Algorithms for Massive Datasets Project:

Link Analysis

Jana Trisha Tanchan Go - V12228

I. Dataset

The Amazon Book Reviews dataset on Kaggle contains data about 3 million book reviews over 212,404 unique books. The dataset has two CSV files: Book Ratings and Book Details.

Books_rating	
Attribute	**Description**
Id	The Id of Book
Title	Book Title
Price	The price of Book
User_id	Id of user who rate the book
profileName	Name of user who rate the book
review/helpfulness	helpfulness rating of the review, e.g. 2/3
review/score	rating from 0 to 5 for the book
review/time	time of the given review
review/summary	the summary of text review
review/text	the full text of a review

Books_data	
Attribute	**Description**
Title	Book title
description	description of book
authors	Name of book authors
image	url for book cover
previewLink	link to access this book on Google books
publisher	Name of the publisher
publishedDate	the date of publish
infoLink	link to get more information about the book on Google books
categories	genres of books
ratingsCount	averaging rating for book

Data Dictionary for Amazon Book Reviews Dataset

The Kaggle dataset (Version 1) was last accessed on 16 May 2025 for this project. This dataset was last updated three years ago.

II. Data Organization and Pre-processing

The book ID, book title, and user ID attributes were selected from the Book Ratings file to perform link analysis. Rows with missing values in the book title and user ID attributes were dropped. The entities to be ranked are books, which are defined by constructing a book-book graph wherein each node is a book and edges are formed from a pair of books reviewed by the same user. For example, if a user reviews books A, B, and C, the following edges will be formed: (A,B), (A,C), (B,C), (B,A), (C,A), (C,B).

The following code snippets are how the graphs were created for each implementation of the PageRank algorithm. For both Pyspark and vanilla Python implementations, a set of books is created based on the books reviewed by a user. Combinations of two unique books are then created from the set and are stored in an adjacency list. The NetworkX implementation has a built-in function to create the graph which takes in a parameter that contains the edges of the graph.

```python
# create graph for pyspark impl (using RDD)
sample_rdd = sample_df.rdd.map(lambda row: (row['User_id'], row['Id'])).aggregateByKey(set(), lambda acc, x: acc | {x}, lambda a, b: a | b)
book_pairs_rdd = sample_rdd.flatMap(lambda row: combinations(row[1], 2))
# create adjacency list
book_edges_rdd = book_pairs_rdd.flatMap(lambda pair: [(pair[0], pair[1]), (pair[1], pair[0])]).distinct()
books_adj_list_rdd = book_edges_rdd.groupByKey().mapValues(list)
books_adj_list_rdd = books_adj_list_rdd.cache()
```

Pyspark implementation for creating the undirected book-book graph

```python
# create graph edges for vanilla python impl
book_edges = set()
for books in user_books:
    book_edges.update(combinations(books, 2))
# create adjacency list for the graph edges
books_adj_list = defaultdict(set)
for book1, book2 in book_edges:
    books_adj_list[book1].add(book2)
    books_adj_list[book2].add(book1)
```

Vanilla Python implementation for creating the undirected book-book graph

```
# create graph for networkX
graph_nx = nx.Graph()
graph_nx.add_edges_from(book_edges)
```

NetworkX implementation for creating the undirected book-book graph

III. Considered Algorithms and Implementations

The PageRank algorithm is used to rank the books in the graph via power iteration in order to approximate the limiting distribution of a random surfer in a Markov chain. The final PageRank distribution is determined when the maximum number of iterations has reached or when the L1 norm of the current and the previous page rank falls below a threshold of 1×10^{-6} . To address the possibility of dead ends and spider traps, taxation is implemented using a damping factor $\beta = 0.85$. The resulting power iteration formula is as follows:

$$v' = \beta M v + \frac{(1 - \beta) e}{n}$$

Where,

v' is the new vector estimate of PageRanks
v is the current vector estimate of PageRanks
M is the transition matrix
β is the damping factor
n is the number of nodes in the graph
e is a vector of all 1's with length equal to n

The term $\beta M v$ models the probability that the random surfer follows an out-link from the current page, while the term $(1 - \beta) e/n$ represents the random surfer jumping to a random page with probability of the complement of the damping factor $(1 - \beta)$.

A PySpark implementation of the algorithm was implemented to ensure scalability for large datasets. Additionally, two other implementations are explored to compare performance in terms of runtimes across increasing subsamples of the dataset, which will be discussed more in detail under the *Experiment Description* section.

Other algorithms considered were topic-sensitive PageRank and Hyper-link Induced Topic Search (HITS) Index. However, including these specialised algorithms would make the project scope too large when the primary investigation is more on performance and scalability. Focusing on the default PageRank algorithm would offer a good baseline in identifying the trade-offs between different implementations, especially with large datasets.

