**ICT337**

**Big Data Computing in the Cloud**

**July 2024**

**ECA**

|  |  |
| --- | --- |
| Name: | YANG XIAN WEI SHAWN |
| T-Group | T01 |
| Date Submitted | 30 October 2024 Wednesday |

**(Full marks: 100)**

**Question 1**

**Question 1(a) (6 marks)**

**ANS:**

The Spark job execution process in a cluster environment involves several steps.

1. **Spark Job Submission**

The first step involves a user submitting an application to Spark using the terminal command “spark-submit”. When this command is executed, a Spark job would be initialized, and the command would also launch a driver program that is responsible for ensuring that the main functions of the application are executed.

The driver program will define a “SparkContext” object which is stored in the driver program and its purpose is to establish a connection with the Cluster Manager to provide computing resources to the respective worker nodes. A user could configure the Cluster Manager so that it would be configured as the Spark’s standalone Cluster Manager or Mesos, YARN, or Kubernetes by specifying a terminal command.

Once a connection has been established between the “SparkContext” object and the Cluster Manager, the “SparkContext” object would be able to connect with the Worker Nodes who would be able to store and process data.

1. **SparkContext initialization**

The “SparkContext” object which is defined and is stored in the driver program, performs several tasks such as connecting with a Cluster Manager and connecting with Worker Nodes through the Cluster Manager to acquire Executors which are stored in the Worker Nodes.

The executors represent processes that perform computations and store data for active applications that are currently running.

1. **Application Code Distribution**

The driver program would send application code that is defined by the Spark JAR folder which represents a Java Archive folder which stores Spark library files or Python files that are stored in the “SparkContext” object, would be sent to the Executors in the Worker Nodes.

This action ensures that each executor is equipped with necessary source codes to perform their respective assigned tasks. The “SparkContext” object would then send tasks via the Cluster Manager to each executor to execute the tasks.

1. **DAG Creation**

When a user executes an action on the Resilient Distributed Dataset (RDD), this would trigger an execution plan that is represented by a Directed Acyclic Graph (DAG) which would be sent to a cluster. Subsequently, the DAG Scheduler would allocate tasks to different Worker Nodes that are located in the cluster.

The DAG Scheduler purpose is to monitor RDDs and create a minimum schedule to execute a job.

1. **Stage Scheduling**

The DAG Scheduler would divide the DAG into task stages and each stage stores tasks based on partitioning of input data. The task stages are then sent to the Task Scheduler for execution and the Task Scheduler executes the task stages by the Cluster Manager.

The Task Scheduler purpose is to execute the tasks in the executors.

1. **Task Execution**

The Task Scheduler get the set of tasks sent by the DAG Scheduler for each task stage and its purpose is to upload the tasks that are stored in the task stages to the cluster, execute the tasks and retry the execution of the tasks if there are any execution failures.

1. **Fault Tolerance**

Spark achieves fault tolerance through the adoption of RDDs which is a Spark user-facing API which is an immutable distributed datasets that are partitioned by Worker Nodes in a cluster.

The Spark RDDs maintains the lineage of transformations utilized by datasets so that in case of an executor failure, Spark would be able to perform data recovery by recomputing the failed RDD partition by using the lineage of transformations.

1. **Result Retrieval**

Once all tasks within a stage has completed its execution, the results would be transmitted back to the “SparkContext” object that is stored in the Driver Program and subsequently, the “SparkContext” object would transform the results by aggregation to display the final results.

**Question 1(b) (4 marks)**

**ANS:**

The Directed Acyclic Graph (DAG) plays an important role in the Spark framework because for a Spark job to be executed in a cluster, Spark would need to partition the job into small independent tasks that will be executed in parallel.

DAG purpose is to provide a Spark job a logical execution plan by partitioning the job into a sequence of stages and where each stage stores tasks that would be executed independently of other tasks and in parallel.

The DAG enables Spark to perform several types of optimizations for example, pipelining, task reordering and removing unutilized operations to enhance the job execution efficiency.

The DAG partitions jobs into stages and tasks which enables Spark to execute tasks in parallel and allocate them into a cluster of machines for more efficient execution of large processing jobs.

**Question 1(c) (10 marks)**

**ANS:**

**Spark Resilient Distributed Datasets (RDDs):**

**Concept:**

RDDs represent the original API for Apache Spark and RDDs are a immutable set of data objects that are distributed across Worker Nodes within a Spark Cluster of machines.

RDDs enables parallel processing by partitioning a job into small independent tasks and provides fault tolerance by being immutable which enables RDDs to avoid data corruption by creating a new data object from the existing failed RDD partition instead of overwriting it. RDDs also provides efficient performance of Spark jobs within a cluster of machines by executing in-built memory computations.

**Structure:**

RDDs represent an immutable set of data objects, and these are either Scala or Java objects that can process structured and unstructured data but do not have predefined schemas which means that users would have to define the schemas.

**API:**

RDDs provide an OOP-style API that can perform low level transformations, actions and persistence operations. Transformations represent operations performed on RDDs to return a new RDD which for example includes several functions such as map(), filter(), flatMap(), groupByKey(), reduceByKey and join().

Actions represent operations that will either send a Spark job results to the Driver Program or write the results to an external storage location which for example includes several functions such as collect(), count(), take() and reduce().

Persistence represents operations that enables users to persist or cache an RDD in the in-built memory which is a useful operation when you want to reuse an RDD on several occasions which for example includes functions such as persist() and cache().

**Optimization:**

Spark RDDs do not have built-in optimization engine which means that users would have to manually optimize code performance.

**Use Cases:**

Spark RDDs would be the preferable option for use cases where users need to have fine-grained control over their datasets and need to perform low-level transformation and action operations on their datasets.

Spark RDDs should also be used for other uses cases such as when users need to work with unstructured data for example media streams and streams of text as well as in uses cases where users don’t need to create a schema that applies a format such as a columnar format in a Spark cluster of machines.

**DataFrames:**

**Concept:**

Spark DataFrames are immutable distributed sets of data which is similar to Spark RDDs however, Spark DataFrames format data into named columns which is similar to a table that is stored in a relational database. Spark DataFrames provide higher-level abstraction by enabling users to apply a structure to immutable distributed sets of data.

Spark DataFrames provides users with a domain specific language (DSL) API which enables users to manipulate their datasets using DataFrames and opens Spark to a large audience.

**Structure:**

Spark DataFrames are designed to manage structured and semi-structured data such as CSV, JSON and Parquet files because they have a schema which means that a Spark DataFrame would format data into 2-dimensional table of rows and columns where the columns that have a name and type as well as rows which represent a single record for more efficient data organization.

**API:**

Spark DataFrames have built-in APIs that provides schema support which enables them to automatically infer a predefined schema when processing structured data so that when the DataFrame is defined, it would have a structure that resembles a SQL table. Users would be able to use the schema to execute query operations on the DataFrame to manipulate the data.

Spark DataFrames also offer users a domain specific language (DSL) API which is used for data manipulation that enables users to perform SQL-like queries that involves several types of query operations such as joining, filtering and aggregation.

**Optimization:**

Spark DataFrames can use Spark’s Catalyst Optimizer to enhance query execution plans to increase performance efficiency by analysing DataFrame query operations followed by producing optimized code.

The Spark’s Catalyst Optimizer includes optimizations such as predicated pushdown, column pruning and advanced code generation. The Spark’s Catalyst Optimizer main purpose is to allow users to add new Spark DataFrame optimization solutions and features to Spark SQL as well as extend the Catalyst Optimizer.

**Use Cases:**

Spark DataFrames would be ideal use cases that require a schema to specified and used for processing of structured data or semi-structured data from specific sources such as CSV, JSON and MySQL using higher level abstractions.

Spark DataFrames would also be ideal in use cases that involve data analysis where users need to perform handling of missing values and transformations operations such as groupBy(), agg() and pivot() for the purpose of data exploration and cleaning to quickly identify data distributions.

