Homework 5

CS51510-001, Fall 2025
Purdue University Northwest
11/21/2025

Name	Email
Raaj Patel	**Pate2682@pnw.edu**

Table of Contents

Table of Figures

ABSTRACT

This homework focuses on identifying and repairing data inconsistencies in the provided shopping_data.csv file using classical graph algorithms. The dataset contains four key attributes Genre, Age, Annual Income, and Spending Score several of which include typographical mistakes, unrealistic numeric values, and entries that fall outside valid ranges. Since even small errors can distort downstream analysis, a systematic and algorithmic approach was required to clean the dataset accurately.

To correct categorical mistakes in the Genre column, I constructed an edit-distance graph that connects strings differing by a single character. Applying Breadth-First Search (BFS) on this graph allowed automatic identification of the closest valid label ("Male" or "Female") for every misspelled entry. This technique reliably corrected variations such as "Fmale", "Mle", "Femae", and "Mal" without relying on manual mapping or hard-coded rules. The BFS-based correction ensures that each typo is resolved through the shortest valid path in the graph.

For the numeric attributes (Age, Annual Income, and Spending Score), the dataset was modeled as a weighted feature graph, where each row becomes a node and edges connect neighboring rows. The weight of each edge corresponds to the geometric distance between two customers in the three-dimensional feature space. Using this structure, Dijkstra's shortest-path algorithm was applied to every row containing invalid values. The algorithm locates the nearest valid neighbor based on true similarity patterns in the data, allowing corrections such as replacing negative ages, extremely low incomes, or zero scoring values with realistic values

from the closest valid row. This approach maintains the internal consistency of the dataset and avoids arbitrary or exaggerated adjustments.

Overall, this assignment demonstrates how classic graph algorithms BFS for categorical typo repair and Dijkstra for numeric correction can be combined into a practical and reliable workflow for data cleaning. The project reinforces theoretical understanding while providing hands-on experience in constructing graphs, detecting errors, applying shortest-path methods, and ensuring data integrity for subsequent analysis.

INTRODUCTION

In this homework, the main objective was to clean and repair the provided "shopping_data.csv" file using several graph algorithms discussed in class. The dataset includes 200 customer records with four attributes Genre, Age, Annual Income, and Spending Score. and although it looks simple at first glance, it contains numerous inconsistencies. These include typographical errors in the Genre column, negative or unrealistic values in the numeric fields, and entries that fall outside reasonable ranges. Using such a dataset directly for analysis would lead to misleading insights, so the focus of this assignment was to detect and correct these errors through algorithmic techniques rather than manual editing.

To fix the categorical mistakes in the Genre column, I built an edit-distance graph connecting words that differ by just one character. Applying Breadth-First Search (BFS) on this graph made it possible to automatically map misspelled values like "Mle," "Mal," "Femae," and "Fmale" back to the valid labels "Male" and "Female." This method ensures that every typo correction is based on graph structure, making the process systematic, scalable, and reproducible even for much larger datasets.

For the numerical attributes Age, Annual Income, and Spending Score .That represented all rows as nodes in a feature graph, where edges connect neighboring rows and edge weights reflect similarity in the three-dimensional feature space. By using Dijkstra's shortest-path algorithm, each invalid row was matched with the closest valid row in terms of Age, Income, and Score. This approach replaces unrealistic values (such as Age = –19, Income = 1, or Spending Score = 0) with numbers that are consistent with the dataset's natural

distribution. Instead of guessing or filling values arbitrarily, Dijkstra ensures that every correction is grounded in actual customer similarity.

Overall, this homework brings together multiple concepts including edit distance, BFS, weighted graph construction, and Dijkstra to create a complete workflow for data cleaning. Working through these steps helped me understand not only how each algorithm functions individually, but also how they interact when applied to real-world datasets that contain noise, typos, and inconsistencies.

