000	1867	-1 0 11900 61600	3178
5	1 100 1600	#0 medium	-1 2 16000 64000	3178	-1 1 13900 63600	4738
6	1 600 1500	#0 medium	-1 2 16000 64000	3178	-1 0 13300 62100	3189
7	2 500 400	#0 medium	-1 2 16000 64000	4738	-1 0 13500 63600	5759
8	2 500 2800	#0 medium	-1 2 16000 64000	5759	-1 0 13500 61200	51186
9	1 600 200	#0 medium	-1 2 16000 64000	51186	-1 1 13400 63800	52883

BF

Job ID	Job Requirements	Server no	Server Resources	Job Start	Server Resources	Job End
0	1 200 1200	#0 small	84 1 4000 16000	24	-1 0 3800 15800	19407
1	1 600 700	#1 small	137 1 4000 16000	77	-1 0 3400 15300	1232
2	1 600 800	#0 medium	140 2 16000 64000	80	-1 1 15400 64200	1867
3	2 2100 1300	#1 medium	144 2 16000 64000	84	-1 0 12900 63700	1377
4	2 1100 2400	#0 medium	-1 2 16000 64000	1867	-1 0 14900 61600	3178
5	1 100 1600	#0 medium	-1 2 16000 64000	3178	-1 1 15900 63600	4738
6	1 600 1500	#0 medium	-1 2 16000 64000	3178	-1 0 15300 62100	3189
7	2 500 400	#0 medium	-1 2 16000 64000	4738	-1 0 15500 63600	5759
8	2 500 2800	#0 medium	-1 2 16000 64000	5759	-1 0 15500 61200	51186
9	1 600 200	#0 medium	-1 2 16000 64000	51186	-1 1 15400 63800	52883