**Question 2**

**Question 2(a) (6 marks)**

**ANS:**

|  |
| --- |
| #Import libraries  from pyspark.sql import SparkSession  from pyspark.sql import functions as f  from pyspark.sql.functions import \*  from functools import reduce  from pyspark.sql.functions import avg, desc, round  from pyspark.sql.types import IntegerType, DoubleType, DateType, NumericType  from pyspark.sql.functions import to\_date  from pyspark.sql.functions import lower, trim, when  from pyspark.sql.functions import sum, col  from pyspark.sql.functions import sum as spark\_sum, col  from pyspark.sql.functions import col, avg, format\_number  import matplotlib.pyplot as plt  import seaborn as sns  import pandas as pd  import numpy as np  """ Set the SPARK\_LOCAL\_IP environment variable: Before running your script, set this environment variable: """  import os  os.environ['SPARK\_LOCAL\_IP'] = 'localhost'  # Start spark session  """ Set the spark.driver.bindAddress: Add the following configuration to your SparkSession builder:  Use a specific port: If the issue persists, try specifying a port explicitly: """  spark = SparkSession \  .builder \  .appName("ICT337 ECA July 2024 Semester Question 2") \  .config("spark.driver.bindAddress", "localhost") \  .config("spark.driver.port", "4043") \  .config("spark.some.config.option", "some-value") \  .getOrCreate()  """ Question 2(a) (6 marks) """  print("\nQuestion 2(a) (6 marks)\n")  # Read csv file and store into dataframe  airbnb\_data\_PySpark\_DF = spark\  .read\  .option("inferSchema", "true")\  .option("header", "true")\  .csv("/Users/shawnyang/Downloads/ICT337 ECA July 2024 Semester/ECA Datasets/airbnb\_data.csv")  """  Sample airbnb\_data.csv data:  id,name,host\_id,host\_name,neighbourhood\_group,neighbourhood,latitude,longitude,room\_type,price,minimum\_nights,number\_of\_reviews,last\_review,reviews\_per\_month,calculated\_host\_listings\_count,availability\_365  """  # Show the type of the object to check if its a dataframe  print(f"Show the type of the Airbnb Data DataFrame:\n{type(airbnb\_data\_PySpark\_DF)}\n")  print("Show the content of airbnb data:\n")  airbnb\_data\_PySpark\_DF.show(20)  """ To change the data types of the columns in your airbnb\_data\_PySpark\_DF based on their content,  you can use the following PySpark code:  This code will change the data types as follows:  Integer columns: id, host\_id, minimum\_nights, number\_of\_reviews, calculated\_host\_listings\_count  Double (decimal) columns: latitude, longitude, price, reviews\_per\_month  Date column: last\_review  The other columns (name, host\_name, neighbourhood\_group, neighbourhood, room\_type) will remain as strings. """  airbnb\_data\_PySpark\_DF = airbnb\_data\_PySpark\_DF\  .withColumn("id", airbnb\_data\_PySpark\_DF["id"].cast(IntegerType())) \  .withColumn("host\_id", airbnb\_data\_PySpark\_DF["host\_id"].cast(IntegerType())) \  .withColumn("latitude", airbnb\_data\_PySpark\_DF["latitude"].cast(DoubleType())) \  .withColumn("longitude", airbnb\_data\_PySpark\_DF["longitude"].cast(DoubleType())) \  .withColumn("price", airbnb\_data\_PySpark\_DF["price"].cast(DoubleType())) \  .withColumn("minimum\_nights", airbnb\_data\_PySpark\_DF["minimum\_nights"].cast(IntegerType())) \  .withColumn("number\_of\_reviews", airbnb\_data\_PySpark\_DF["number\_of\_reviews"].cast(IntegerType())) \  .withColumn("last\_review", to\_date(airbnb\_data\_PySpark\_DF["last\_review"])) \  .withColumn("reviews\_per\_month", airbnb\_data\_PySpark\_DF["reviews\_per\_month"].cast(DoubleType())) \  .withColumn("calculated\_host\_listings\_count", airbnb\_data\_PySpark\_DF["calculated\_host\_listings\_count"].cast(IntegerType()))  print("Display the schema of the Airbnb Data DataFrame:\n")  """ This shows the structure of the DataFrame, including column names and their data types. """  airbnb\_data\_PySpark\_DF.printSchema()  # Show the dimensions (number of rows and columns) of the DataFrame  """ .count() and len(airbnb\_data\_DataFrame.columns): These get the number of rows and columns in the DataFrame. """  print(f"Airbnb Data DataFrame Dimensions (number of rows and columns):\n({airbnb\_data\_PySpark\_DF.count()}, {len(airbnb\_data\_PySpark\_DF.columns)})\n")  # Find missing data  print("Check for missing data in each column and display the count:\n")  # Initialize an empty list to store column expressions  airbnb\_data\_column\_Expressions = []  # Iterate through each column in the DataFrame  for airbnb\_data\_column in airbnb\_data\_PySpark\_DF.columns:  # Create an expression to count null values for the current column  null\_count\_for\_column\_Expression = count(when(col(airbnb\_data\_column).isNull(), airbnb\_data\_column)).alias(airbnb\_data\_column)  # Add the expression to the list  airbnb\_data\_column\_Expressions.append(null\_count\_for\_column\_Expression)  # Use select() with the list of expressions to create the missing\_data DataFrame  missing\_data\_PySpark\_DF = airbnb\_data\_PySpark\_DF.select(airbnb\_data\_column\_Expressions)  missing\_data\_PySpark\_DF.show()  """ This code selects only the "room\_type" column from the DataFrame. """  distinct\_room\_types\_PySpark\_DF = airbnb\_data\_PySpark\_DF.select("room\_type").distinct().collect()  """ This code will show you all the distinct values in the room\_type column and count them. """  print(f"\nTotal number of distinct room types: {len(distinct\_room\_types\_PySpark\_DF)}\n")  """ To clean up the room\_type column and ensure you only have the three expected values,  you can use a combination of string manipulation and filtering. """  airbnb\_data\_with\_distinct\_room\_types\_PySpark\_DF = airbnb\_data\_PySpark\_DF.withColumn(  "room\_type",  when(  lower(trim("room\_type")).isin(["private room", "shared room", "entire home/apt"]),  lower(trim("room\_type"))  )\  .otherwise("unknown")  )  """ This code selects only the "room\_type" column from the DataFrame. """  distinct\_room\_types = airbnb\_data\_with\_distinct\_room\_types\_PySpark\_DF.select("room\_type").distinct().collect()  print("Distinct room types after cleaning:")  """ It then iterates through each row in the distinct\_room\_types list  For each row, it prints the value of the "room\_type" column """  for row in distinct\_room\_types:  print(row["room\_type"])  """ After printing all distinct room types, it prints the total count of distinct room types using len(distinct\_room\_types) """  print(f"\nTotal number of distinct room types after cleaning: {len(distinct\_room\_types)}\n")  # Count total rows before dropping missing values  total\_rows\_before\_dropping\_missing\_values\_PySpark\_DF = airbnb\_data\_with\_distinct\_room\_types\_PySpark\_DF.count()  print(f"Show the number of rows before dropping missing values: {total\_rows\_before\_dropping\_missing\_values\_PySpark\_DF}\n")  # Drop rows with missing values  """ Create a new DataFrame with missing values removed. """  airbnb\_data\_cleaned\_PySpark\_DF = airbnb\_data\_with\_distinct\_room\_types\_PySpark\_DF.dropna()  # Count total rows after dropping missing values  total\_rows\_after\_dropping\_missing\_values\_PySpark\_DF = airbnb\_data\_cleaned\_PySpark\_DF.count()  print(f"Show the number of rows after dropping missing values: {total\_rows\_after\_dropping\_missing\_values\_PySpark\_DF}\n")  # Calculate the number of rows dropped  """ Calculate and display the number of rows that were dropped. """  total\_rows\_dropped\_PySpark\_DF = total\_rows\_before\_dropping\_missing\_values\_PySpark\_DF - total\_rows\_after\_dropping\_missing\_values\_PySpark\_DF  print(f"Number of rows that were dropped: {total\_rows\_dropped\_PySpark\_DF}\n")  # Show a few rows of the cleaned DataFrame  print("Show a sample of the cleaned DataFrame:\n")  airbnb\_data\_cleaned\_PySpark\_DF.show(20)  """ Calculate basic statistics for all columns  This code will generate a summary of basic statistics for all columns in your DataFrame, including:"""  basic\_stats\_PySpark\_DF = airbnb\_data\_cleaned\_PySpark\_DF.describe()  # Show the results  print("Basic statistics for each column:\n")  basic\_stats\_PySpark\_DF.show() |

**Output:**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA black rectangular object with white text

Description automatically generatedA screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

**Question 2(b) (6 marks)**

**ANS:**

|  |
| --- |
| """ Question 2(b) (6 marks) """  print("\nQuestion 2(b) (6 marks)\n")  """ Group by neighbourhood\_group and calculate average price  Order the results by average price in descending order  Limit the output to the top 10 results """  neighbourhood\_group\_average\_price\_PySpark\_DF = airbnb\_data\_cleaned\_PySpark\_DF.groupBy("neighbourhood\_group") \  .agg(round(avg("price"), 2).alias("average\_price")) \  .orderBy(desc("average\_price")) \  .limit(10)  print("Show the top 10 Airbnb's neighbourhood\_group sorted from highest to lowest average price:\n")  # Show the results  neighbourhood\_group\_average\_price\_PySpark\_DF.show()  """ Convert the Spark DataFrame to a Pandas DataFrame using toPandas(). """  neighbourhood\_group\_average\_price\_Pandas\_DF = neighbourhood\_group\_average\_price\_PySpark\_DF.toPandas()  # Display the Pandas DataFrame  print(f"Display the Pandas DataFrame:\n\n{neighbourhood\_group\_average\_price\_Pandas\_DF}")  """ Set up the plot size. """  plt.figure(figsize=(12, 6))  """ Set up the plot style. """  sns.set\_style("whitegrid")  """ Create a bar plot using seaborn's barplot function. """  ax = sns.barplot(x='neighbourhood\_group', y='average\_price', data=neighbourhood\_group\_average\_price\_Pandas\_DF)  """ Customize the plot with a title. """  plt.title('Top 10 Airbnb Neighbourhood Groups by Average Price ($)', fontsize=16)  """ Customize the plot with labels. """  plt.xlabel('Neighbourhood Group', fontsize=12)  plt.ylabel('Average Price ($)', fontsize=12)  """ Customize the plot with rotated x-axis labels for better readability. """  plt.xticks(rotation=45, ha='right')  """ Add value labels on top of each bar for precise price information. """  for i, v in enumerate(neighbourhood\_group\_average\_price\_Pandas\_DF['average\_price']):  ax.text(i, v, f'${v:.2f}', ha='center', va='bottom')  """ Adjust the layout of the plot """  plt.tight\_layout()  """ Display the plot. """  plt.show()  """ Group the data by neighbourhood and Calculate the average price for each neighbourhood  Order the results by average price in descending order  Limit the output to the top 10 results """  neighbourhood\_average\_price\_PySpark\_DF = airbnb\_data\_cleaned\_PySpark\_DF.groupBy("neighbourhood") \  .agg(round(avg("price"), 2).alias("average\_price")) \  .orderBy(desc("average\_price")) \  .limit(10)  print("Show the top 10 Airbnb's neighbourhood sorted from highest to lowest average price:\n")  # Show the results  neighbourhood\_average\_price\_PySpark\_DF.show()  """ Convert the Spark DataFrame to a Pandas DataFrame using toPandas(). """  neighbourhood\_average\_price\_Pandas\_DF = neighbourhood\_average\_price\_PySpark\_DF.toPandas()  # Display the Pandas DataFrame  print(f"Display the Pandas DataFrame:\n\n{neighbourhood\_average\_price\_Pandas\_DF}")  """ Set up the plot size. """  plt.figure(figsize=(12, 6))  """ Set up the plot style. """  sns.set\_style("whitegrid")  """ Create a bar plot using seaborn's barplot function. """  ax = sns.barplot(x='neighbourhood', y='average\_price', data=neighbourhood\_average\_price\_Pandas\_DF.head(10))  """ Customize the plot with a title. """  plt.title('Top 10 Airbnb Neighbourhoods by Average Price ($)', fontsize=16)  """ Customize the plot with labels. """  plt.xlabel('Neighbourhood', fontsize=12)  plt.ylabel('Average Price ($)', fontsize=12)  """ Customize the plot with rotated x-axis labels for better readability. """  plt.xticks(rotation=45, ha='right')  """ Add value labels on top of each bar for precise price information. """  for i, v in enumerate(neighbourhood\_average\_price\_Pandas\_DF['average\_price'][:10]):  ax.text(i, v, f'${v:.2f}', ha='center', va='bottom')  """ Adjust the layout of the plot """  plt.tight\_layout()  """ Display the plot. """  plt.show() |

**Output:**

**A screen shot of a computer

Description automatically generated**

**A graph showing a number of people

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A graph showing the number of neighborhoods

Description automatically generated**

**Top 10 Airbnb Neighbourhood Groups by Average Price Bar Graph Insights:**

1st Insight:

The 1st insight is that the Manhatan neighourhood group is the most expensive neighbourhood group because it has the highest average price of $180.19.

This insight also may mean that the Manhatan neighourhood group is the more luxurious option that caters to wealthier customers or business customers.

2nd Insight:

The 2nd insight is that the Bronx neighourhood group is the least expensive neighbourhood group because it has the lowest average price of $79.75.

This insight also may mean that the Bronx neighourhood group is the most affordable option that caters to budget-conscious customers or customers who cannot afford the more expensive alternatives.

**Top 10 Airbnb Neighbourhoods by Average Price Bar Graph Insights:**

1st Insight:

The 1st insight is that the most expensive neighbourhood is Sea Gate which has an average price of $601.67.

The 2nd most expensive neighbourhood is Tribeca which has an average price of $460.30

The least expensive neighbourhood is Greenwhich Village which has an average price of $238.85.

This insight means that there is a large variation in average prices across the various neighbourhoods.

2nd Insight:

The 2nd insight is the neighbourhoods like NoHo, Flatiron District, SoHo, Neponsit, Midtown, West Village, Willowbrook and Greenwich Village have average prices ranging from $301.25 to $238.85 which suggest a small average price variation between them.

This insight also means that there are more afforadble neighbourhood options than expensive neighbourhood options.