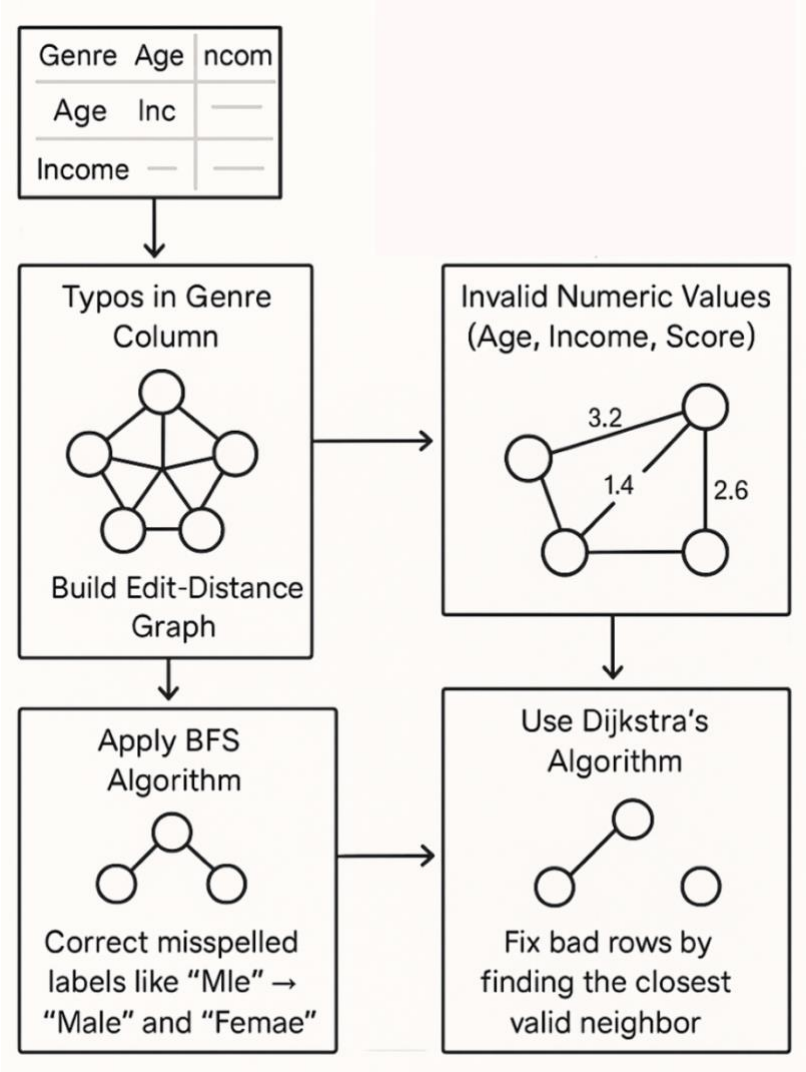


Figure 1 – A flowchart for identify error and fix data points

1. PREPARING THE WORKING ENVIRONMENT

The first step in this homework was to prepare my Ubuntu workspace and organize everything required to run the graph-based data-repair algorithms. Just like previous assignments, I followed a structured setup process so that the dataset, Python code, and Go code all stay organized and easy to test.

The dataset provided was "shopping_data.csv", which contains customer information such as Genre, Age, Annual Income, and Spending Score. Before writing any code, I needed to create a clean working directory for Homework-5 and place this dataset inside the correct folders. To keep things consistent with the earlier assignments, I used the terminal inside the Ubuntu virtual machine to create new directories and verify that the dataset was placed correctly. This step ensures that the remaining parts of the homework running BFS, Dijkstra, and generating results can be executed smoothly without file-path issues.

I completed the setup in two small steps:

Step 1: Create the homework folder and subdirectories.

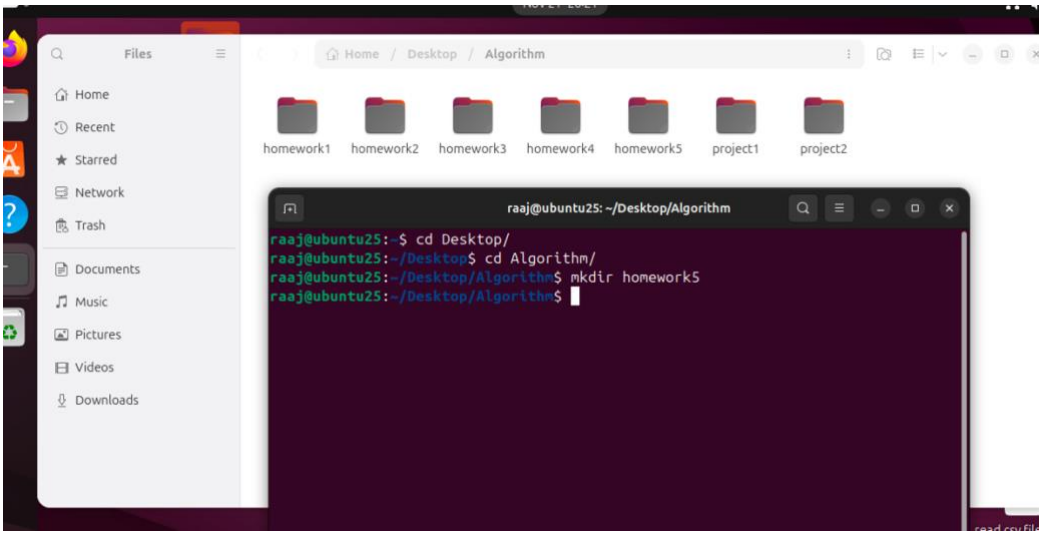


Figure 2 – Creating the homework5 directory inside the Algorithm folder

Step 2: Copy the dataset and create empty Python and Go source files.

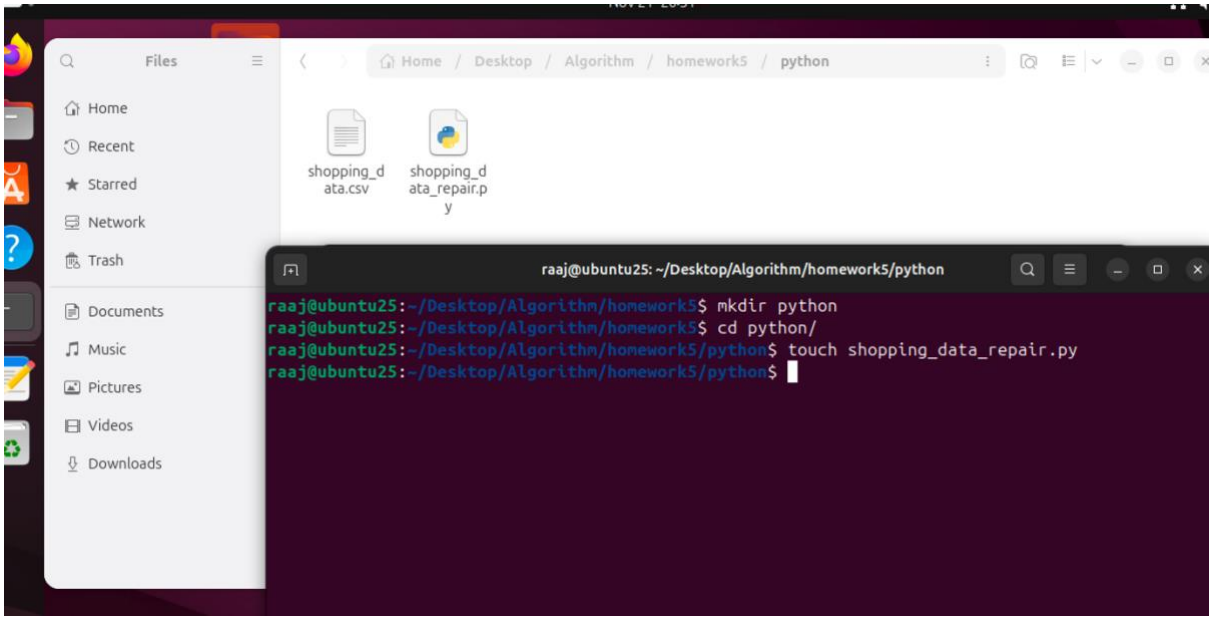


Figure 3 – Python workspace with shopping_data.csv and file shopping_data_repair.py

