Name : Ashwin Panditrao Jadhav

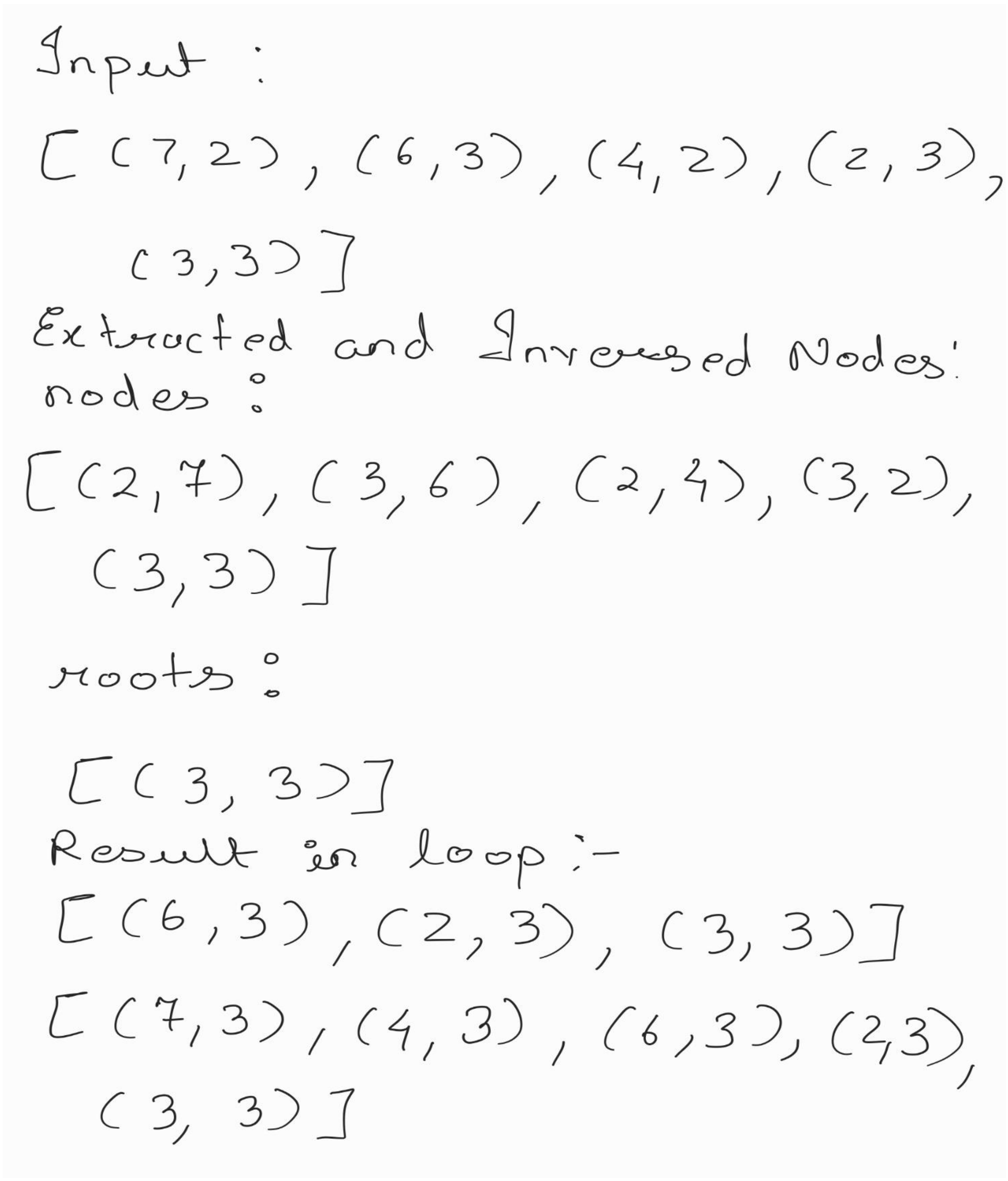
UB Name : ajadhav5

UB Number : 50405435

Algorithm

1. The algorithm first creates a Spark Session and then reads the data from a file through a system argument.
2. Then using map() function the data (u,v) read from the file is then split into u and v by a space using split() function.
3. Then the algorithm filter outs the roots from the data set and save roots in RDD roots when the a adjacent nodes (u,v) are same.
4. Now, the algorithm uses the mapped data from an RDD nodes and reduces it to map the roots which are connected to their respective roots.
5. To do so, the algorithm runs in a infinitely running while loop with a breaking condition of previous nodes count equals to the new nodes counts which contains the nodes mapped to their particular root.
6. Hence, the loop runs and maps the connected components to the root until the resultant RDD nodes1 has same number nodes.
7. Once the loop completes, the algorithm reformats the data and then saves the connected components with their respective root to an output folder.