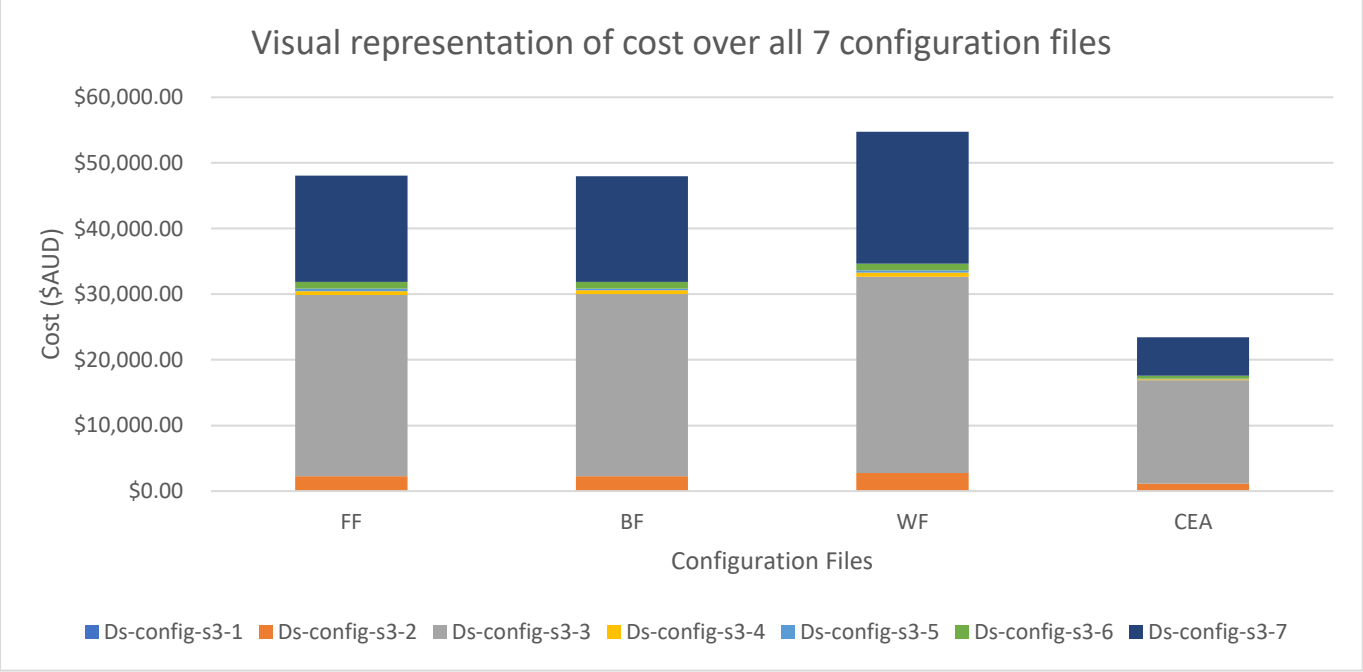
WF

Job ID	Job Requirements	Server no	Server Resources	Job Start	Server Resources	Job End
0	1 200 1200	#0 medium	124 2 16000 64000	24	-1 1 15800 63800	19407
1	1 600 700	#1 medium	137 2 16000 64000	77	-1 1 15400 63300	1232
2	1 600 800	#0 small	120 1 4000 16000	80	-1 0 3400 4200	1867
3	2 2100 1300	#0 medium	-1 2 16000 64000	19407	-1 0 13900 61600	20640
4	2 1100 2400	#0 medium	-1 2 16000 64000	20640	-1 0 14900 61600	21951
5	1 100 1600	#0 medium	-1 2 16000 64000	21951	-1 1 15900 62400	23511
6	1 600 1500	#0 medium	-1 2 16000 64000	21951	-1 0 15200 60900	21962
7	2 500 400	#0 medium	-1 2 16000 64000	23511	-1 0 15500 63600	24532
8	2 500 2800	#0 medium	-1 2 16000 64000	24532	-1 0 15500 61200	69959
9	1 600 200	#0 medium	-1 2 16000 64000	69959	-1 1 15400 63800	71656

Price comparison:

The following table and graphs show the price comparison from ds-config-s3-1 -> ds-confog-s3-7.

Config File	FF	BF	WF	CEA
Ds-config-s3-1	$13.86	$13.86	$15.99	$6.86
Ds-config-s3-2	$2241.44	$2274.35	$2728.55	$1133.19
Ds-config-s3-3	$27643.44	$27738.16	$29933.08	$15785.06
Ds-config-s3-4	$572.01	$568.64	$598.86	$138.24
Ds-config-s3-5	$389.26	$249.43	$357.49	$86.04
Ds-config-s3-6	$1013.62	$1026.81	$998.22	$444.90
Ds-config-s3-7	$16202.63	$16082.36	$20119.38	$5853.90



Implementation Details:

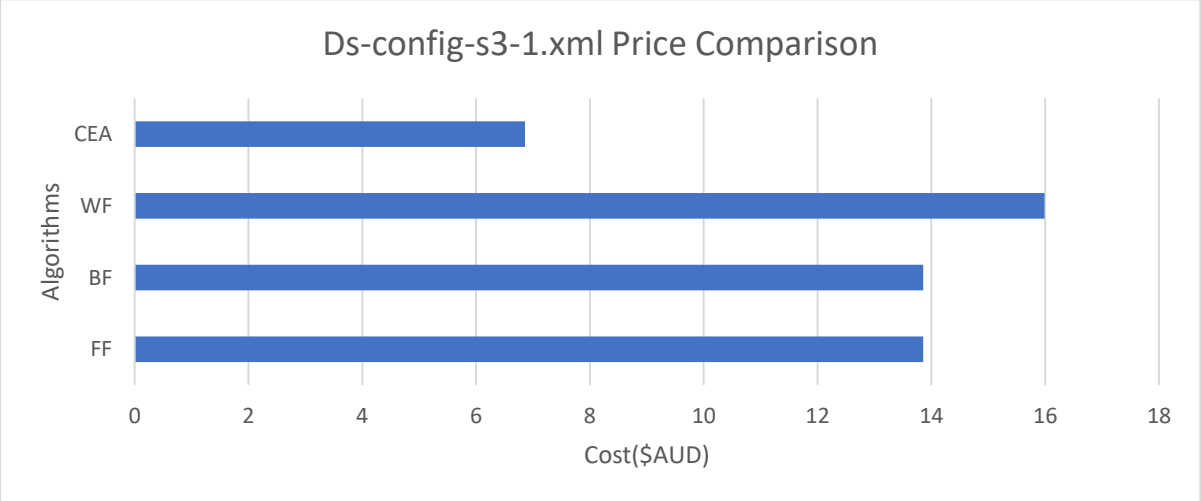
Throughout this implementation, data structures including that of an ArrayList containing server objects, an ArrayList containing strings and finally a hash set also containing strings have been used in creating the new algorithm.