**Question 2(c) (6 marks)**

**ANS:**

|  |
| --- |
| """ Question 2(c) (6 marks) """  print("\nQuestion 2(c) (6 marks)\n")  # Show a few rows of the cleaned DataFrame  print("Show a sample of the cleaned DataFrame:\n")  airbnb\_data\_cleaned\_PySpark\_DF.show(10)  """ Creates a new DataFrame with unique host\_ids and their total number of reviews. """  airbnb\_data\_cleaned\_unique\_hosts\_total\_reviews\_PySpark\_DF = airbnb\_data\_cleaned\_PySpark\_DF\  .groupBy("host\_id") \  .agg(spark\_sum("number\_of\_reviews")\  .alias("total\_reviews\_across\_each\_hosts"))  print("Updated DataFrame with unique hosts and their total reviews:\n")  # Show the results  airbnb\_data\_cleaned\_unique\_hosts\_total\_reviews\_PySpark\_DF.show(10)  """ Shows the top 10 unique hosts by total reviews """  airbnb\_data\_cleaned\_unique\_hosts\_total\_reviews\_PySpark\_DF = airbnb\_data\_cleaned\_unique\_hosts\_total\_reviews\_PySpark\_DF.orderBy(  col("total\_reviews\_across\_each\_hosts").desc()  )  # Show the updated DataFrame  print("Updated DataFrame with the top 10 unique hosts by total reviews:\n")  airbnb\_data\_cleaned\_unique\_hosts\_total\_reviews\_PySpark\_DF.show(10)  """ Calculates the new total number of reviews across all unique hosts """  total\_Num\_Of\_Reviews\_Across\_All\_Hosts = airbnb\_data\_cleaned\_unique\_hosts\_total\_reviews\_PySpark\_DF\  .agg(spark\_sum("total\_reviews\_across\_each\_hosts"))\  .collect()[0][0]  print(f"New total number of reviews across all unique hosts: {total\_Num\_Of\_Reviews\_Across\_All\_Hosts}\n")  """ Calculate the popularity index for each host by adding a new column "popularity\_index" to the DataFrame.  Which is calculated as (total\_reviews\_across\_each\_hosts / new\_total\_reviews) \* 100. """  airbnb\_data\_with\_popularity\_index\_PySpark\_DF = airbnb\_data\_cleaned\_unique\_hosts\_total\_reviews\_PySpark\_DF.withColumn(  "popularity\_index\_(%)",  (col("total\_reviews\_across\_each\_hosts") / total\_Num\_Of\_Reviews\_Across\_All\_Hosts) \* 100  )  print("Show each unique host, their total reviews and popularity index (%):\n")  """ Show the 1st 10 rows of the relevant columns to verify the new column has been added correctly. """  airbnb\_data\_with\_popularity\_index\_PySpark\_DF.show(10)  # Join the DataFrames  """ Joins the original DataFrame with the popularity index DataFrame on the "host\_id" column."""  combined\_PySpark\_DF = airbnb\_data\_cleaned\_PySpark\_DF.join(airbnb\_data\_with\_popularity\_index\_PySpark\_DF, "host\_id")  print("Show the combined DataFrame:\n")  combined\_PySpark\_DF.show(10)  """ This will display the hosts with the highest popularity index first,  giving you a quick view of the most popular hosts based on their share of total reviews. """  host\_by\_popularity\_PySpark\_DF = airbnb\_data\_with\_popularity\_index\_PySpark\_DF.select(  "host\_id",  format\_number("popularity\_index\_(%)", 3)\  .alias("popularity\_index\_(%)")  ) \  .orderBy(col("popularity\_index\_(%)").desc())  print("Top Ten (10) most popular host with the structure (i.e., [host\_id, popularity\_index]):\n")  # Show the results  host\_by\_popularity\_PySpark\_DF.show(10, truncate=False)  """ Creates a new DataFrame by selecting two specific columns from the combined\_df DataFrame:  "neighbourhood": This column contains the names of different neighborhoods.  "popularity\_index\_(%)": This column contains the popularity index for each neighborhood. """  neighborhood\_by\_popularity\_PySpark\_DF = combined\_PySpark\_DF.select("neighbourhood", "popularity\_index\_(%)")  print("Show the most popular neighbourhood by popularity DataFrame:\n")  neighborhood\_by\_popularity\_PySpark\_DF.show(10, truncate=False)  print("Show the neighbourhoods by average popularity DataFrame:\n")  """ Group the data by neighborhood and calculate the average popularity index for each neighborhood. """  neighborhood\_by\_average\_popularity\_PySpark\_DF = neighborhood\_by\_popularity\_PySpark\_DF.groupBy("neighbourhood") \  .agg(avg("popularity\_index\_(%)").alias("avg\_popularity\_index\_(%)"))  neighborhood\_by\_average\_popularity\_PySpark\_DF.show(10, truncate=False)  """ Sort the neighborhoods by their average popularity index in descending order and select the top 10 rows. """  sorted\_neighborhood\_by\_average\_popularity\_PySpark\_DF = neighborhood\_by\_average\_popularity\_PySpark\_DF \  .withColumn("avg\_popularity\_index\_(%)", round(col("avg\_popularity\_index\_(%)"), 3)) \  .orderBy(col("avg\_popularity\_index\_(%)").desc()) \  .limit(10)  # Show the results  print("Top 10 Most Popular Neighborhoods based on Host's Popularity Index:")  sorted\_neighborhood\_by\_average\_popularity\_PySpark\_DF.show(10, truncate=False)  """ Converts the Spark DataFrame to a Pandas DataFrame for easier plotting """  sorted\_neighborhood\_by\_average\_popularity\_Pandas\_DF = sorted\_neighborhood\_by\_average\_popularity\_PySpark\_DF.toPandas()  # Display the Pandas DataFrame  print(f"Display the Pandas DataFrame:\n\n{sorted\_neighborhood\_by\_average\_popularity\_Pandas\_DF}")  # Set up the plot size  plt.figure(figsize=(12, 6))  # Set up the plot style  sns.set\_style("whitegrid")  # Create a bar plot  ax = sns.barplot(x='neighbourhood', y='avg\_popularity\_index\_(%)', data=sorted\_neighborhood\_by\_average\_popularity\_Pandas\_DF)  """ Customize the plot with a title. """  plt.title('Top 10 Most Popular Neighborhoods by Average Host Popularity Index', fontsize=16)  """ Customize the plot with labels. """  plt.xlabel('Neighborhood', fontsize=12)  plt.ylabel('Average Popularity Index (%)', fontsize=12)  """ Customize the plot with rotated x-axis labels for better readability. """  plt.xticks(rotation=45, ha='right')  """ Adds value labels on top of each bar for precise percentage information. """  for i, v in enumerate(sorted\_neighborhood\_by\_average\_popularity\_Pandas\_DF['avg\_popularity\_index\_(%)']):  ax.text(i, v, f'{v:.3f}%', ha='center', va='bottom')  # Adjust the layout  plt.tight\_layout()  # Display the plot  plt.show() |

**Output:**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA black screen with white text

Description automatically generated**

**A graph with blue rectangles

Description automatically generated**

**Top 10 Most Popular Neighbourhoods by Average Host Popularity Index Bar Graph Insights:**

1st Insight:

The 1st insight is that the Financial District is the most popular neighbourhood with the highest average host popularity index of about 0.040% while East Morrisania is the least popular neighbourhood with the lowest average host popularity index of about 0.012%.

2nd Insight:

The 2nd insight is that the popularity of the neighbourhoods began to decrease gradually from Tompkinsville to East Morrisania with the average host popularity index ranging from about 0.018% to 0.012%.

**Question 2(d) (6 marks)**

**ANS:**

|  |
| --- |
| """ Question 2(d) (6 marks) """  print("\nQuestion 2(d) (6 marks)\n")  print("Show the combined DataFrame:\n")  combined\_PySpark\_DF.show(10)  """ Select the "room\_type" column from the combined DataFrame.  Use the distinct() function to get unique values. """  distinct\_room\_types\_PySpark\_DF = combined\_PySpark\_DF\  .select("room\_type")\  .distinct()  # Show the results  print("Available room types:\n")  distinct\_room\_types\_PySpark\_DF.show()  """ Get a list of distinct room types from the previously created DataFrame. """  """ It creates an empty list room\_types """  room\_types = []  """ Then iterates through each row in the DataFrame,  appending the 'room\_type' value from each row to the list. """  for room\_type\_row in distinct\_room\_types\_PySpark\_DF.collect():  room\_types.append(room\_type\_row['room\_type'])  print(f"Distinct room types: {room\_types}\n")  """ Create a new DataFrame that calculates the average price for each combination of neighbourhood and room type. """  average\_price\_by\_neighbourhood\_room\_type\_PySpark\_DF = combined\_PySpark\_DF\  .groupBy("neighbourhood", "room\_type")\  .agg(  round(avg("price"), 2).alias("avg\_price")  )  print("DataFrame with average price for each neighbourhood and room type:\n")  average\_price\_by\_neighbourhood\_room\_type\_PySpark\_DF.show(10)  """ Pivots the DataFrame to create columns for each room type. """  average\_price\_by\_neighbourhood\_room\_type\_pivoted\_PySpark\_DF = average\_price\_by\_neighbourhood\_room\_type\_PySpark\_DF\  .groupBy("neighbourhood")\  .pivot("room\_type")\  .agg(  round(avg("avg\_price"), 2)  ).na.fill(0)  print("Pivot the DataFrame to create columns for each room type:\n")  average\_price\_by\_neighbourhood\_room\_type\_pivoted\_PySpark\_DF.show(10)  """ Sort the results by neighbourhood name in ascending order """  sorted\_average\_price\_by\_neighbourhood\_room\_type\_pivoted\_PySpark\_DF = average\_price\_by\_neighbourhood\_room\_type\_pivoted\_PySpark\_DF.orderBy("neighbourhood")  # Show the results  print("Average price for each neighbourhood and room type sorted by neighbourhood in acscending order:\n")  sorted\_average\_price\_by\_neighbourhood\_room\_type\_pivoted\_PySpark\_DF.show()  """ Get the top 20 neighborhoods by total average price. """  top\_neighborhoods\_by\_neighborhoods\_and\_room\_types\_PySpark\_DF = sorted\_average\_price\_by\_neighbourhood\_room\_type\_pivoted\_PySpark\_DF\  .select(  "neighbourhood",  (col("Entire home/apt") + col("Private room") + col("Shared room")).alias("total\_avg\_price")    )\  .orderBy("total\_avg\_price", ascending=False)\  .limit(20)  print("Show the ranking of the top 20 neighborhoods based on their total average price across all room types\n")  top\_neighborhoods\_by\_neighborhoods\_and\_room\_types\_PySpark\_DF.show()  """ Filters and joins DataFrames to create a new DataFrame containing detailed information about the top 20 neighborhoods. """  top\_20\_neighborhoods\_data\_PySpark\_DF = sorted\_average\_price\_by\_neighbourhood\_room\_type\_pivoted\_PySpark\_DF.join(top\_neighborhoods\_by\_neighborhoods\_and\_room\_types\_PySpark\_DF, "neighbourhood")  print("Filter the original DataFrame to include only the top 20 neighborhoods:\n")  top\_20\_neighborhoods\_data\_PySpark\_DF.show(20, truncate=False)  """ Convert to Pandas DataFrame for easier plotting """  top\_20\_neighborhoods\_data\_Pandas\_DF = top\_20\_neighborhoods\_data\_PySpark\_DF.toPandas()  print("Check the actual column names in your Pandas DataFrame:\n")  print(top\_20\_neighborhoods\_data\_Pandas\_DF.columns)  """ Use the pandas melt() function to reshape the DataFrame from wide to long format. """  top\_20\_neighborhoods\_data\_Melted\_Pandas\_DF = pd.melt(  top\_20\_neighborhoods\_data\_Pandas\_DF,  id\_vars=['neighbourhood'],  value\_vars=['entire home/apt', 'private room', 'shared room'],  var\_name='room\_type',  value\_name='avg\_price'  )  print("\nMelt the DataFrame to long format\n")  print(top\_20\_neighborhoods\_data\_Melted\_Pandas\_DF)  # Create the plot  # Set up the plot size  plt.figure(figsize=(15, 10))  """ Create a bar plot using the seaborn library. """  sns.barplot(x='neighbourhood', y='avg\_price', hue='room\_type', data=top\_20\_neighborhoods\_data\_Melted\_Pandas\_DF)  # Customize the plot  """ Customize the plot with a title. """  plt.title('Average Price by Room Type for Top 20 Neighbourhoods', fontsize=16)  """ Customize the plot with labels. """  plt.xlabel('Neighbourhood', fontsize=12)  plt.ylabel('Average Price ($)', fontsize=12)  """ Customize the plot with rotated x-axis labels for better readability. """  plt.xticks(rotation=45, ha='right')  """ Customize the plot with a legend. """  plt.legend(title='Room Type')  # Adjust the layout  plt.tight\_layout()  # Display the plot  plt.show() |