Example :



Input :

[(7, 2), (6, 3), (4, 2), (2, 3),

(3, 3)]

Extracted and Inversed Nodes:

nodes :

[(2, 7), (3, 6), (2, 4), (3, 2),

(3, 3)]

roots :

[(3, 3)]

Result in loop :-

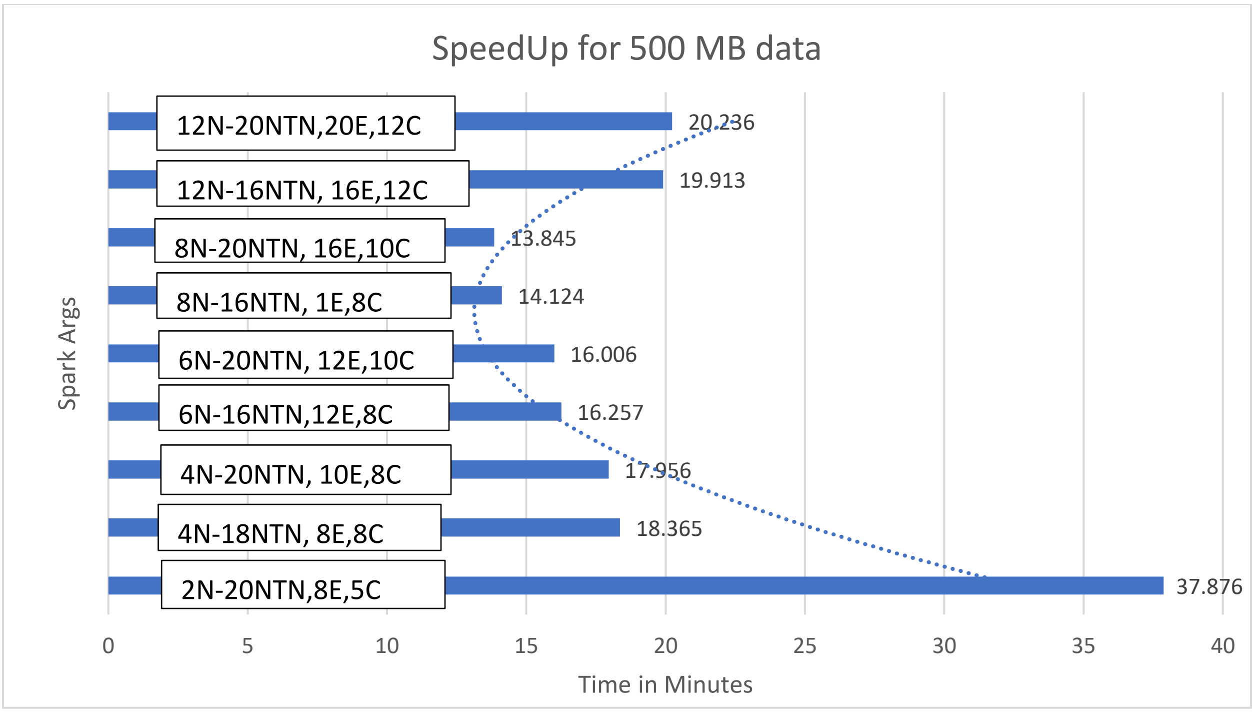
[(6, 3), (2, 3), (3, 3)]

[(7, 3), (4, 3), (6, 3), (2, 3),

(3, 3)]

Graph, Speed Up and Efficiency :

Sppeed up for 500 Mb of data and 29999997 Million Edges



Where :