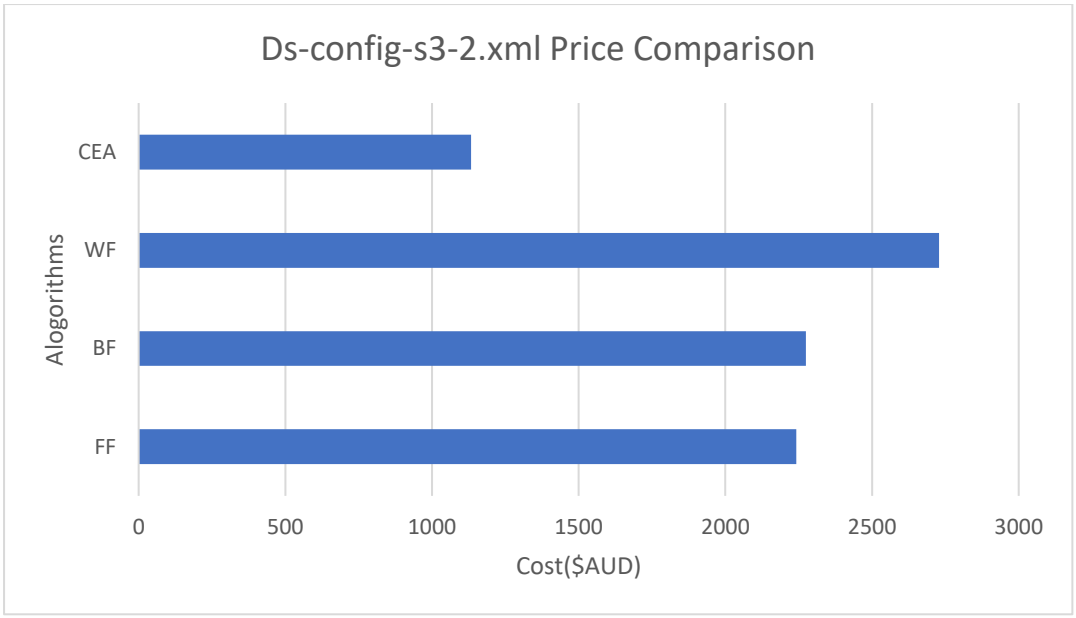
The first ArrayList contains respectively all servers sent through the use sending the command RESC Capable to the server. When a new job is dispatched, this ArrayList will be cleared in order to get the updated servers and avoid any conflicts.

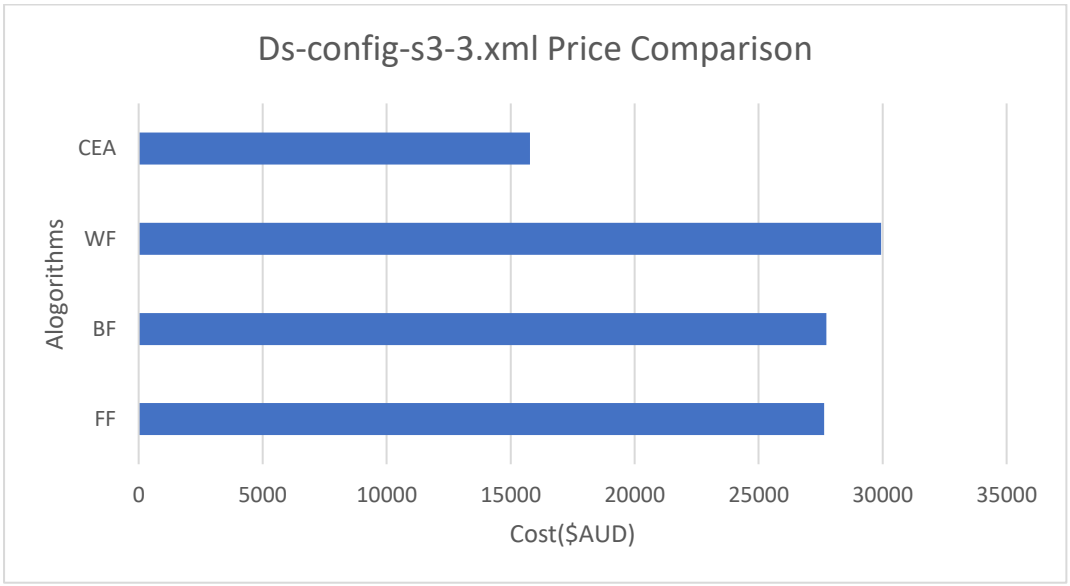
The second ArrayList contains all the current server types. This is done by using the hash set data structure previously mentioned, in which all server types are added hash set in order to avoid duplicate server types, and then from there added to the ArrayList.