**Output:**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA black screen with white text

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA black rectangle with white text

Description automatically generated**

**A graph of different colored bars

Description automatically generated**

**Average Price by Room Type for Top 20 Neighbourhoods Side-By-Side Bar Chart Insights:**

1st Insight:

The 1st insight is room type entire home/apartment has consistently the highest average prices which makes it the most expensive room type across the neighbourhoods.

2nd Insight:

The 2nd insight is that the neighbourhood Sea Gate has the highest average price for the the room type entire home/apartment and the Sea Gate average price for the the room type entire home/apartment is much higher compared to other neighbourhoods.

**Question 2(e) (16 marks)**

**ANS:**

|  |
| --- |
| """ Question 2(e) (16 marks) """  print("\nQuestion 2(e) (16 marks)\n")  print("Show the combined DataFrame:\n")  combined\_PySpark\_DF.show(10)  """ Calculate the average price for each combination of neighbourhood group and room type. """  sorted\_average\_price\_by\_neighbourhood\_group\_and\_room\_type\_PySpark\_DF = combined\_PySpark\_DF\  .groupBy("neighbourhood\_group", "room\_type") \  .agg(  round(avg("price"), 2)\  .alias("average\_price")  ) \  .orderBy(  col("average\_price").desc()  )  # Show the results  print("Average price for each neighbourhood\_group and room type, sorted from highest to lowest:\n")  sorted\_average\_price\_by\_neighbourhood\_group\_and\_room\_type\_PySpark\_DF.show()  """ Convert to Pandas DataFrame for easier plotting """  sorted\_average\_price\_by\_neighbourhood\_group\_and\_room\_type\_Pandas\_DF = sorted\_average\_price\_by\_neighbourhood\_group\_and\_room\_type\_PySpark\_DF.toPandas()  # Display the Pandas DataFrame  print(f"Display the Pandas DataFrame:\n\n{sorted\_average\_price\_by\_neighbourhood\_group\_and\_room\_type\_Pandas\_DF}")  # Set up the plot  plt.figure(figsize=(12, 8))  """ Create a bar plot using the seaborn library. """  ax = sns.barplot(x='neighbourhood\_group', y='average\_price', hue='room\_type', data=sorted\_average\_price\_by\_neighbourhood\_group\_and\_room\_type\_Pandas\_DF)  # Customize the plot  """ Customize the plot with a title. """  plt.title('Average Price by Neighbourhood Group and Room Type', fontsize=16)  """ Customize the plot with labels. """  plt.xlabel('Neighbourhood Group', fontsize=12)  plt.ylabel('Average Price ($)', fontsize=12)  """ Customize the plot with rotated x-axis labels for better readability. """  plt.xticks(rotation=45)  """ Customize the plot with a legend. """  plt.legend(title='Room Type')  """ Add value labels on top of each bar. """  for container in ax.containers:  ax.bar\_label(container, fmt='${:.2f}', label\_type='edge')  """ Adjusts the y-axis limit. """  plt.ylim(0, plt.ylim()[1] \* 1.1)  # Adjust the layout  plt.tight\_layout()  # Display the plot  plt.show()    """ Count the number of listings for each neighbourhood group. """  host\_listings\_by\_neighbourhood\_group\_PySpark\_DF = combined\_PySpark\_DF\  .groupBy("neighbourhood\_group") \  .agg(  count("id")  .alias("total\_listings")  ) \  .orderBy(  col("total\_listings")\  .desc()  )  # Show the results  print("Total number of host listings in each neighbourhood group, sorted from highest to lowest:\n")  host\_listings\_by\_neighbourhood\_group\_PySpark\_DF.show()  """ Convert to Pandas DataFrame for easier manipulation if needed """  host\_listings\_by\_neighbourhood\_group\_Pandas\_DF = host\_listings\_by\_neighbourhood\_group\_PySpark\_DF.toPandas()  # Display the Pandas DataFrame  print(f"Display the Pandas DataFrame:\n\n{host\_listings\_by\_neighbourhood\_group\_Pandas\_DF}")  # Set up the plot  plt.figure(figsize=(10, 6))  # Create the bar plot  sns.barplot(x='neighbourhood\_group', y='total\_listings', data=host\_listings\_by\_neighbourhood\_group\_Pandas\_DF)  # Customize the plot  """ Customize the plot with a title. """  plt.title('Total Host Listings by Neighbourhood Group', fontsize=16)  """ Customize the plot with labels. """  plt.xlabel('Neighbourhood Group', fontsize=12)  plt.ylabel('Total Listings', fontsize=12)  """ Customize the plot with rotated x-axis labels for better readability. """  plt.xticks(rotation=45)  """ Add value labels on top of each bar. """  for i, v in enumerate(host\_listings\_by\_neighbourhood\_group\_Pandas\_DF['total\_listings']):  plt.text(i, v, str(v), ha='center', va='bottom')  # Adjust the layout  plt.tight\_layout()  # Display the plot  plt.show()    """ Group by host\_name and count the number of listings. """  host\_names\_by\_listings\_count\_PySpark\_DF = combined\_PySpark\_DF\  .groupBy("host\_name") \  .agg(  count("id")\  .alias("listing\_count")  ) \  .orderBy(  col("listing\_count")\  .desc()  ) \  .limit(10)  # Show the results  print("Top 10 hosts with the highest number of listings:\n")  host\_names\_by\_listings\_count\_PySpark\_DF.show()  """ Convert to Pandas DataFrame for easier manipulation if needed """  host\_names\_by\_listings\_count\_Pandas\_DF = host\_names\_by\_listings\_count\_PySpark\_DF.toPandas()  # Display the Pandas DataFrame  print(f"Display the Pandas DataFrame:\n\n{host\_names\_by\_listings\_count\_Pandas\_DF}")  # Set up the plot  plt.figure(figsize=(12, 6))  # Create the bar plot  sns.barplot(x='host\_name', y='listing\_count', data=host\_names\_by\_listings\_count\_Pandas\_DF)  # Customize the plot  """ Customize the plot with a title. """  plt.title('Top 10 Hosts with Highest Number of Listings', fontsize=16)  """ Customize the plot with labels. """  plt.xlabel('Host Name', fontsize=12)  plt.ylabel('Number of Listings', fontsize=12)  """ Customize the plot with rotated x-axis labels for better readability. """  plt.xticks(rotation=45, ha='right')  """ Add value labels on top of each bar. """  for i, v in enumerate(host\_names\_by\_listings\_count\_Pandas\_DF['listing\_count']):  plt.text(i, v, str(v), ha='center', va='bottom')  # Adjust the layout  plt.tight\_layout()  # Display the plot  plt.show()    """ Group by host\_name and room\_type, calculate average number of reviews. """  host\_names\_and\_room\_type\_by\_average\_reviews\_PySpark\_DF = combined\_PySpark\_DF\  .groupBy("host\_name", "room\_type") \  .agg(  round(avg("number\_of\_reviews"), 2)\  .alias("avg\_reviews")  ) \  .orderBy(  col("avg\_reviews")\  .desc()  ) \  .limit(20)  # Show the results  print("Top 20 hosts with the highest average number of reviews by room type:\n")  """ Use truncate=False to show the full content of each column. """  host\_names\_and\_room\_type\_by\_average\_reviews\_PySpark\_DF.show(20, truncate=False)  """ Convert to Pandas DataFrame for easier manipulation if needed """  host\_names\_and\_room\_type\_by\_average\_reviews\_Pandas\_DF = host\_names\_and\_room\_type\_by\_average\_reviews\_PySpark\_DF.toPandas()  # Display the Pandas DataFrame  print(f"Display the Pandas DataFrame:\n\n{host\_names\_and\_room\_type\_by\_average\_reviews\_Pandas\_DF}")  # Set up the plot  plt.figure(figsize=(15, 10))  """ Create a bar plot using the seaborn library. """  sns.barplot(x='host\_name', y='avg\_reviews', hue='room\_type', data=host\_names\_and\_room\_type\_by\_average\_reviews\_Pandas\_DF)  # Customize the plot  """ Customize the plot with a title. """  plt.title('Top 20 Hosts with Highest Average Number of Reviews by Room Type', fontsize=16)  """ Customize the plot with labels. """  plt.xlabel('Host Name', fontsize=12)  plt.ylabel('Average Number of Reviews', fontsize=12)  """ Customize the plot with rotated x-axis labels for better readability. """  plt.xticks(rotation=90)  """ Customize the plot with a legend. """  plt.legend(title='Room Type')  """ Add value labels on top of each bar.  The value labels on top of each bar provide the  exact average number of reviews for each host and room type combination.c"""  for i, v in enumerate(host\_names\_and\_room\_type\_by\_average\_reviews\_Pandas\_DF['avg\_reviews']):  plt.text(i, v, f'{v:.2f}', ha='center', va='bottom', fontsize=8)  # Adjust the layout  plt.tight\_layout()  # Display the plot  plt.show()  spark.stop() |

**Output:**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

**A graph of different colored bars

Description automatically generated**

**A screen shot of a computer

Description automatically generated**

**A graph of a number of blue bars

Description automatically generated**

**A black screen with white text

Description automatically generatedA black rectangle with white text

Description automatically generated**

**A graph of blue rectangular bars

Description automatically generated**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

**A graph of different colored bars

Description automatically generated**

**Average Price by Neighbourhood Group and Room Type Side-By-Side Bar Chart Insights:**

1st Insight:

The 1st insight is the room type entire home/apartment consistently had the highest average price across all the neighbourhood groups which means that the room type entire home/apartment was the most expensive room type which suggest a higher premiums for the room type entire home/apartment.

2nd Insight:

The 2nd insight is the room type shared rooms consistently had the lowest average price across all the neighbourhood groups which means that the room type shared rooms was the most affordable room type.