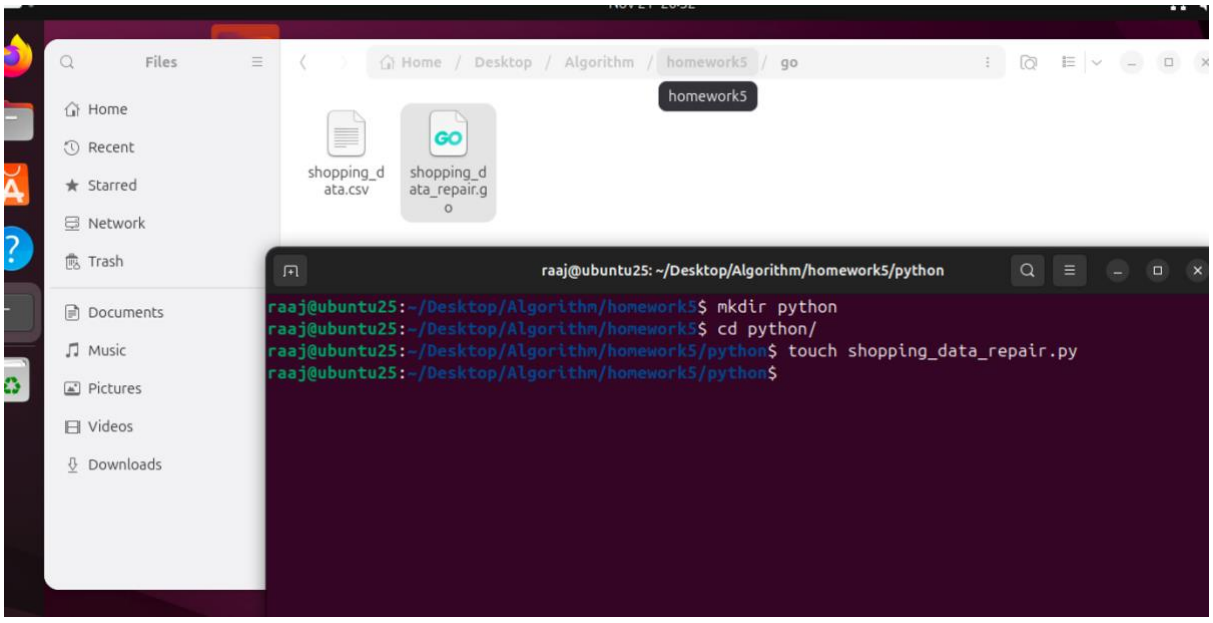


Figure 4 – Go workspace with shopping_data.csv and file shopping_data_repair.py

2. IDENTIFY AND REPAIR DATA ERRORS WITH GRAPHIC ALGORITHMS

In this assignment, identifying and repairing data errors was the central part of the project. Instead of manually correcting incorrect values, the objective was to use graphic algorithms (graph-based algorithms) to automatically detect mistakes and replace them with the most suitable valid values. The dataset contains different types of issues: spelling mistakes in the Genre column, negative or unrealistic numbers in Age, incorrect Annual Income values, and invalid Spending Scores. Each type of error required a different kind of graph algorithm to repair it effectively.

I. Detecting and fixing errors in categorical data using BFS

In the original dataset, the Genre column contained several small spelling mistakes such as “Mle”, “Mal”, “Femae”, and “Fmale”. Since these errors were close to the correct words “Male” and “Female,” I needed a systematic way to detect which incorrect entries were likely just one-character mistakes. To solve this, I constructed a small edit-distance graph in which each unique Genre string from the dataset became a node, and an edge was added between two nodes whenever the words differed by exactly one character. With this approach, strings like “Fmale” and “Femae” were directly connected to “Female”, while “Mle” and “Mal” were connected to “Male”. This forms two small, connected components: one around “Female” and one around “Male.”

Once this graph was built, I applied Breadth-First Search (BFS) for every Genre value that did not exactly match either “Male” or “Female.” Because BFS explores the graph level-by-level, the first valid Genre label it encounters is guaranteed to be the closest correct spelling.

This ensures that we correct typos based on the minimum number of edits, rather than guessing. Using this method, the algorithm automatically repaired all four incorrect values: “Fmale” and “Femae” were mapped to “Female”, while “Mle” and “Mal” were successfully corrected to “Male.”

The effectiveness of this approach is shown in the terminal output. The screenshot displays the detected unique Genre values, the constructed edit-distance graph with each word’s neighbors, and finally the BFS-based corrections, where each incorrect spelling is replaced with the closest valid label along with its corresponding CustomerID. Including this screenshot in the report helps demonstrate the internal steps of the algorithm and confirms that BFS corrected all misspelled Genre values in a consistent and explainable way.