A. PySpark Implementation

For the PySpark implementation, a series of map and reduce functions are utilized to calculate PageRank in a distributed environment. The graph is represented as a resilient distributed dataset (RDD) of adjacency lists and the ranks are initialised uniformly. With each iteration, the terms of the power iteration formula are computed through the use of the flatMap(), reduceByKey(), and mapValues() functions.

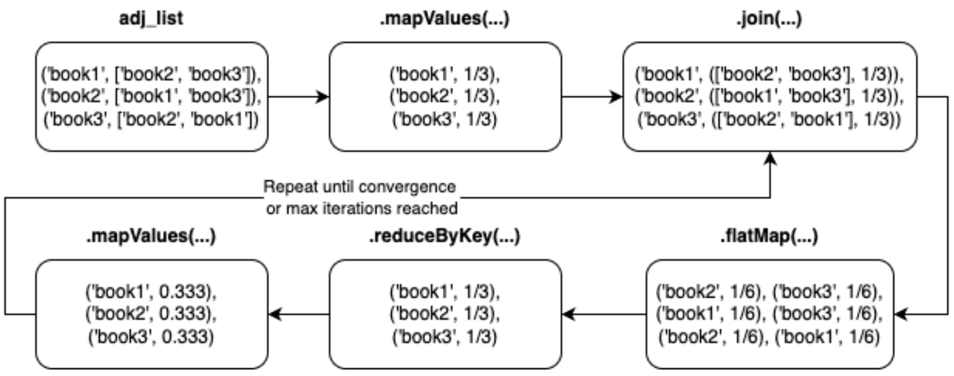
```
def pagerank_pyspark(adj_list, max_iter=25, damping=0.85, tol=1e-6):
    # initialize ranks
    N = adj_list.count()
    ranks_rdd = adj_list.mapValues(lambda _ : 1/N)

    # power iteration
    for _ in range(max_iter):
        adj_ranks = adj_list.join(ranks_rdd) # (book, ([neighbors], current_rank))
        contribution = adj_ranks.flatMap(lambda x: [(neighbor, x[1][1]/len(x[1][0])) for neighbor in x[1][0]]) # (neighbor, contribution)
        new_ranks_rdd = contribution.reduceByKey(lambda x, y: x+y).mapValues(lambda x: (1 - damping) / N + damping * x) # (neighbor, new_rank)

        # early stopping if the difference is smaller than the threshold (using L1 norm)
        diff = ranks_rdd.join(new_ranks_rdd).mapValues(lambda x: abs(x[0] - x[1])).values().sum()
        if diff < tol:
            break

        ranks_rdd = new_ranks_rdd
    return dict(ranks_rdd.collect())
```

PySpark implementation for the PageRank algorithm


adj_list
('book1', ['book2', 'book3']),
('book2', ['book1', 'book3']),
('book3', ['book2', 'book1'])
.mapValues(...)
('book1', 1/3),
('book2', 1/3),
('book3', 1/3)
.join(...)
('book1', (['book2', 'book3'], 1/3)),
('book2', (['book1', 'book3'], 1/3)),
('book3', (['book2', 'book1'], 1/3))
Repeat until convergence
or max iterations reached
.mapValues(...)
('book1', 0.333),
('book2', 0.333),
('book3', 0.333)
.reduceByKey(...)
('book1', 1/3),
('book2', 1/3),
('book3', 1/3)
.flatMap(...)
('book2', 1/6), ('book3', 1/6),
('book1', 1/6), ('book3', 1/6),
('book2', 1/6), ('book1', 1/6)

Visualisation of the RDDs throughout the MapReduce operations in the algorithm

B. *Vanilla Python Implementation*

For this implementation, simple Pandas dataframes and Python operations are used to construct the graph and to compute the ranks. The logic closely follows the PySpark implementation, but without the presence of RDDs and MapReduce functions, making it unable to run in parallel in a distributed environment.

```
def pagerank_vanilla_python(adj_list, max_iter=25, damping=0.85, tol=1e-6):
    # initialize ranks
    N = len(adj_list)
    ranks = {book: 1 / N for book in adj_list}

    # power iteration
    for _ in range(max_iter):
        new_ranks = defaultdict(float)
        for book, neighbors in adj_list.items():
            contribution = ranks[book] / len(neighbors)
            for neighbor in neighbors:
                new_ranks[neighbor] += contribution

        for book in adj_list:
            new_ranks[book] = (1 - damping) / N + damping * new_ranks[book]

        # early stopping if the difference is smaller than the threshold (using L1 norm)
        diff = sum(abs(new_ranks[book] - ranks[book]) for book in adj_list)
        if diff < tol:
            break

        ranks = new_ranks
    return ranks
```