**Total Host Listings by Neighbourhood Group Bar Chart Insights:**

1st Insight:

The 1st insight is the neighbourhood groups Manhatttan and Brooklyn have the most number of Host Listings with 16564 and 16395 listings compared to the other neighbourhood groups which means that Manhattan and Brooklyn have the largest Airbnb Market share.

2nd Insight:

The 2nd insight is the distribution of Airbnb Host Listings data is very unenven across the various neighbourhood groups because Manhattan and Brookyln account for majority of the Airbnb Host Listings while the other neighbourhood groups such as Queens, Bronx and Staten Island only account for a small share of the Airbnb Host Listings.

**Top 10 Hosts with Highest Number of Listings Bar Chart Insights:**

1st Insight:

The 1st insight is the Michael has the highest number of Airbnb Listings at 335 which means that he is the top host and Michael is followed by David who has the 2nd highest number of Airbnb Listings at 308 which means that he is the 2nd most popular host while Anna has the least number of Airbnb Listings at 159 making her the least popular host.

2nd Insight:

The 2nd insight is the 2 hosts Jessica and Daniel both have the same number of Airbnb Listings at 170 and the data distribution of Airbnb Hosts Listings gradually increases from Anna to Michael.

**Top 20 Hosts with Highest Average Number of Reviews by Room Type Bar Chart Insights:**

1st Insight:

The 1st insight is the Host Dora has the highest average number of reviews at 602.50 for the Room Type private room while the Host Welcome To My Place has the least average number of reviews at 330.00 for the Room Type private room.

2nd Insight:

The 2nd insight is that 19 out of 20 of the top Hosts with the Highest Average Number of Reviews which means a mass majority are reviewed by the Room Type's private room and home/aprtments with only 1 host Lloyed is in the top 20 reviewed by the Room Type shared room.

**Question 3**

**Question 3(a) (3 marks)**

**ANS:**

|  |
| --- |
| from pyspark.sql import SparkSession  from pyspark.sql import functions as f  from pyspark.sql.functions import \*  from functools import reduce  from builtins import max as py\_max  import matplotlib.pyplot as plt  import seaborn as sns  import pandas as pd  import numpy as np  import sys  """ Set the SPARK\_LOCAL\_IP environment variable: Before running your script, set this environment variable: """  import os  os.environ['SPARK\_LOCAL\_IP'] = 'localhost'  # Start spark session  """ Set the spark.driver.bindAddress: Add the following configuration to your SparkSession builder:  Use a specific port: If the issue persists, try specifying a port explicitly: """  spark = SparkSession \  .builder \  .appName("ICT337 ECA July 2024 Semester Question 3") \  .config("spark.driver.bindAddress", "localhost") \  .config("spark.driver.port", "4043") \  .config("spark.some.config.option", "some-value") \  .getOrCreate()  """ Get the SparkContext from the SparkSession  This line of code retrieves the SparkContext object from a SparkSession:  The SparkContext (sc) is the main entry point for Spark functionality,  allowing you to create RDDs and perform lower-level operations. """  sc = spark.sparkContext  """ Question 3(a) (3 marks) """  print("\nQuestion 3(a) (3 marks)\n")  """ Read the 5-node-graph.txt file and store the content using Spark RDDs. """  five\_node\_graph\_Spark\_RDD = sc.textFile("/Users/shawnyang/Downloads/ICT337 ECA July 2024 Semester/ECA Datasets/5-node-graph.txt")  """ Defines a function to process each line of the input file """  def parse\_line\_from\_node\_graph\_file(line):  """ The line is split into parts using line.split() """  line\_Parts = line.split()  """ The first part of each line is converted to an integer and assigned as the node\_id. """  line\_Node\_ID = int( line\_Parts[0] )  """ The second part of each line is converted to an integer and assigned as the distance. """  line\_Distance = int( line\_Parts[1] )  """ An empty list neighbors is initialized. """  node\_neighbours\_List = []  """ Use an if condition to check if there are more than two parts in the line (i.e., if there are neighbors): """  if len(line\_Parts) > 2:    """ Checks if there are more than 3 parts, loop through each neighbor to split the third part by ':'    to separate different neighbors. """  for node\_Neighbor in line\_Parts[2].split(':'):    """ Checks if the neighbor string is not empty. """  if node\_Neighbor:    """ For each neighbor, it splits by ',' to get the neighbor's ID and weight. """  node\_Neighbor\_Node\_ID, node\_Neighbor\_Weight = node\_Neighbor.split(',')  """ The neighbor's ID and weight are converted to integers and    appended as a tuple to the neighbors list. """  node\_neighbours\_List.append(  ( int(node\_Neighbor\_Node\_ID), int(node\_Neighbor\_Weight) )  )    """ A path list is created. If the node\_id is 1, it contains [1], otherwise it's empty. """  """ An empty list called path is initialized. """  path\_List = []    """ Checks if the node\_id is equal to 1. """  if line\_Node\_ID == 1:  """ If the condition is true, it appends the node\_id to the path list. """  path\_List.append(line\_Node\_ID)  """ Finally, it returns a tuple containing the node\_id and another tuple with distance, neighbors, and path. """  return ( line\_Node\_ID, (line\_Distance, node\_neighbours\_List, path\_List) )  """ Uses map() to apply the parse\_line() function to each line, creating the required RDD structure.  Apply the parse\_line function to each line in five\_node\_graph\_rdd. """  node\_graph\_RDD\_Five = five\_node\_graph\_Spark\_RDD.map(parse\_line\_from\_node\_graph\_file)  print("5-node-graph.txt RDD Content with the structure (node\_ID,(distance, list of neighbors with associated weight, path)):\n")  """ Show the 5-node-graph.txt input file RDD content  Collect all elements from the distributed graph\_rdd into the driver program as a list and starts iterating over them. """  for node\_graph\_RDD\_Five\_Contents in node\_graph\_RDD\_Five.collect():  """ For each item in the collected list, it prints the item to the console. """  print(node\_graph\_RDD\_Five\_Contents) |

**Output:**

**A computer screen with white text

Description automatically generated**

**Question 3(b) (8 marks)**

**ANS:**

|  |
| --- |
| **""" Question 3(b) (8 marks) """**  **print("\nQuestion 3(b) (8 marks)\n")**  """ This function defines a custom minimum finder to replace Python's built-in min() function,  which may not be available in certain Spark environments. """  def custom\_Minimum\_Finder(iterable, key=None):  """ Checks if the iterable is empty. If so, it raises a ValueError. """  if not iterable:  raise ValueError("iterable is empty")    """ We define a new function called identity that simply returns its input. """  def identity(x):  return x  """ If no key function is provided, it uses the identity function to key (returns the item itself). """  if key is None:  key = identity    """ Initializes the minimum item and its value with the first item in the iterable. """  mininmum\_Item = iterable[0]    minimum\_Value = key(mininmum\_Item)    """ Iterates through the remaining items, updating min\_item and min\_value if a smaller value is found. """  for item in iterable[1:]:  value = key(item)  if value < minimum\_Value:  mininmum\_Item = item  minimum\_Value = value    """ Returns the item with the minimum value. """  return mininmum\_Item  """ This is the main function implementing Dijkstra's algorithm. It takes a graph RDD as input. """  def dijkstra\_Algorithm(node\_graph\_RDD\_Five):  """ This inner function updates the distances for neighboring nodes. """  def update\_Distances\_For\_Neighbouring\_Nodes(node):  """ It unpacks the node information. """  node\_ID, (neighbour\_Distance, node\_Neighbours, node\_Path) = node  """ It creates an empty list to store the updated neighbor information. """  updated\_Neighbours = []    """ It iterates through each neighbor and its weight. """  for node\_neighbour, node\_weight in node\_Neighbours:    """ For each neighbor, it calculates the new distance and creates a new path. """  new\_Neighbour\_Distance = neighbour\_Distance + node\_weight    new\_Neighbour\_Path = node\_Path + [node\_neighbour]  """ It appends the updated information for each neighbor to the list. """  updated\_Neighbours.append((node\_neighbour, (new\_Neighbour\_Distance, new\_Neighbour\_Path)))  """ Finally, it returns the list of updated neighbors. """  return updated\_Neighbours  """ This inner function chooses the shortest path between two options.    This function compares the first element (index 0) of two tuples a and b,    which represent the distances of two paths. """  def select\_Shortest\_Path(a, b):  if a[0] < b[0]:  """ It returns the tuple with the smaller distance, effectively selecting the shorter path. """  return a  else:  return b  """ Initializes the visited set with the starting node (1)    and the result list with the starting node's information. """  visited\_set\_with\_Starting\_Node = set([1])  shortest\_Distance = {1: 0}  shortest\_Path = {1: [1]}  """ This is the main loop of the algorithm, continuing until all nodes have been visited. """  while len(visited\_set\_with\_Starting\_Node) < node\_graph\_RDD\_Five.count():  """ We define a new function is\_node\_unvisited that takes a node tuple    and returns True if the node has not been visited yet. """  def nodes\_yet\_to\_be\_Reached(unreached\_Node\_Tuple):  unreached\_Node\_ID = unreached\_Node\_Tuple[0]  return unreached\_Node\_ID not in visited\_set\_with\_Starting\_Node  """ This part finds candidate nodes to visit next, filtering out already visited nodes    and choosing the shortest path to each unvisited node. """  next\_Candidate\_Reachable\_Nodes = node\_graph\_RDD\_Five \  .flatMap(update\_Distances\_For\_Neighbouring\_Nodes) \  .filter(nodes\_yet\_to\_be\_Reached) \  .reduceByKey(select\_Shortest\_Path) \  .collect()    """ Finally, if there are no more candidate reachable nodes, the loop breaks. """  if not next\_Candidate\_Reachable\_Nodes:  break  """ We define a new function get\_node\_distance that takes a node tuple and returns the distance value    which is the first element (index 0) of the second element (index 1) of the tuple. """  def get\_Next\_Candidate\_Reachable\_Node\_Distance(candidate\_Node\_Tuple):  return candidate\_Node\_Tuple[1][0]    """ Replace the min() function with the custom minimum finder    Selects the next node to visit based on the shortest distance. """  next\_Reachable\_Node, (neighbour\_Distance, node\_Path) = custom\_Minimum\_Finder(next\_Candidate\_Reachable\_Nodes, key=get\_Next\_Candidate\_Reachable\_Node\_Distance)    """ We add the next reachable node to the visited set"""  visited\_set\_with\_Starting\_Node.add(next\_Reachable\_Node)  """ Update the visited set shortest distance and path in the respective dictionaries. """  shortest\_Distance[next\_Reachable\_Node] = neighbour\_Distance  shortest\_Path[next\_Reachable\_Node] = node\_Path  """ Update the graph RDD with the new distance and path information    We define a separate function called update\_node\_info. This function takes a node as input and returns the updated node information."""  def update\_Shortest\_Path\_Node\_Info(node):  """ Extract the node\_id and its current information (distance, neighbors, and path). """  node\_ID = node[0]  current\_Shortest\_Distance, neighbor\_Node, current\_Shortest\_Path = node[1]    """ Check if we have a new shortest distance for this node in the shortest\_Distance dictionary. """  if node\_ID in shortest\_Distance:  """ If we do, we use the new distance and path; otherwise, we keep the current ones. """  new\_Shortest\_distance = shortest\_Distance[node\_ID]  new\_Shortest\_Path = shortest\_Path[node\_ID]  else:  new\_Shortest\_distance = current\_Shortest\_Distance  new\_Shortest\_Path = current\_Shortest\_Path  """ Return the updated node information. """  return (node\_ID, (new\_Shortest\_distance, neighbor\_Node, new\_Shortest\_Path))  """ We then use this update\_node\_info function with the map operation on our RDD to update all nodes. """  node\_graph\_RDD\_Five = node\_graph\_RDD\_Five.map(update\_Shortest\_Path\_Node\_Info)  """ Initialize an empty result list """  dijkstra\_Algorithm\_Results\_List = []    """ We iterate through the sorted keys of shortest\_Distance """  for node\_ID in sorted(shortest\_Distance.keys()):  """ Create a tuple for each node with its shortest distance and path, and append it to the result list. """  dijkstra\_Algorithm\_Results\_List.append((node\_ID, (shortest\_Distance[node\_ID], shortest\_Path[node\_ID])))  return dijkstra\_Algorithm\_Results\_List  # Run the algorithm  shortest\_Paths\_List = dijkstra\_Algorithm(node\_graph\_RDD\_Five)  # Format and print the results  print("Show the iterative Dijkstra's algorithm RDD Content with the structure (node\_ID, (shortest path distance, path traversal)):\n")  """ Prints the final result, showing the shortest distance and path for each node. """  for node\_ID, (shortest\_Distance, shortest\_Path) in shortest\_Paths\_List:  path\_string = '→'.join(map(str, shortest\_Path))  print(f"Node {node\_ID}: (Shortest distance: {shortest\_Distance}, Path: {path\_string})") |