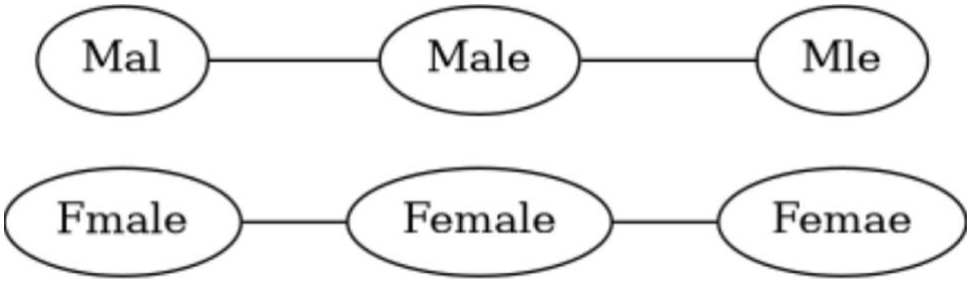


Figure 5 – Edit-Distance Graph

```
Find Unique rows and fix with BFS in categorical data
Unique raw genres: [Fmale Male Female Mal Mle Femae]
Genre typo graph using edit distance = 1 :
   Femae -> [Female]
   Fmale -> [Female]
   Female -> [Fmale Femae]
   Male -> [Mal Mle]
   Mal -> [Male]
   Mle -> [Male]
Fix which wrong spellings are one edit away from the correct spelling:
[BFS fix] Mal -> Male (CustomerID=0066)
[BFS fix] Mle -> Male (CustomerID=0110)
[BFS fix] Femae -> Female (CustomerID=0158)
[BFS fix] Fmale -> Female (CustomerID=0197)
Total genre fixed: 4.
```

Figure 6 – BFS-Based Genre Correction Output

The screenshot above shows the complete output of the BFS-based typo correction process applied to the Genre column. First, the program lists all the unique raw Genre values

found in the dataset, including the correct labels (“Male”, “Female”) and several misspelled variations such as “Mle”, “Mal”, “Femae”, and “Fmale”. After that, the script prints the edit-distance graph, where each line shows which words differ by only one character. For example, “Fmale” and “Femae” are both connected to “Female”, while “Mle” and “Mal” are directly connected to “Male”. This small graph helps visualize how the algorithm interprets each typo as a near-match to its correct counterpart.

Once the graph is constructed, the program uses Breadth-First Search (BFS) to repair incorrect spellings. BFS explores the nearest neighbors first, so the first valid word it finds (“Male” or “Female”) is the correct correction. The lines labeled [BFS fix] show exactly which values were corrected, along with the corresponding CustomerID. For instance, “Mal” and “Mle” were both mapped to “Male”, and “Femae” and “Fmale” were mapped to “Female”. The final line confirms that four Genre entries were successfully repaired. This screenshot demonstrates that the BFS algorithm correctly identified and fixed all typographical errors in a transparent and explainable way.

II. Identifying invalid numeric values

Once the Genre column was corrected, the next step was to evaluate the three numerical attributes Age, Annual Income (k$), and Spending Score (1–100) to identify values that were clearly outside realistic boundaries. These columns should follow logical constraints, yet the dataset contained multiple entries that broke those rules. For instance, one customer had an age of –19, which is impossible, and another had an age recorded as 0, which is also invalid. Similarly, an income value of 1 k$ does not fit the income distribution of the dataset, and a Spending Score of 0 falls outside the defined 1–100 scale. Values like these can mislead visualizations, skew clustering, or distort any statistical analysis, so they needed to be repaired using a method that considers the natural structure of the data.

To approach this in a principled way, I built a feature graph where each customer row becomes a node. Nodes are connected to their adjacent rows, forming a simple chain-like graph. The weight of each edge is based on the geometric distance between the two rows in the three-dimensional feature space (Age, Income, and Spending Score). The idea behind constructing this graph is that rows with similar numeric patterns will be closer in this metric space. Later, this structure allows us to use Dijkstra's shortest path algorithm to identify the closest valid row for every invalid one. By borrowing corrected values from the nearest neighbour, each repair stays consistent with the dataset's own distribution rather than relying on arbitrary thresholds or fixed averages.

Before running Dijkstra, the script performs one quick correction: any negative Age value is converted to its absolute value because negative ages are logically impossible. This can be seen in the terminal output:

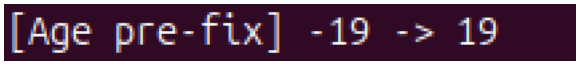


Figure 7 – Terminal Output Showing Automatic Correction of Negative Age Values

After this initial adjustment, the script scans the entire dataset and prints the indices of rows that violate the established numeric ranges. In the screenshot, the program reports:

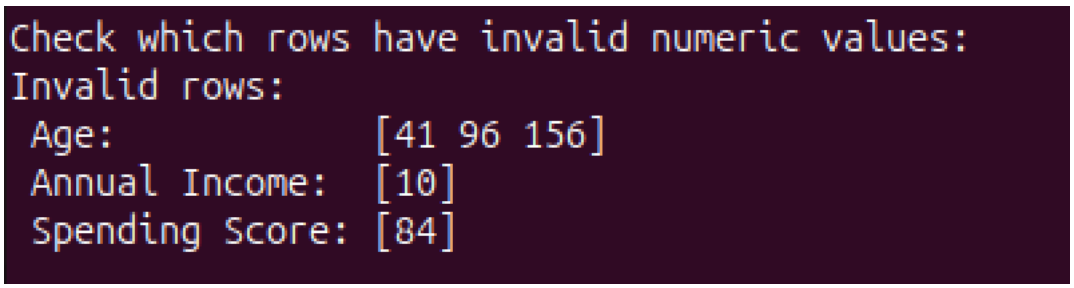


Figure 8 – Terminal Output Listing All Rows with Invalid Numeric Values Before Repair

These indices represent entries that require additional repairs. Including this screenshot in the report clearly demonstrates how the algorithm identifies problematic values before applying any corrections. Once these rows are flagged, Dijkstra's algorithm is used to compute their distances to all other rows in the feature graph. The algorithm then finds the nearest valid neighbour, and that row's corresponding value is used as the replacement. This strategy ensures that every correction is grounded in real data relationships, resulting in more trustworthy and naturally aligned numeric values across the cleaned dataset.