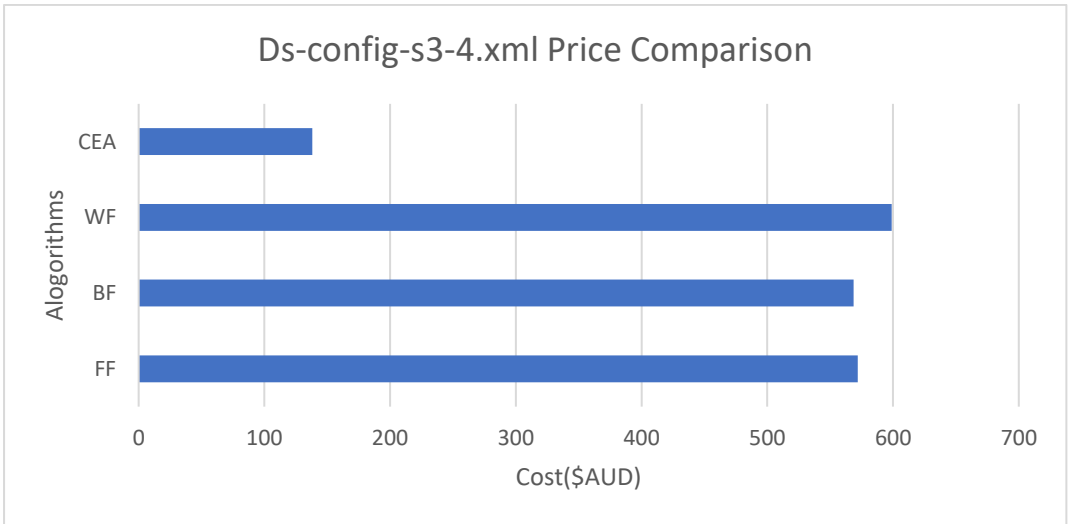
Evaluation:

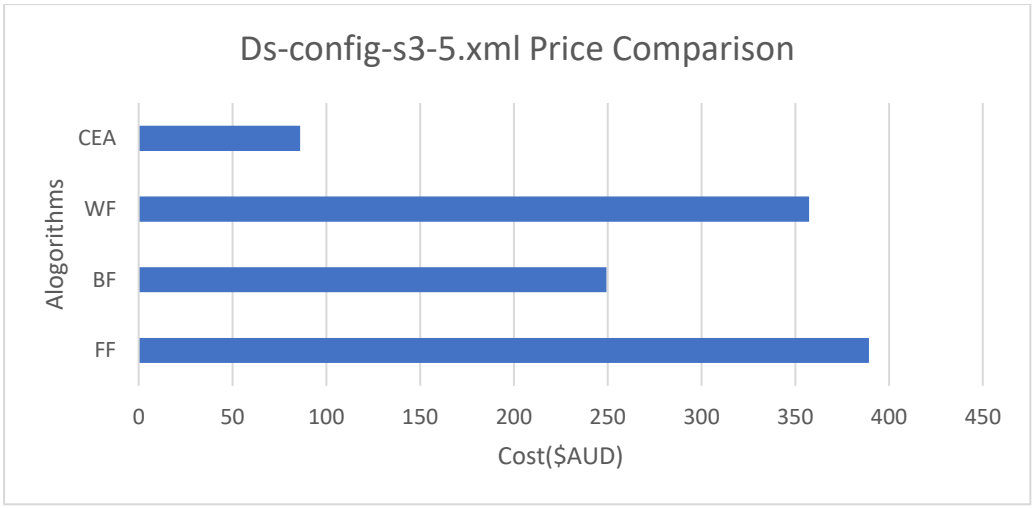
As is evident from the tables and charts provided both above and below, the new implementation of the cost-efficient algorithm runs far cheaper than all base line algorithms. When you run all these algorithms against each other, the cost-efficient algorithm comes in at about 50% cheaper. The comparisons regarding, the average waiting time, and average execution time show that due to the nature of the cost-efficient algorithm, the average waiting time has been upped by a considerable amount while the average execution times idle around the same mark as the baseline algorithms. In terms of waiting time, the cost-efficient algorithm doesn't compare due to always getting the cheapest server. However, considering the price reduction this is totally fair. Please refer below to a total visual comparison between algorithms and configuration files 1 through 7 regarding the cost, average execution time, and avg waiting time.