**Output:**

**A screen shot of a computer

Description automatically generated**

**Question 3(c) (8 marks)**

.**ANS:**

|  |
| --- |
| **Based on the Question 3(a) and Question 3(b) source codes and outputs, this is how the shortest path computations are performed using each iteration step:**  **Iteration 1:**  Begin at Node 1 which is the Source Node.  The path distance from Node 1 is 0 and the path is just itself which is [1].  The Dijkstra algorithm now checks the path distances between node 1 and its unvisited neighbours which are Node 2 and Node 3   * Node 2: path distance = 0 + 10 = 10, path = [1, 2]   This means that the path distance to Node 2 is 10.  So the new path distance from Node 1 to Node 2 is 0 + 10 = 10.  The **path = [1, 2]** means that the path begins at Node 1 and ends at Node 2.   * Node 3: path distance = 0 + 5 = 5, path [1, 3]   This means that the path distance to Node 3 is 5.  So the new path distance from Node 1 to Node 3 is 0 + 5 = 5.  The **path = [1, 3]** means that the path begins at Node 1 and ends at Node 3.  The path distance from Node 1 to its neighbours Node 2 is 10 and to Node 3 is 5.  Select Node 3 as the next node to visit with the path [1, 3] because it has the shortest distance of 5.  **Iteration 2:**  From Node 3,  The Dijkstra algorithm now checks the path distances between Node 3 and its unvisited neighbours which are Node 2, Node 4 and Node 5.   * **Node 2: path distance = (min(10, 5 + 3) = 8), path = [1, 2]**   **The existing path distance to Node 2 is 10 (1st iteration)**  **The new path distance is 5 (path distance to Node 3) + 3 (weight from Node 2 to Node 3) = 8**  **The algorithm opts for the minimum distance between either 10 or 8 and 8 is the shortest distance.**   * **Node 4: path distance = 5 + 9 = 14, path = [1, 3, 4]**   This means that the path distance to Node 3 is 5 and the weight from Node 3 to Node 4 is 9.  So the new path distance from Node 3 to Node 4 is 5 + 9 = 14.  The **path = [1, 3, 4]** means that the path begins at Node 1, moves to Node 3 and ends at Node 4.   * **Node 5: path distance = 5 + 2 = 7, path = [1, 3, 5]**   This means that the path distance to Node 3 is 5 and the weight from Node 3 to Node 5 is 2.  So the new path distance from Node 3 to Node 2 is 5 + 2 = 7.  The **path = [1, 3, 5]** means that the path begins at Node 1, moves to Node 3 and ends at Node 5.  Select Node 5 as the next node to visit with the path [1, 3, 5] because it has the shortest distance of 7.  **Iteration 3:**  From Node 5,  The Dijkstra algorithm now checks the path distances between Node 5 and its unvisited neighbours which are Node 2 and Node 4.   * **Node 4: path distance = (min(14, 7 + 6) = 13), path = [1, 3, 4]**   **The existing path distance to Node 4 is 14 (2nd iteration)**  **The new path distance is 7 (path distance to Node 5) + 6 (weight from Node 5 to Node 4) = 13**  **The algorithm opts for the minimum distance between either 14 or 13 and 13 is the shortest distance.**  Select Node 2 as the next node to visit with the path [1, 2] because it is the closest unvisited node with a path distance of 8.  **Iteration 4:**  From Node 2,  The Dijkstra algorithm now checks the path distances between Node 2 and its unvisited neighbours which is Node 4.   * **Node 4: path distance = (min(13, 8 + 1) = 9), path = [1, 3, 4]**   **The existing path distance to Node 4 is 13 (3rd iteration)**  **The new path distance is 8 (path distance to Node 2) + 1 (weight from Node 2 to Node 4) = 9**  **The algorithm opts for the minimum distance between either 13 or 9 and 9 is the shortest distance.**  Select Node 4 as the next node to visit with the path [1, 3, 4] because it is the last unvisited node.  The Dijkstra algorithm stops running after all nodes have been visited.  The condition to break from the infinite while loop executes when there are 0 unvisited nodes. |

**Question 3(d) (4 marks)**

**ANS:**

|  |
| --- |
| **""" Question 3(d) (4 marks) """**  **print("\nQuestion 3(d) (4 marks)\n")**  """ This is the main function implementing Dijkstra's algorithm. It takes a graph RDD as input. """  def dijkstra\_Algorithm(node\_graph\_RDD\_Five):  """ This inner function updates the distances for neighboring nodes. """  def update\_Distances\_For\_Neighbouring\_Nodes(node):  """ It unpacks the node information. """  node\_ID, (neighbour\_Distance, node\_Neighbours, node\_Path) = node  """ It creates an empty list to store the updated neighbor information. """  updated\_Neighbours = []    """ It iterates through each neighbor and its weight. """  for node\_neighbour, node\_weight in node\_Neighbours:    """ For each neighbor, it calculates the new distance and creates a new path. """  new\_Neighbour\_Distance = neighbour\_Distance + node\_weight    new\_Neighbour\_Path = node\_Path + [node\_neighbour]  """ It appends the updated information for each neighbor to the list. """  updated\_Neighbours.append((node\_neighbour, (new\_Neighbour\_Distance, new\_Neighbour\_Path)))  """ Finally, it returns the list of updated neighbors. """  return updated\_Neighbours  """ This inner function chooses the shortest path between two options.    This function compares the first element (index 0) of two tuples a and b,    which represent the distances of two paths. """  def select\_Shortest\_Path(a, b):  if a[0] < b[0]:  """ It returns the tuple with the smaller distance, effectively selecting the shorter path. """  return a  else:  return b  """ Initializes the visited set with the starting node (1)    and the result list with the starting node's information. """  visited\_set\_with\_Starting\_Node = set([1])  shortest\_Distance = {1: 0}  shortest\_Path = {1: [1]}  number\_of\_Iterations = 0  """ This is the main loop of the algorithm, continuing until all nodes have been visited. """  while len(visited\_set\_with\_Starting\_Node) < node\_graph\_RDD\_Five.count():  number\_of\_Iterations = number\_of\_Iterations + 1  """ We define a new function is\_node\_unvisited that takes a node tuple    and returns True if the node has not been visited yet. """  def nodes\_yet\_to\_be\_Reached(unreached\_Node\_Tuple):  unreached\_Node\_ID = unreached\_Node\_Tuple[0]  return unreached\_Node\_ID not in visited\_set\_with\_Starting\_Node  """ This part finds candidate nodes to visit next, filtering out already visited nodes    and choosing the shortest path to each unvisited node. """  next\_Candidate\_Reachable\_Nodes = node\_graph\_RDD\_Five \  .flatMap(update\_Distances\_For\_Neighbouring\_Nodes) \  .filter(nodes\_yet\_to\_be\_Reached) \  .reduceByKey(select\_Shortest\_Path) \  .collect()    """ Finally, if there are no more candidate reachable nodes, the loop breaks. """  if not next\_Candidate\_Reachable\_Nodes:  break  """ We define a new function get\_node\_distance that takes a node tuple and returns the distance value    which is the first element (index 0) of the second element (index 1) of the tuple. """  def get\_Next\_Candidate\_Reachable\_Node\_Distance(candidate\_Node\_Tuple):  return candidate\_Node\_Tuple[1][0]    """ Replace the min() function with the custom minimum finder    Selects the next node to visit based on the shortest distance. """  next\_Reachable\_Node, (neighbour\_Distance, node\_Path) = custom\_Minimum\_Finder(next\_Candidate\_Reachable\_Nodes, key=get\_Next\_Candidate\_Reachable\_Node\_Distance)    """ We add the next reachable node to the visited set"""  visited\_set\_with\_Starting\_Node.add(next\_Reachable\_Node)  """ Update the visited set shortest distance and path in the respective dictionaries. """  shortest\_Distance[next\_Reachable\_Node] = neighbour\_Distance  shortest\_Path[next\_Reachable\_Node] = node\_Path    """ Update the graph RDD with the new distance and path information    We define a separate function called update\_node\_info. This function takes a node as input and returns the updated node information."""  def update\_Shortest\_Path\_Node\_Info(node):  """ Extract the node\_id and its current information (distance, neighbors, and path). """  node\_ID = node[0]  current\_Shortest\_Distance, neighbor\_Node, current\_Shortest\_Path = node[1]    """ Check if we have a new shortest distance for this node in the shortest\_Distance dictionary. """  if node\_ID in shortest\_Distance:  """ If we do, we use the new distance and path; otherwise, we keep the current ones. """  new\_Shortest\_distance = shortest\_Distance[node\_ID]  new\_Shortest\_Path = shortest\_Path[node\_ID]  else:  new\_Shortest\_distance = current\_Shortest\_Distance  new\_Shortest\_Path = current\_Shortest\_Path  """ Return the updated node information. """  return (node\_ID, (new\_Shortest\_distance, neighbor\_Node, new\_Shortest\_Path))  """ We then use this update\_node\_info function with the map operation on our RDD to update all nodes. """  node\_graph\_RDD\_Five = node\_graph\_RDD\_Five.map(update\_Shortest\_Path\_Node\_Info)  """ Initialize an empty result list """  dijkstra\_Algorithm\_Results\_List = []    """ We iterate through the sorted keys of shortest\_Distance """  for node\_ID in sorted(shortest\_Distance.keys()):  """ Create a tuple for each node with its shortest distance and path, and append it to the result list. """  dijkstra\_Algorithm\_Results\_List.append((node\_ID, (shortest\_Distance[node\_ID], shortest\_Path[node\_ID])))  return dijkstra\_Algorithm\_Results\_List, number\_of\_Iterations  """ This line runs the Dijkstra's algorithm on the input graph RDD    and returns the shortest paths and the number of iterations it took to complete."""  shortest\_Paths\_List, num\_of\_Shortest\_Path\_Iterations = dijkstra\_Algorithm(node\_graph\_RDD\_Five)  print(f"Number of iterations to complete the shortest path computation for 5-node-graph.txt: {num\_of\_Shortest\_Path\_Iterations}")  print("\nFinal output (node\_ID, (shortest path distance, path traversal)), sorted by ascending node\_ID:\n")  """ The for loop iterates through the shortest\_paths result:  This unpacks each result into node ID, distance, and path. """  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in shortest\_Paths\_List:  """ This prints the formatted output for each node including, node ID,    shortest distance to that node from the start node and path to reach that node,    formatted as a string with arrows (→) between node numbers """  print(f"(Node {node\_ID}: (Shortest Path Distance: {shortest\_Path\_Distance}, Path Traversal: {'→'.join(map(str, shortest\_Path))}))") |