III. Why use Dijkstra's algorithm to repair numeric values

Before repairing the numeric attributes, it was important to choose a method that not only fixes invalid entries but also keeps the corrected values realistic. Simple approaches like replacing errors with the mean, median, or nearest hard-coded value. often distort the dataset and create unnatural patterns. Instead, I needed a method that considers the similarity between customers and uses patterns already present in the data.

To repair numeric values in a consistent and data-driven way, I represented the dataset as a feature graph. In this graph, each customer row becomes a node, and edges connect the row to its neighbouring row in the dataset. The weight of each edge is the geometric distance between the two rows in the three-dimensional feature space consisting of Age, Annual Income, and Spending Score. This distance captures how similar two customers are: rows with similar values lie close together, while rows with very different values lie farther apart. Since Dijkstra relies on weighted edges, each connection needed a distance value that reflects how similar or different two rows are.

- To compute this weight, I used the geometric distance (Euclidean distance) between two consecutive rows based on their three numerical attributes:
    1. Age
    2. Annual Income (k$)
    3. Spending Score (1–100)

$$\text{Distance} = \sqrt{(Age_1 - Age_2)^2 + (Income_1 - Income_2)^2 + (Score_1 - Score_2)^2}$$

Figure 9 – Distance Formula to which find to decide nearest behavior to peak

This means that two customers who have very similar numeric attributes will have a small edge weight, while two customers who differ a lot will have a larger weight. These distances were then placed between each pair of nodes, forming a weighted chain-like graph.

The figure below shows an example of how seven consecutive rows are represented in this feature graph. Each circle corresponds to a row (with its CustomerID labelled), and the numbers between nodes show the actual distance values computed using the Euclidean formula.

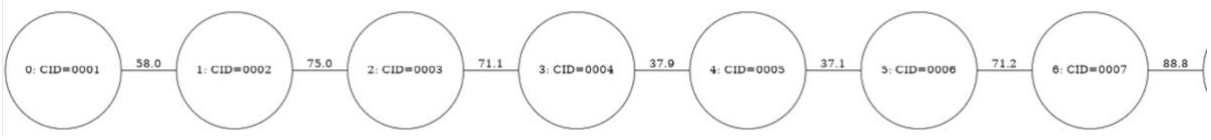


Figure 10 – Visualization of the Weighted Feature Graph Used for Dijkstra's Algorithm

Even though this graph is structured like a simple chain, it is extremely useful because it preserves the natural similarity patterns in the dataset. When Dijkstra's algorithm runs on this graph, it can identify the nearest valid row to any invalid one based on actual attribute similarities. This prevents unrealistic corrections and makes sure every replacement value remains consistent with the dataset's overall distribution. The small size of the graph also makes it efficient to compute shortest paths while still capturing meaningful numerical relationships.

This is why I used a graph-based approach. By turning the dataset into a feature graph where each customer row becomes a node and edges connect similar rows, the algorithm treats the dataset as a connected structure rather than a list of disconnected values. This ensures that every correction stays grounded in real relationships among the data points.

Dijkstra's algorithm fits this purpose perfectly because:

- It finds the closest valid row based on weighted similarity.
- The weights represent how different two rows are in Age, Income, and Spending Score.
- The nearest valid neighbour tends to have realistic values that align with the dataset's natural patterns.
- It avoids unrealistic jumps or introducing values that don't make sense.

So instead of guessing or imposing assumptions, the dataset repairs itself using the shortest path to the most similar correct record.

IV. Repairing numeric values using Dijkstra's algorithm

Once the graph was built, I applied Dijkstra's shortest path algorithm to repair invalid numeric entries. The intuition is straightforward: if a row contains an incorrect value, the algorithm should locate the closest valid row based on the feature distances. Dijkstra systematically explores all reachable nodes in increasing order of cost and identifies the nearest valid neighbor. That neighbor's numeric value is then used as the replacement. This ensures that the corrected values remain consistent with the dataset's distribution instead of being replaced by arbitrary or averaged numbers.

- These rows were then corrected using Dijkstra's results. For example, age values of 0 or 7 were replaced by valid ages from the most similar rows in the graph:

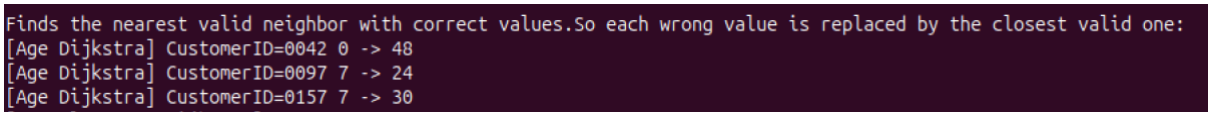


Figure 11 – Terminal Output Showing Dijkstra-Based Corrections Applied to Invalid Age Values

- The same approach was applied to correct income and spending scores:

Figure 12 – Terminal Output Showing Dijkstra-Based Corrections for Annual Income and Spending Score

These results demonstrate that every repaired value comes from a real, similar customer, making the corrections realistic and data driven. The use of Dijkstra's algorithm ensures accuracy, consistency, and fairness in the cleaning process far superior to hard-coded fixes or column averages.

3. CODING IMPLEMENTION

The code implementation is divided into three main stages: data cleaning, graph-based repair, and Huffman compression. In the data-cleaning part, the program reads the original "shopping_data.csv" file, converts the numeric columns to integers, and scans for obvious problems such as misspelled genres, negative ages, very small incomes, and spending scores outside the 1–100 range. After detecting these issues, the code builds the required graphs: a small typo graph for the Genre column and a feature graph for the numeric attributes. These structures are then used to apply BFS for correcting categorical values and Dijkstra's algorithm for repairing numeric values in a systematic way. Finally, once the dataset is fully cleaned, the program runs Huffman coding on the entire CSV text to produce the encoded bitstream, the decoded file, and the Huffman dictionary table, along with a compression report.