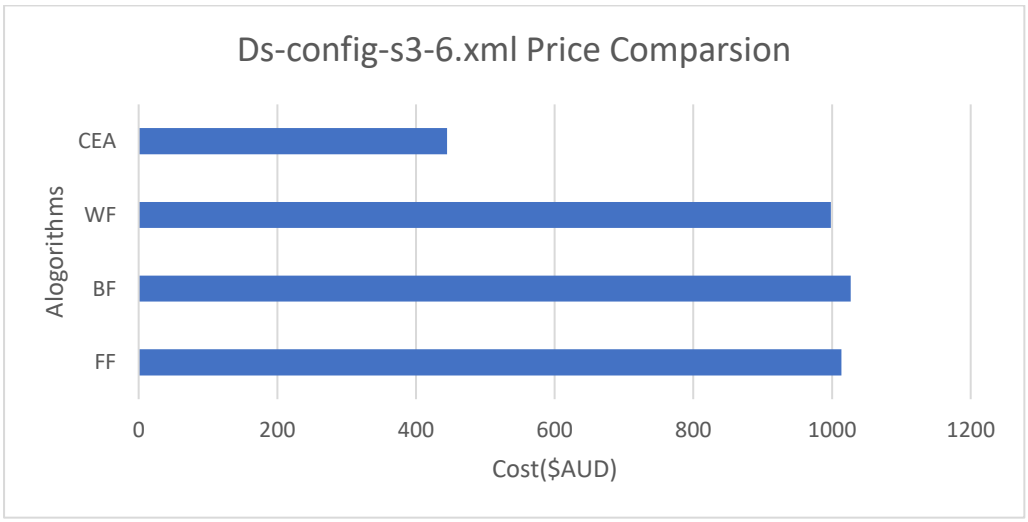


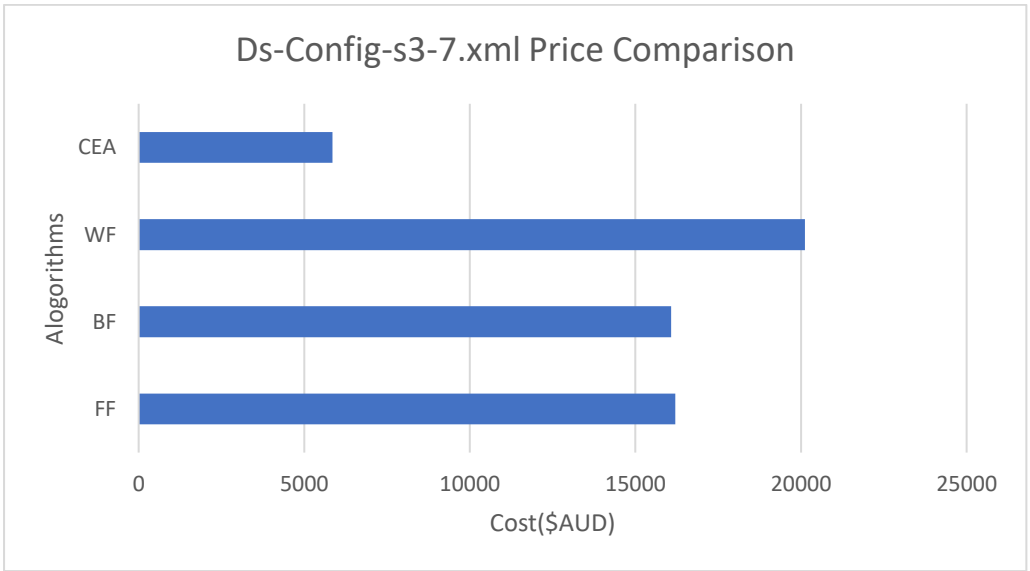

Ds-config-s3-2.xml Price Comparison
Alogorithms
CEA
WF
BF
FF
0
500
1000
1500
2000
2500
3000
Cost($AUD)


Ds-config-s3-3.xml Price Comparison
Alogorithms
CEA
WF
BF
FF
0
5000
10000
15000
20000
25000
30000
35000
Cost($AUD)