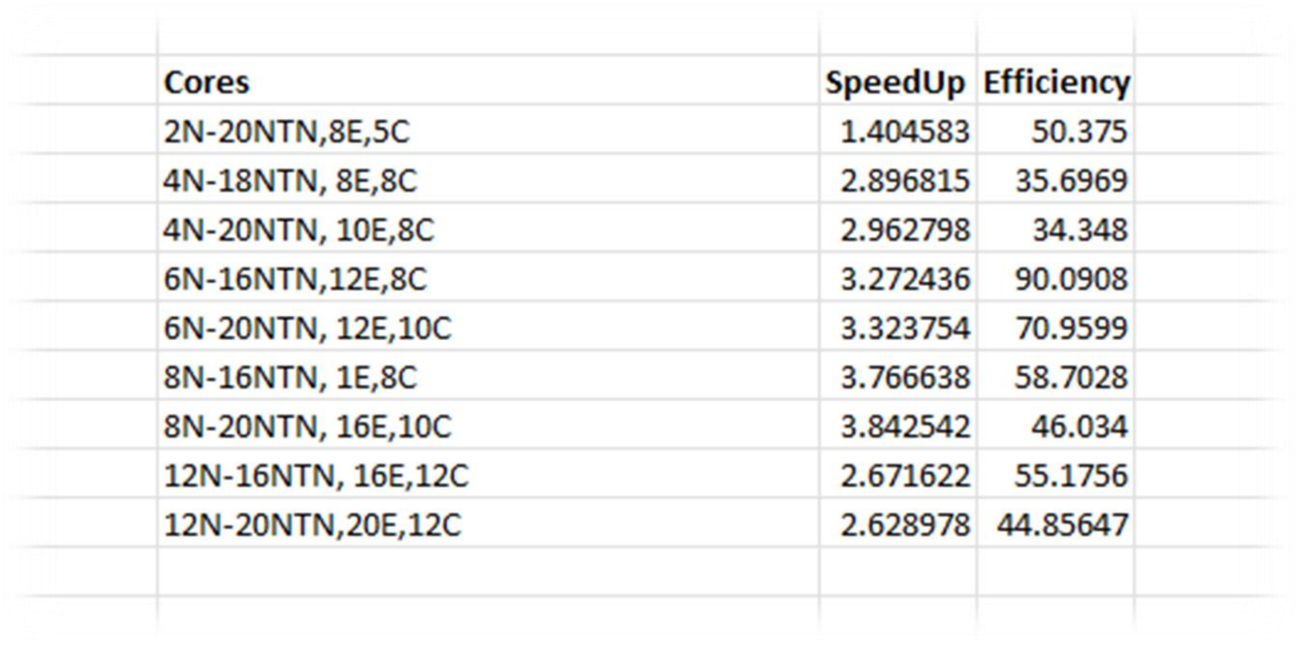
N – Number of Nodes

NTN – Ntasks per nodes

E – Number of Executors

C – Executor Cores

SpeedUp and Efficiency :



Cores	SpeedUp	Efficiency
2N-20NTN,8E,5C	1.404583	50.375
4N-18NTN, 8E,8C	2.896815	35.6969
4N-20NTN, 10E,8C	2.962798	34.348
6N-16NTN,12E,8C	3.272436	90.0908
6N-20NTN, 12E,10C	3.323754	70.9599
8N-16NTN, 1E,8C	3.766638	58.7028
8N-20NTN, 16E,10C	3.842542	46.034
12N-16NTN, 16E,12C	2.671622	55.1756
12N-20NTN,20E,12C	2.628978	44.85647

Scaling :

Strong Scaling:

Yes, the code shows strong scaling as when the number of cores are increased while keeping the same amount of data, the execution time also decreases. This shows that the code is strongly scalable.

Analysis :

The code is not ideally scalable on very large clusters but shows strong scaling when the number processors increased is not large.

Whereas the code runs best when the cores count is smaller and doesn’t cross over 100 which are distributed over 8 or more nodes.

Is it always scalable?

No, the code is not always scalable. As when the number of processors when increased in larger numbers the code losses its strong scalability as well as speed up.

If it is scalable in what sense?

The code scales very well for 2 to 8 cores. It very efficiently shows weak scaling even for large data sets but when the number of processors become very large like 32 or 64 cores or more, the code losses it’s weak scaling.

Can the performance be increased?

Yes, the performance can be significantly increased. As I have used join() function which itself is a very expensive function. If somehow my code can avoid join() function and take out roots of connected components using less expensive functions, then the speedup as well as efficiency which is ultimately performance can be increased.