To satisfy the assignment requirement of using two programming languages and implemented this logic in both Python and Go. Python is used as the main language because it allows fast prototyping and easy handling of CSV files, lists, dictionaries, and frequency counts. It is also convenient for printing intermediate results and generating the Huffman statistics. After the Python version was working correctly, I wrote a corresponding Go implementation that follows the same steps and function structure loading the data, constructing graphs, running BFS and Dijkstra, and performing Huffman encoding. This two-language approach shows that the solution is not tied to a single environment and that the core ideas (graph construction, shortest path search, and prefix-free coding) can be reproduced in different programming ecosystems.

The code is organized into small functions so that each algorithmic step is clear and easy to test. The overall flow of the program can be summarized as follows:

1. Loading the dataset

The program starts by reading the shopping_data.csv file using csv.DictReader. While loading, it immediately converts the three numeric attributes Age, Annual Income (k$), and Spending Score (1–100) from strings to integers. This avoids type issues later when performing arithmetic and distance calculations. All rows are stored as dictionaries inside a list so that individual columns can be accessed by name.

2. Fixing Genre typos using a graph and BFS

Next, the code collects all distinct values from the Genre column and builds a small typo graph. Each unique string becomes a node, and an edge is added between two nodes if they differ by only one character (for example, “Fmale” and “Female”). After this graph is built, the script runs Breadth-First Search from any Genre value that is not exactly “Male” or “Female”. BFS walks through the neighbouring typo nodes until it reaches one of the valid labels and then replaces the original value with that correct spelling. The terminal output prints each correction along with the corresponding CustomerID, so we can verify that all misspellings like “Mle”, “Mal”, “Femae”, and “Fmale” are fixed.

3. Constructing the graph

Once the categorical column is cleaned, the script turns the entire dataset into a feature graph for numeric repair. Each row is treated as a node, and edges connect row i to row i+1. The weight on an edge is the geometric distance in the three-dimensional space formed by Age, Income, and Spending Score. This step encodes how similar two neighbouring customers are.

Before using this graph, the script also performs a small preprocessing rule: if Age is negative, it replaces it with its absolute value (for example, –19 becomes 19), since negative ages are impossible.

4. Detecting invalid numeric values

With the graph ready, the Python code scans all rows to find entries that violate basic constraints:

Age not in the range [18, 70]

Annual Income not in the range [10, 200]

Spending Score not in the range [1, 100]

The indices of rows that fail these checks are printed under "Invalid rows" in the terminal. This makes it easy to see which records will be repaired by Dijkstra's algorithm.

5. Repairing Age, Income, and Score using Dijkstra's algorithm

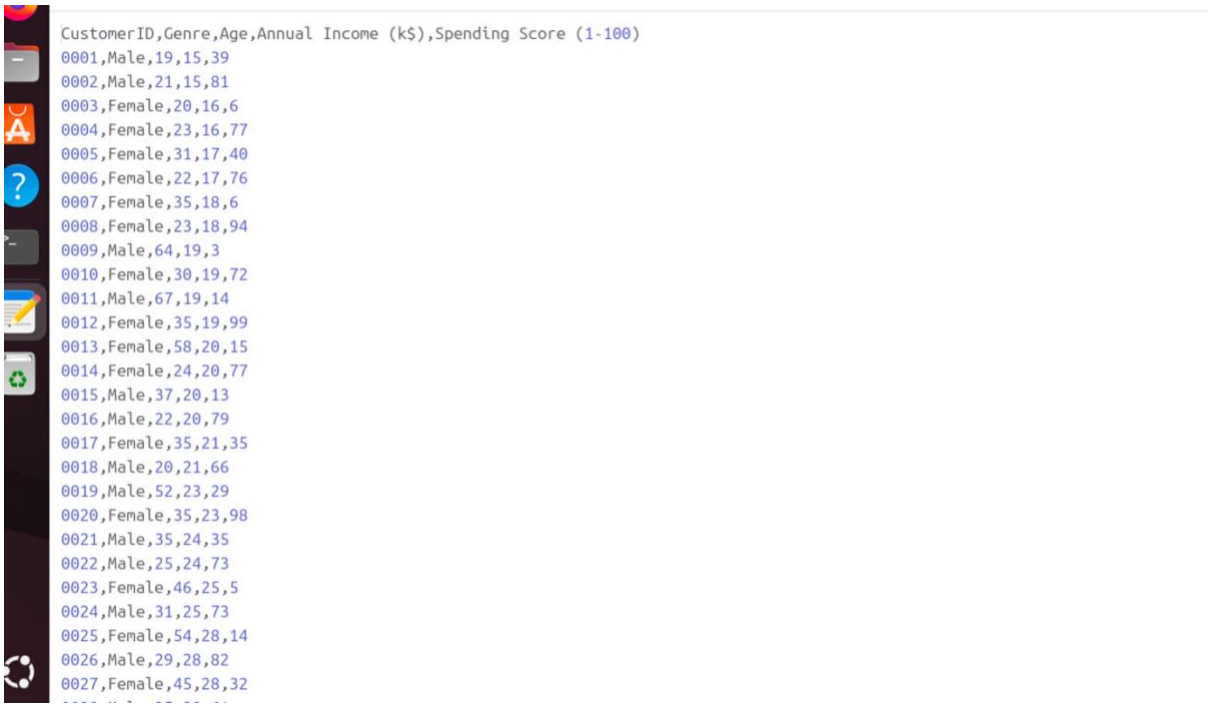
For each invalid row, the script runs Dijkstra's shortest path algorithm on the feature graph, starting from that row's node. The algorithm computes the distance from the invalid row to every other row. Among all rows that have a valid value for the attribute being fixed (Age, Income, or Score), the program chooses the row with the smallest distance. The incorrect value is then replaced by this neighbour's value. The script prints messages such as

"[Age Dijkstra] CustomerID=0042 0 -> 48"

so, we can see exactly how each invalid entry was corrected and from which nearby row the replacement came.