Vanilla Python implementation for the PageRank algorithm

C. *NetworkX Implementation*

The Pagerank function from the NetworkX library was called to compare the performance of a built-in function from a library versus a *from scratch* implementation.

```
result = nx.pagerank(graph_nx)
```

NetworkX implementation for the PageRank algorithm

IV. Solution Scalability

To ensure that the PageRank implementation scales up with data size, Spark is utilized to perform MapReduce functions and enable the distribution of tasks to different compute nodes together with the use of RDDs. Through RDDs, the graph data and rank values can be partitioned across multiple machines and processed in parallel, which would greatly reduce the processing time for large datasets. Although this implementation was tested within the constraints of Google Colab, which only provides a single compute node, this implementation should theoretically scale up in a distributed system such as Spark clusters.

Moreover, Spark's directed acyclic graph (DAG) execution and lazy evaluation contribute to its efficiency. The DAG captures the sequence of transformations to be executed eventually, but not immediately. Lazy evaluation ensures that Spark only executes the operations once an action is called, enabling Spark to optimise the DAG before execution. These techniques improve performance and memory efficiency by removing unnecessary steps and optimising the flow of transformations.

V. Experiment Description

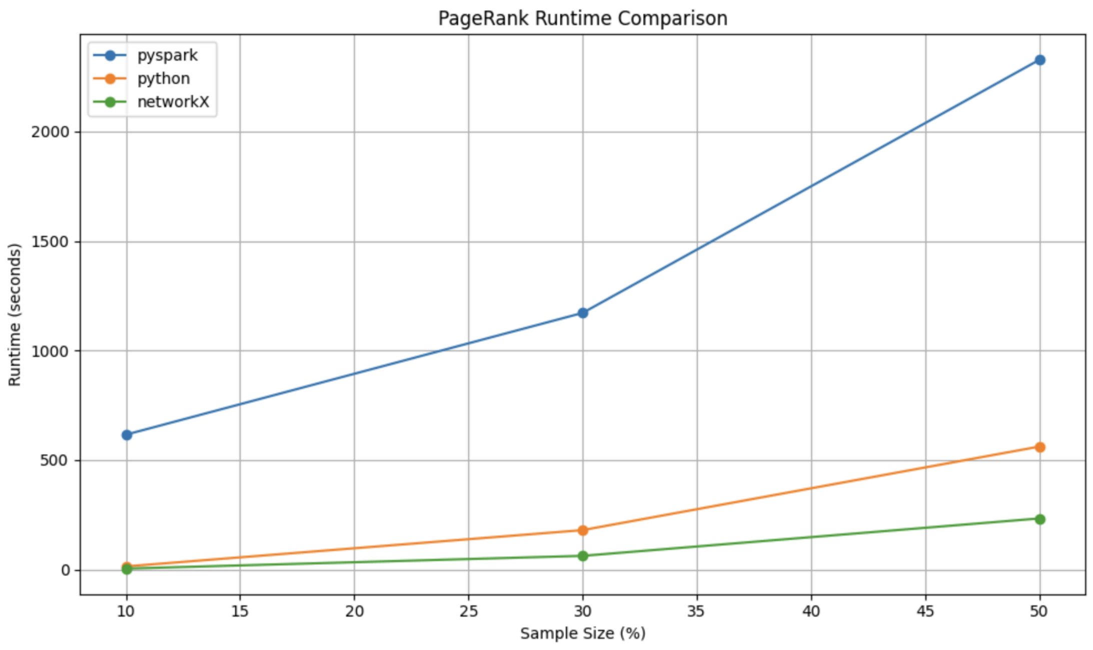
To evaluate performance, the runtime of PySpark, vanilla Python, and NetworkX implementations is measured across increasing subsamples of the dataset. More specifically, subsamples consisting of 10%, 30%, and 50% of the full dataset are used to measure each implementation's runtime and to evaluate how it scales with larger datasets. For replicability, the random seed is set to 42 when creating each subsample. In addition to the runtime comparison, the top 5 ranked books for each implementation are printed out to qualitatively evaluate any differences in the top book titles and their final PageRank values.

VI. Discussion and Comments on Experimental Results

Upon running all three implementations for all subsample sizes, the runtimes for each implementation were plotted into a graph to visualize the results. At a glance, the NetworkX implementation outperforms both PySpark and vanilla python implementations, which is particularly more evident in the 50% subsample. The fast runtimes of NetworkX could be attributed to the fact that the library's function has been optimised to use more efficient data structures and operations.