**Output:**

**A screenshot of a computer program

Description automatically generated**

**Question 3(e) (9 marks)**

**ANS:**

|  |
| --- |
| **""" Question 3(e) (9 marks) """**  print("\nQuestion 3(e) (9 marks)\n")  """ This line runs the Dijkstra's algorithm on the input graph RDD    and returns the shortest paths and the number of iterations it took to complete."""  shortest\_Paths\_List, num\_of\_Shortest\_Path\_Iterations = dijkstra\_Algorithm(node\_graph\_RDD\_Five)  """ We define a new function get\_node\_distance that takes a node tuple and returns the distance value    which is the first element (index 0) of the second element (index 1) of the tuple. """  def get\_Candidate\_Reachable\_Node\_Distance(candidate\_Node\_Tuple):  return candidate\_Node\_Tuple[1][0]  """ Sort the paths by distance in descending order  We pass this function as the key argument to sorted.  This tells the sorting function to use the distance value when comparing paths.  The reverse=True argument ensures that the paths are sorted in descending order of distance. """  sorted\_shortest\_Paths\_List = sorted(shortest\_Paths\_List, key=get\_Candidate\_Reachable\_Node\_Distance, reverse=True)  """ Get the top 3 furthermost nodes """  top\_3\_Furthermost\_Shortest\_Paths\_Nodes = sorted\_shortest\_Paths\_List[:3]  print("Top Three (3) furthermost nodes, their paths & distances (sorted by descending distance):\n")  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in top\_3\_Furthermost\_Shortest\_Paths\_Nodes:  print(f"Node {node\_ID}: (Distance: {shortest\_Path\_Distance}, Path: {'→'.join(map(str, shortest\_Path))})")    """ This line runs the Dijkstra's algorithm on the input graph RDD    and returns the shortest paths and the number of iterations it took to complete."""  shortest\_Paths\_List, num\_of\_Shortest\_Path\_Iterations = dijkstra\_Algorithm(node\_graph\_RDD\_Five)  """ We define a new function that takes a path tuple and returns the length of the path.  The function unpacks the tuple to access the path, then returns its length. """  def get\_Shortest\_Paths\_Length(path\_tuple):  \_, (\_, path) = path\_tuple  return len(path)  """ Find the path with the most hops  We use the py\_max() function to apply get\_path\_length to each item in shortest\_Paths\_List.  The py\_max() function is then used to find the maximum length among all paths.  Using py\_max() instead of max() to find the maximum number of hops. """  maximum\_Num\_of\_Shortest\_Paths\_Hops = py\_max(map(get\_Shortest\_Paths\_Length, shortest\_Paths\_List))  """ We initialize an empty list nodes\_with\_max\_hops to store the results. """  shortest\_Paths\_Nodes\_With\_Maximum\_Hops\_List = []  """ Find all nodes with paths equal to max\_hops  We iterate through each item in shortest\_Paths\_List, unpacking it into node, distance, and path. """  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in shortest\_Paths\_List:    """ For each item, we check if the length of the path is equal to maximum\_Num\_of\_Shortest\_Paths\_Hops. """  if len(shortest\_Path) == maximum\_Num\_of\_Shortest\_Paths\_Hops:  """ If the condition is true, we append a tuple containing the node, distance,    and path to nodes\_with\_max\_hops. """  shortest\_Paths\_Nodes\_With\_Maximum\_Hops\_List.append((node\_ID, (shortest\_Path\_Distance, shortest\_Path)))  print("\nDestination node(s) with the most number of traversal hops in the path:\n")  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in shortest\_Paths\_Nodes\_With\_Maximum\_Hops\_List:  print(f"Node {node\_ID}: (Distance: {shortest\_Path\_Distance}, Path: {'→'.join(map(str, shortest\_Path))}, Hops: {len(shortest\_Path) - 1})")    """ This line runs the Dijkstra's algorithm on the input graph RDD    and returns the shortest paths and the number of iterations it took to complete."""  shortest\_Paths\_List, num\_of\_Shortest\_Path\_Iterations = dijkstra\_Algorithm(node\_graph\_RDD\_Five)  """ Get all node IDs from the graph  It gets all node IDs from the graph using graph\_rdd.keys().collect(). """  get\_Set\_of\_Node\_ID\_From\_All\_Graph\_Nodes = set(node\_graph\_RDD\_Five.keys().collect())  """ We initialize an empty set get\_Set\_of\_Reachable\_Graph\_Nodes to store the reachable nodes. """  get\_Set\_of\_Reachable\_Graph\_Nodes\_Set = set()  """ Get the set of reachable nodes  It creates a set of reachable nodes by checking which nodes have a finite distance in shortest\_paths.  We iterate through each item in shortest\_Paths\_List, unpacking it into node, distance, and path. """  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in shortest\_Paths\_List:  """ For each item, we check if the distance is not equal to infinity (float('inf')). """  if shortest\_Path\_Distance != float('inf'):  """ If the condition is true, we add the node to the get\_Set\_of\_Reachable\_Graph\_Nodes set. """  get\_Set\_of\_Reachable\_Graph\_Nodes\_Set.add(node\_ID)  """ Find unreachable nodes  It finds unreachable nodes by subtracting the set of reachable nodes from the set of all nodes. """  unreachable\_Graph\_Nodes = get\_Set\_of\_Node\_ID\_From\_All\_Graph\_Nodes - get\_Set\_of\_Reachable\_Graph\_Nodes\_Set  """ Sort unreachable nodes by ascending node\_ID  It sorts the unreachable nodes in ascending order. """  sorted\_by\_Ascending\_Unreachable\_Graph\_Nodes\_List = sorted(unreachable\_Graph\_Nodes)  print("\nNodes that are not reachable from source node (node\_ID=1), sorted by ascending node\_ID:\n")  """ It then checks if there are any unreachable nodes in sorted\_by\_Ascending\_Unreachable\_Graph\_Nodes.  If there are unreachable nodes: """  if sorted\_by\_Ascending\_Unreachable\_Graph\_Nodes\_List:  """ It iterates through each node in sorted\_by\_Ascending\_Unreachable\_Graph\_Nodes """  for unreachable\_Nodes\_From\_Source\_Node in sorted\_by\_Ascending\_Unreachable\_Graph\_Nodes\_List:  """ For each unreachable node, it prints "Node {node}" """  print(f"Node {unreachable\_Nodes\_From\_Source\_Node}")  else:  """ If there are no unreachable nodes:    It prints the statement """  print("All nodes are reachable from the Node 1.") |