6. Saving the cleaned dataset

After all numeric repairs are completed, the cleaned rows are written back to a new file called shopping_data_cleaned.csv. This CSV contains the corrected Genre labels plus fixed Age, Income, and Spending Score columns. At this point, there are no remaining negative ages, invalid incomes, or out-of-range scores.



```
CustomerID,Genre,Age,Annual Income (k$),Spending Score (1-100)
0001,Male,19,15,39
0002,Male,21,15,81
0003,Female,20,16,6
0004,Female,23,16,77
0005,Female,31,17,40
0006,Female,22,17,76
0007,Female,35,18,6
0008,Female,23,18,94
0009,Male,64,19,3
0010,Female,30,19,72
0011,Male,67,19,14
0012,Female,35,19,99
0013,Female,58,20,15
0014,Female,24,20,77
0015,Male,37,20,13
0016,Male,22,20,79
0017,Female,35,21,35
0018,Male,20,21,66
0019,Male,52,23,29
0020,Female,35,23,98
0021,Male,35,24,35
0022,Male,25,24,73
0023,Female,46,25,5
0024,Male,31,25,73
0025,Female,54,28,14
0026,Male,29,28,82
0027,Female,45,28,32
```

Figure 13 – Preview of the Final Cleaned shopping_data_cleaned.csv File

I. Python Code Output

When the Python script is executed in the Ubuntu terminal, it begins by loading all 200 rows from shopping_data.csv and extracting the unique Genre values. The program then builds the edit-distance graph and applies BFS to correct misspelled categories such as “Mle,” “Mal,” “Fmale,” and “Femae,” mapping them back to the correct labels (“Male” or “Female”). The output printed in the terminal shows the constructed typo graph as well as each BFS fix applied to the dataset.

After the categorical corrections, the program proceeds with numeric validation. It first handles impossible values such as negative ages (e.g., converting -19 → 19) and then identifies all rows that contain invalid ages, incomes, or spending scores. These invalid row indices are displayed clearly in the output.

Next, Dijkstra’s algorithm is run to locate the closest valid neighbor for each invalid row. The terminal logs each automatic correction, such as:

Age corrected: 0 → 48, 7 → 24, 7 → 30

Income corrected: 1 → 19

Score corrected: 0 → 44

Finally, the script saves the cleaned dataset as shopping_data_cleaned.csv, confirming that all values are repaired and consistent. Figures 14 and 15 show the step-by-step execution of this process inside the terminal.