The vanilla Python implementation follows closely behind where its runtime is slightly slower than NetworkX at the 10% sample, but the increase in runtime was steeper as the sample size grew. Nonetheless, the vanilla Python implementation still performs at a decent speed, processing the 50% subsample a little over 500 seconds. Inefficient code logic like nested loops and unoptimised data structures could be the reason for a slower performance compared to NetworkX.

Interestingly, the PySpark implementation exhibited the longest runtimes across all sample sizes, with a runtime of over 10 minutes with the 10% subsample and nearly 40 minutes with the 50% subsample. The most probable cause for such underperformance is the overhead costs associated with Spark, especially in a single-node environment like Google Colab. Spark's processing capabilities would be more apparent when running on a distributed system and with much larger, massive datasets.



Graph of runtimes for the three implementations with different sample sizes

Implementation	Sample Size		
	10%	**30%**	**50%**
PySpark	615.7985418	1171.912172	2327.884283
Vanilla Python	14.12634277	179.9652486	561.6489651
NetworkX	4.177074194	62.44371128	233.3365169

Runtimes of the three implementations with different sample sizes (seconds)

Regarding result accuracy, the top 5 ranked results for the three implementations are quite similar, with PySpark and vanilla Python being identical and NetworkX having a few differences (highlighted in the table below). The difference in results for NetworkX could be attributed to small differences in the implementation of the algorithm. Upon reading NetworkX's documentation, specifically its PageRank function, the most apparent difference is the early stopping condition. While both PySpark and vanilla Python implementations terminate the algorithm once the L1 norm is less than the tolerance threshold, NetworkX scales the threshold by the number of nodes $L1\, distance\ <\ N * tol$. This variation in the stopping condition allows for earlier convergence especially for larger graphs, which explains the slight difference in ranks for NetworkX from the other two implementations.

	50.0% of the full dataset:					
	PySpark		**Vanilla Python**		**NetworkX**	
	Book	**Rank**	**Book**	**Rank**	**Book**	**Rank**
1	Harry Potter and The Sorcerer's Stone	0.000307	Harry Potter and The Sorcerer's Stone	0.000307	Harry Potter and The Sorcerer's Stone	0.000336
2	The Catcher in the Rye	0.000268	The Catcher in the Rye	0.000268	Blink: The Power of Thinking Without Thinking	0.000287
3	The Catcher in the Rye [Audiobook] [Cd] [Unab	0.000266	The Catcher in the Rye [Audiobook] [Cd] [Unab	0.000266	The Catcher in the Rye	0.000272
4	Blink: The Power of Thinking Without Thinking	0.000258	Blink: The Power of Thinking Without Thinking	0.000258	The Catcher in the Rye [Audiobook] [Cd] [Unab	0.000268
5	THE CATCHER IN THE RYE	0.000254	THE CATCHER IN THE RYE	0.000254	THE CATCHER IN THE RYE	0.000259

Top 5 ranked books for each implementation in the 50% subsample dataset

VII. Conclusion and Future Work

These findings demonstrate that NetworkX is the most efficient implementation for this given dataset, due to its optimised operations and data structures. However, the PySpark implementation may have potential in scalability, specifically when dealing with datasets that will exceed a single machine's memory.

The vanilla Python and NetworkX implementation works decently for the current dataset, but would probably fail at larger-than-memory datasets since both implementations are constrained by the machine's memory capacity, are not designed for distributed environments, and are unable to parallelise workloads across multiple machines.

Future work could include running the PySpark implementation on a distributed environment to assess the true performance capabilities of Spark. Additionally, the current implementation could be extended to handle topic-sensitive PageRank, which could be determined by metadata like book genres, author, or other relevant attributes to provide a more tailored result in ranking.

VIII. References

Rajaraman, A., & Ullman, J. D. (2014). *Mining of massive datasets* (2nd ed.). Cambridge University Press.

Bakhet, M. (n.d.). *Amazon books reviews* [Data set]. Kaggle. https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews

Byte Size Data Science. (2018, December 11). 011-Introduction to spark [Video]. YouTube. https://www.youtube.com/watch?v=moPoOakVVSQ

Byte Size Data Science. (2018, December 18). 012-Spark RDDS [Video]. YouTube. https://www.youtube.com/watch?v=nH6C9vqtyYU

NetworkX developers. (n.d.). *NetworkX documentation*. https://networkx.org/documentation

Lao, P. (2022, April 25). *Why Spark isn't always the best choice for not-so-big data*. Medium. https://medium.com/@pined.lao/why-spark-isnt-always-the-best-choice-for-not-so-big-data-f7b888c3ce59