**Output:**

**A screenshot of a computer program

Description automatically generated**

**Question 3(f) (8 marks)**

**ANS:**

|  |
| --- |
| **""" Question 3(f) (8 marks) """**  **print("\nQuestion 3(f) (8 marks)\n")**  **print("Computations and Reslts for 20-node-graph.txt:\n")**  **""" Question 3(a) Section: """**  """ Read the 20-node-graph.txt file and store the content using Spark RDDs. """  twenty\_node\_graph\_Spark\_RDD = sc.textFile("/Users/shawnyang/Downloads/ICT337 ECA July 2024 Semester/ECA Datasets/20-node-graph.txt")  """ Uses map() to apply the parse\_line() function to each line, creating the required RDD structure. """  node\_graph\_RDD\_Twenty = twenty\_node\_graph\_Spark\_RDD.map(parse\_line\_from\_node\_graph\_file)  print("20-node-graph.txt RDD Content with the structure (node\_ID,(distance, list of neighbors with associated weight, path)):\n")  """ Show the 5-node-graph.txt input file RDD content  Prints the content of the RDD using collect() and a for loop. """  for node\_graph\_RDD\_Twenty\_Contents in node\_graph\_RDD\_Twenty.collect():  print(node\_graph\_RDD\_Twenty\_Contents)  **""" Question 3(d) Section: """**  # Run Dijkstra's algorithm  twenty\_Node\_Graph\_RDD\_Shortest\_Paths\_Lists, twenty\_Node\_Graph\_RDD\_num\_of\_Shortest\_Path\_Iterations = dijkstra\_Algorithm(node\_graph\_RDD\_Twenty)  print(f"\nNumber of iterations to complete the shortest path computation for 20-node-graph.txt: {twenty\_Node\_Graph\_RDD\_num\_of\_Shortest\_Path\_Iterations}")  print("\nFinal output (node\_ID, (shortest path distance, path traversal)), sorted by ascending node\_ID:\n")  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in twenty\_Node\_Graph\_RDD\_Shortest\_Paths\_Lists:  print(f"Node {node\_ID}: (Shortest distance: {shortest\_Path\_Distance}, Path: {'→'.join(map(str, shortest\_Path))})")  **""" Question 3(e) Section: """**  """ We define a new function get\_node\_distance that takes a node tuple and returns the distance value    which is the first element (index 0) of the second element (index 1) of the tuple. """  def get\_Candidate\_Reachable\_Node\_Distance(candidate\_Node\_Tuple):  return candidate\_Node\_Tuple[1][0]  """ Sort the paths by distance in descending order  We pass this function as the key argument to sorted.  This tells the sorting function to use the distance value when comparing paths.  The reverse=True argument ensures that the paths are sorted in descending order of distance. """  twenty\_Sorted\_Shortest\_Paths\_List = sorted(twenty\_Node\_Graph\_RDD\_Shortest\_Paths\_Lists, key=get\_Candidate\_Reachable\_Node\_Distance, reverse=True)  """ Get the top 3 furthermost nodes """  twenty\_Node\_Top\_3\_Furthermost\_Shortest\_Paths\_Nodes = twenty\_Sorted\_Shortest\_Paths\_List[:3]  print("\n20-node-graph.txt Top Three (3) furthermost nodes, their paths & distances (sorted by descending distance):\n")  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in twenty\_Node\_Top\_3\_Furthermost\_Shortest\_Paths\_Nodes:  print(f"Node {node\_ID}: (Distance: {shortest\_Path\_Distance}, Path: {'→'.join(map(str, shortest\_Path))})")    """ Find the path with the most hops  We use the py\_max() function to apply get\_path\_length to each item in shortest\_Paths\_List.  The py\_max() function is then used to find the maximum length among all paths.  Using py\_max() instead of max() to find the maximum number of hops. """  maximum\_Num\_of\_Shortest\_Paths\_Hops = py\_max(map(get\_Shortest\_Paths\_Length, twenty\_Sorted\_Shortest\_Paths\_List))  """ We initialize an empty list nodes\_with\_max\_hops to store the results. """  twenty\_Node\_Shortest\_Paths\_With\_Maximum\_Hops\_List = []  """ Find all nodes with paths equal to max\_hops  We iterate through each item in shortest\_Paths\_List, unpacking it into node, distance, and path. """  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in twenty\_Sorted\_Shortest\_Paths\_List:    """ For each item, we check if the length of the path is equal to maximum\_Num\_of\_Shortest\_Paths\_Hops. """  if len(shortest\_Path) == maximum\_Num\_of\_Shortest\_Paths\_Hops:  """ If the condition is true, we append a tuple containing the node, distance,    and path to nodes\_with\_max\_hops. """  twenty\_Node\_Shortest\_Paths\_With\_Maximum\_Hops\_List.append((node\_ID, (shortest\_Path\_Distance, shortest\_Path)))  print("\n20-node-graph.txt Destination node(s) with the most number of traversal hops in the path:\n")  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in twenty\_Node\_Shortest\_Paths\_With\_Maximum\_Hops\_List:  print(f"Node {node\_ID}: (Distance: {shortest\_Path\_Distance}, Path: {'→'.join(map(str, shortest\_Path))}, Hops: {len(shortest\_Path) - 1})")    """ Get all node IDs from the graph  It gets all node IDs from the graph using graph\_rdd.keys().collect(). """  get\_Set\_of\_Node\_ID\_From\_All\_Twenty\_Nodes = set(node\_graph\_RDD\_Twenty.keys().collect())  """ We initialize an empty set get\_Set\_of\_Reachable\_Graph\_Nodes to store the reachable nodes. """  get\_Set\_of\_Reachable\_Twenty\_Nodes\_Set = set()  """ Get the set of reachable nodes  It creates a set of reachable nodes by checking which nodes have a finite distance in shortest\_paths.  We iterate through each item in shortest\_Paths\_List, unpacking it into node, distance, and path. """  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in twenty\_Node\_Graph\_RDD\_Shortest\_Paths\_Lists:  """ For each item, we check if the distance is not equal to infinity (float('inf')). """  if shortest\_Path\_Distance != float('inf'):  """ If the condition is true, we add the node to the get\_Set\_of\_Reachable\_Graph\_Nodes set. """  get\_Set\_of\_Reachable\_Twenty\_Nodes\_Set.add(node\_ID)  """ Find unreachable nodes  It finds unreachable nodes by subtracting the set of reachable nodes from the set of all nodes. """  twenty\_Node\_Unreachable\_Nodes\_List = sorted(set( get\_Set\_of\_Node\_ID\_From\_All\_Twenty\_Nodes - get\_Set\_of\_Reachable\_Twenty\_Nodes\_Set ))  """ Sort unreachable nodes by ascending node\_ID  It sorts the unreachable nodes in ascending order. """  sorted\_by\_Ascending\_Twenty\_Node\_Unreachable\_Nodes\_List = sorted(twenty\_Node\_Unreachable\_Nodes\_List)  print("\nNodes that are not reachable from source node (node\_ID=1), sorted by ascending node\_ID:\n")  """ It then checks if there are any unreachable nodes in sorted\_by\_Ascending\_Unreachable\_Graph\_Nodes.  If there are unreachable nodes: """  if sorted\_by\_Ascending\_Twenty\_Node\_Unreachable\_Nodes\_List:  """ It iterates through each node in sorted\_by\_Ascending\_Unreachable\_Graph\_Nodes """  for unreachable\_Twenty\_Nodes\_From\_Source\_Node in sorted\_by\_Ascending\_Twenty\_Node\_Unreachable\_Nodes\_List:  """ For each unreachable node, it prints "Node {node}" """  print(f"Node {unreachable\_Twenty\_Nodes\_From\_Source\_Node}")  else:  """ If there are no unreachable nodes:    It prints the statement """  print("All nodes are reachable from the Node 1.")    **print("\n\n\nComputations and Reslts for 40-node-graph.txt:\n")**  **""" Question 3(a) Section: """**  """ Read the 40-node-graph.txt file and store the content using Spark RDDs. """  fourty\_node\_graph\_Spark\_RDD = sc.textFile("/Users/shawnyang/Downloads/ICT337 ECA July 2024 Semester/ECA Datasets/40-node-graph.txt")  """ Uses map() to apply the parse\_line() function to each line, creating the required RDD structure. """  node\_graph\_RDD\_Fourty = fourty\_node\_graph\_Spark\_RDD.map(parse\_line\_from\_node\_graph\_file)  print("40-node-graph.txt RDD Content with the structure (node\_ID,(distance, list of neighbors with associated weight, path)):\n")  """ Show the 40-node-graph.txt input file RDD content  Prints the content of the RDD using collect() and a for loop. """  for item in node\_graph\_RDD\_Fourty.collect():  print(item)  **""" Question 3(d) Section: """**  # Run Dijkstra's algorithm  fourty\_Node\_Graph\_RDD\_Shortest\_Paths\_Lists, fourty\_Node\_Graph\_RDD\_num\_of\_Shortest\_Path\_Iterations = dijkstra\_Algorithm(node\_graph\_RDD\_Fourty)  print(f"\nNumber of iterations to complete the shortest path computation for 20-node-graph.txt: {fourty\_Node\_Graph\_RDD\_num\_of\_Shortest\_Path\_Iterations}")  print("\nFinal output (node\_ID, (shortest path distance, path traversal)), sorted by ascending node\_ID:\n")  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in fourty\_Node\_Graph\_RDD\_Shortest\_Paths\_Lists:  print(f"Node {node\_ID}: (Shortest distance: {shortest\_Path\_Distance}, Path: {'→'.join(map(str, shortest\_Path))})")  **""" Question 3(e) Section: """**  """ We define a new function get\_node\_distance that takes a node tuple and returns the distance value    which is the first element (index 0) of the second element (index 1) of the tuple. """  def get\_Candidate\_Reachable\_Node\_Distance(candidate\_Node\_Tuple):  return candidate\_Node\_Tuple[1][0]  """ Sort the paths by distance in descending order  We pass this function as the key argument to sorted.  This tells the sorting function to use the distance value when comparing paths.  The reverse=True argument ensures that the paths are sorted in descending order of distance. """  fourty\_Sorted\_Shortest\_Paths\_List = sorted(fourty\_Node\_Graph\_RDD\_Shortest\_Paths\_Lists, key=get\_Candidate\_Reachable\_Node\_Distance, reverse=True)  """ Get the top 3 furthermost nodes """  fourty\_Node\_Top\_3\_Furthermost\_Shortest\_Paths\_Nodes = fourty\_Sorted\_Shortest\_Paths\_List[:3]  print("\n20-node-graph.txt Top Three (3) furthermost nodes, their paths & distances (sorted by descending distance):\n")  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in fourty\_Node\_Top\_3\_Furthermost\_Shortest\_Paths\_Nodes:  print(f"Node {node\_ID}: (Distance: {shortest\_Path\_Distance}, Path: {'→'.join(map(str, shortest\_Path))})")    """ Find the path with the most hops  We use the py\_max() function to apply get\_path\_length to each item in shortest\_Paths\_List.  The py\_max() function is then used to find the maximum length among all paths.  Using py\_max() instead of max() to find the maximum number of hops. """  maximum\_Num\_of\_Shortest\_Paths\_Hops = py\_max(map(get\_Shortest\_Paths\_Length, fourty\_Sorted\_Shortest\_Paths\_List))  """ We initialize an empty list nodes\_with\_max\_hops to store the results. """  fourty\_Node\_Shortest\_Paths\_With\_Maximum\_Hops\_List = []  """ Find all nodes with paths equal to max\_hops  We iterate through each item in shortest\_Paths\_List, unpacking it into node, distance, and path. """  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in fourty\_Sorted\_Shortest\_Paths\_List:    """ For each item, we check if the length of the path is equal to maximum\_Num\_of\_Shortest\_Paths\_Hops. """  if len(shortest\_Path) == maximum\_Num\_of\_Shortest\_Paths\_Hops:  """ If the condition is true, we append a tuple containing the node, distance,    and path to nodes\_with\_max\_hops. """  fourty\_Node\_Shortest\_Paths\_With\_Maximum\_Hops\_List.append((node\_ID, (shortest\_Path\_Distance, shortest\_Path)))  print("\n20-node-graph.txt Destination node(s) with the most number of traversal hops in the path:\n")  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in fourty\_Node\_Shortest\_Paths\_With\_Maximum\_Hops\_List:  print(f"Node {node\_ID}: (Distance: {shortest\_Path\_Distance}, Path: {'→'.join(map(str, shortest\_Path))}, Hops: {len(shortest\_Path) - 1})")    """ Get all node IDs from the graph  It gets all node IDs from the graph using graph\_rdd.keys().collect(). """  get\_Set\_of\_Node\_ID\_From\_All\_Fourty\_Nodes = set(node\_graph\_RDD\_Fourty.keys().collect())  """ We initialize an empty set get\_Set\_of\_Reachable\_Graph\_Nodes to store the reachable nodes. """  get\_Set\_of\_Reachable\_Fourty\_Nodes\_Set = set()  """ Get the set of reachable nodes  It creates a set of reachable nodes by checking which nodes have a finite distance in shortest\_paths.  We iterate through each item in shortest\_Paths\_List, unpacking it into node, distance, and path. """  for node\_ID, (shortest\_Path\_Distance, shortest\_Path) in fourty\_Node\_Graph\_RDD\_Shortest\_Paths\_Lists:  """ For each item, we check if the distance is not equal to infinity (float('inf')). """  if shortest\_Path\_Distance != float('inf'):  """ If the condition is true, we add the node to the get\_Set\_of\_Reachable\_Graph\_Nodes set. """  get\_Set\_of\_Reachable\_Fourty\_Nodes\_Set.add(node\_ID)  """ Find unreachable nodes  It finds unreachable nodes by subtracting the set of reachable nodes from the set of all nodes. """  fourty\_Node\_Unreachable\_Nodes\_List = sorted(set( get\_Set\_of\_Node\_ID\_From\_All\_Fourty\_Nodes - get\_Set\_of\_Reachable\_Fourty\_Nodes\_Set ))  """ Sort unreachable nodes by ascending node\_ID  It sorts the unreachable nodes in ascending order. """  sorted\_by\_Ascending\_Fourty\_Node\_Unreachable\_Nodes\_List = sorted(fourty\_Node\_Unreachable\_Nodes\_List)  print("\nNodes that are not reachable from source node (node\_ID=1), sorted by ascending node\_ID:\n")  """ It then checks if there are any unreachable nodes in sorted\_by\_Ascending\_Unreachable\_Graph\_Nodes.  If there are unreachable nodes: """  if sorted\_by\_Ascending\_Fourty\_Node\_Unreachable\_Nodes\_List:  """ It iterates through each node in sorted\_by\_Ascending\_Unreachable\_Graph\_Nodes """  for unreachable\_Fourty\_Nodes\_From\_Source\_Node in sorted\_by\_Ascending\_Fourty\_Node\_Unreachable\_Nodes\_List:  """ For each unreachable node, it prints "Node {node}" """  print(f"Node {unreachable\_Fourty\_Nodes\_From\_Source\_Node}")  else:  """ If there are no unreachable nodes:    It prints the statement """  print("All nodes are reachable from the Node 1.") |

**Output:**

**A computer screen shot of a black screen

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer

Description automatically generatedA computer screen with white text

Description automatically generated**

**----- END OF ECA PAPER -----**