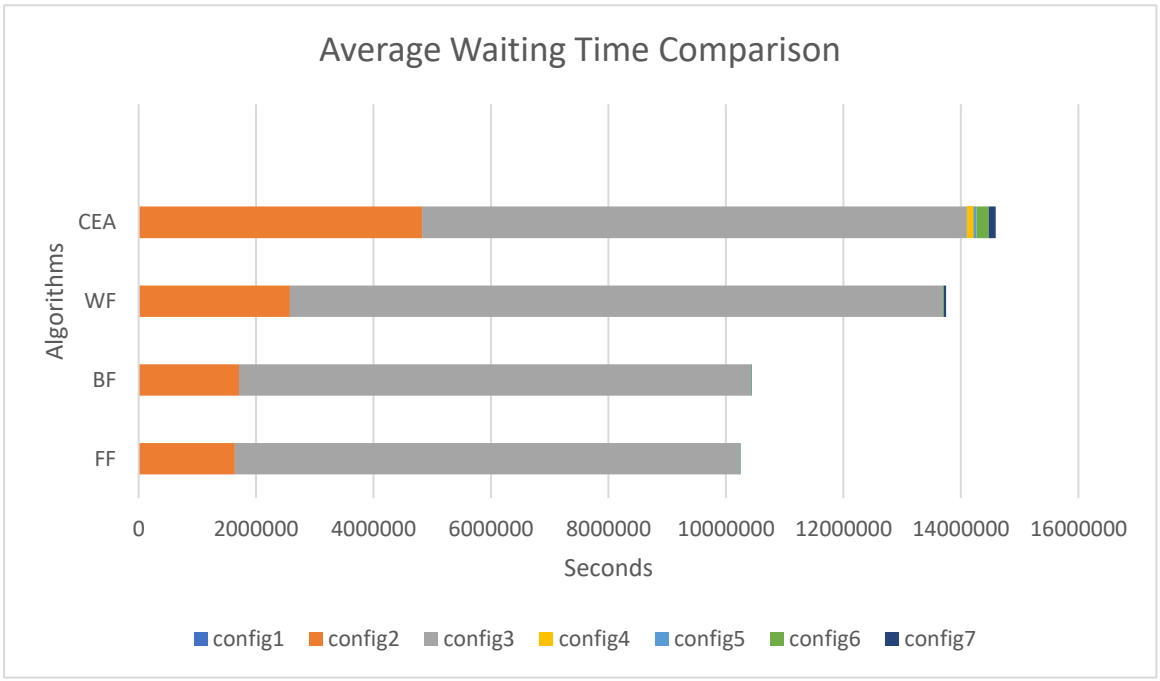

Ds-config-s3-4.xml Price Comparison
Alogorithms
CEA
WF
BF
FF
0
100
200
300
400
500
600
700
Cost($AUD)


Ds-config-s3-5.xml Price Comparison
Alogorithms
CEA
WF
BF
FF
0
50
100
150
200
250
300
350
400
450
Cost($AUD)


Ds-config-s3-6.xml Price Comparsion
Alogorithms
CEA
WF
BF
FF
0
200
400
600
800
1000
1200
Cost($AUD)


Ds-Config-s3-7.xml Price Comparison
Alogorithms
CEA
WF
BF
FF
0
5000
10000
15000
20000
25000
Cost($AUD)


Average Execution Time Comparison
Algorithms
CEA
WF
BF
FF
0
10000
20000
30000
40000
50000
60000
70000
80000
Seconds
config1
config2
config3
config4
config5
config6
config7


Average Waiting Time Comparison
Algorithms
CEA
WF
BF
FF
0
2000000
4000000
6000000
8000000
10000000
12000000
14000000
16000000
Seconds
config1
config2
config3
config4
config5
config6
config7

Conclusion:

After checking all the comparisons between all the configuration files for all the algorithms, it's clear that as a baseline cost efficient algorithm the results speak for themselves as there is a cost cut of about 50% total for all configuration files, while some particular configuration files cuts up to 70-80% in total costs expended. For a new platform or start up these numbers are extremely beneficial as if the appropriate entity did not have the capital to use on renting servers and also the steep costs of running, they wouldn't be able to continue running their particular service. Luckily through the implementation of this algorithm, the previously mentioned entity would not have to worry about wasting an enormous sum on renting and running servers. I think the best-case long term with the selected algorithms is to run it for cost performance and slowly increase into other algorithms when profit margins are steady, and the entity is not losing a huge chunk of their capital towards running servers.

References:

https://github.com/upotudrop/COMP3100GP

Demo instructions:

1. Extract submission folder
2. Go to submission folder
3. Run `tar -xvf ds-sim.tar`
4. Run `mv ds-server /…/Stage3Submission`, Note path may be different on different machines.
5. Run 'mv ds-client /…/Stage3Submission`, similarly.
6. To compile code, open the 'comp3100stage3' folder
7. And run the commands
8. Javac comp3100stage3/*.java
9. Java comp3100stage3/Client -a [CEA] -n while the server is running