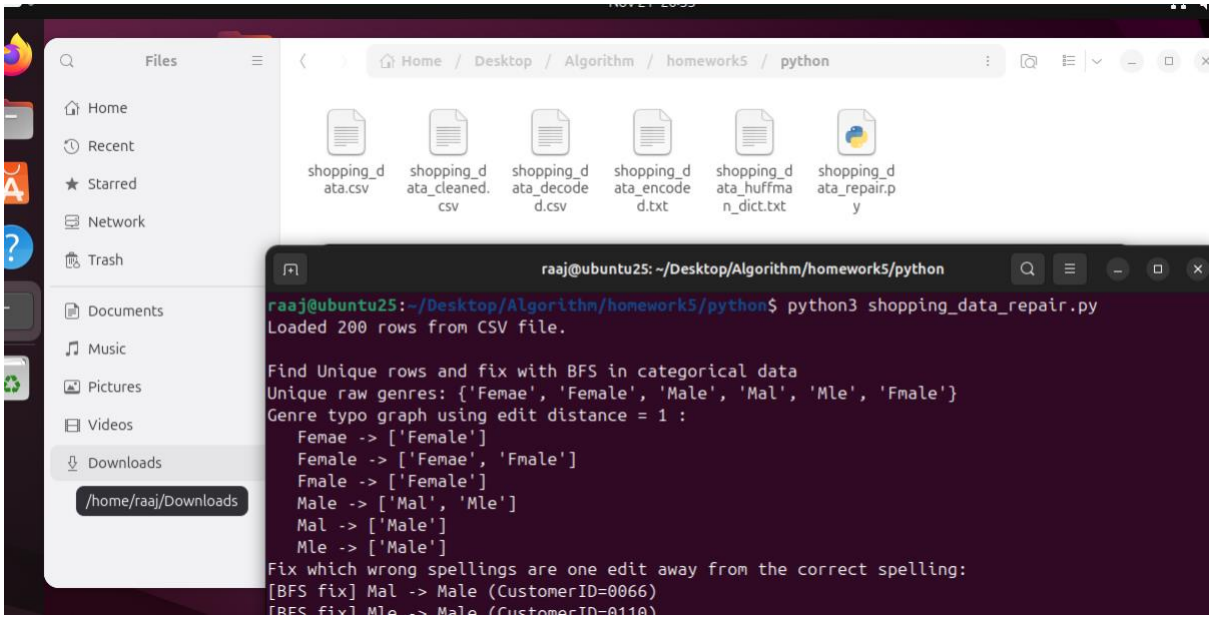


Figure 14 – Initial Execution of Python code in ubuntu

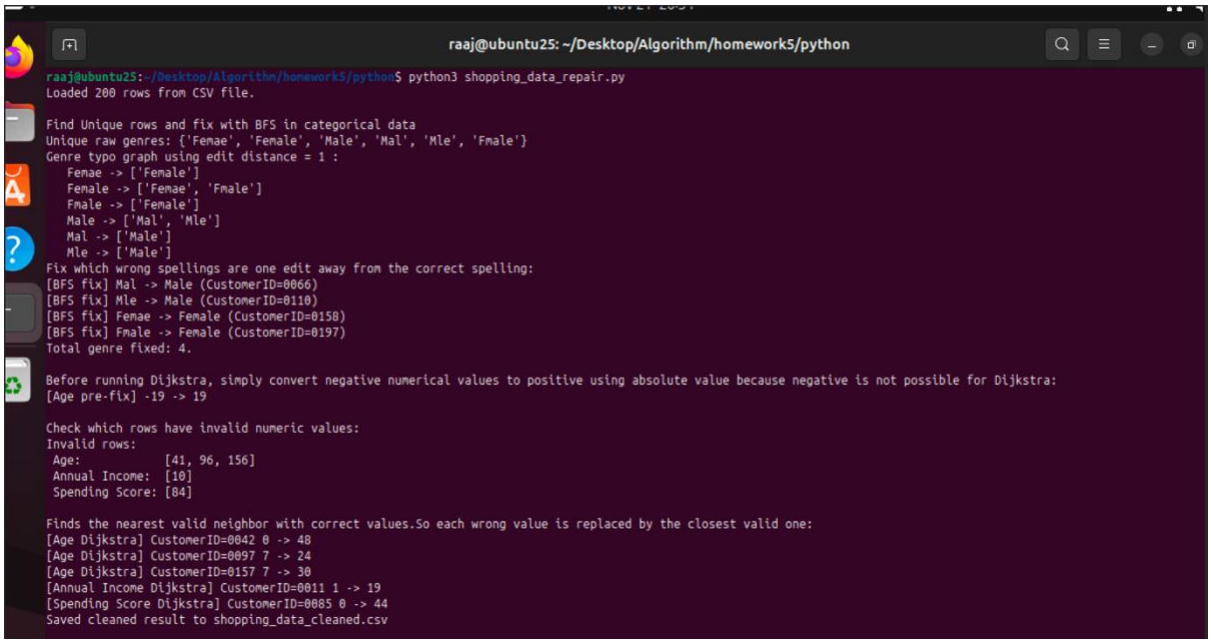


Figure 15 – Terminal Output Showing Numeric Error Detection and Dijkstra-Based Repairs in Python

II. Go Code Output

After verifying the Python implementation, I also implemented the same data-cleaning workflow in Go to satisfy the requirement of writing the program in two different programming languages. The Go version performs the same sequence of steps: reading the CSV file, fixing typos in the Genre column using BFS, detecting invalid numeric values, and correcting them using Dijkstra's algorithm.

When the Go program is executed using go run, the terminal output mirrors the Python results. It begins by loading all 200 rows from the CSV and printing the set of unique Genre values. The program then constructs the edit-distance graph and applies BFS to convert incorrect spellings like "Mle," "Mal," "Femae," and "Fmale" into their correct forms.

Next, the Go script identifies negative, zero, or out-of-range numeric values and prints the list of invalid row indices. It applies the same preprocessing rule as the Python version (for example, converting -19 → 19) before running Dijkstra's algorithm. The output shows each correction step:

Age corrections (e.g., 0 → 48, 7 → 24, 7 → 30)
Income correction (1 → 19)
Spending Score correction (0 → 44)

Finally, the cleaned dataset is saved to shopping_data_cleaned.csv, confirming that both implementations produce identical results. Figures 16 and 17 illustrate the full Go program execution in the terminal.

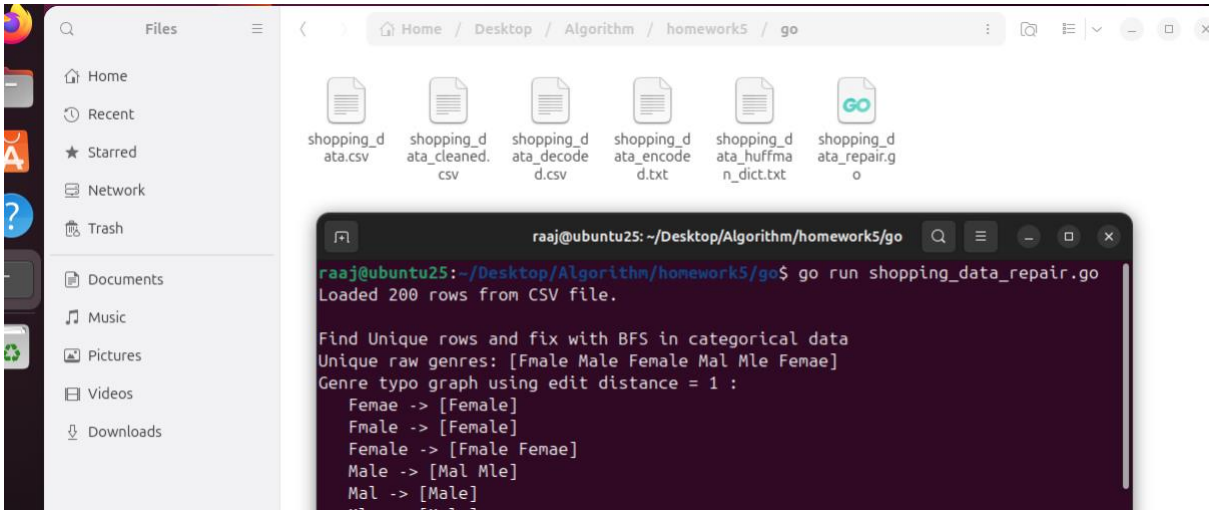


Figure 16 – Initial Terminal Output of Go Implementation

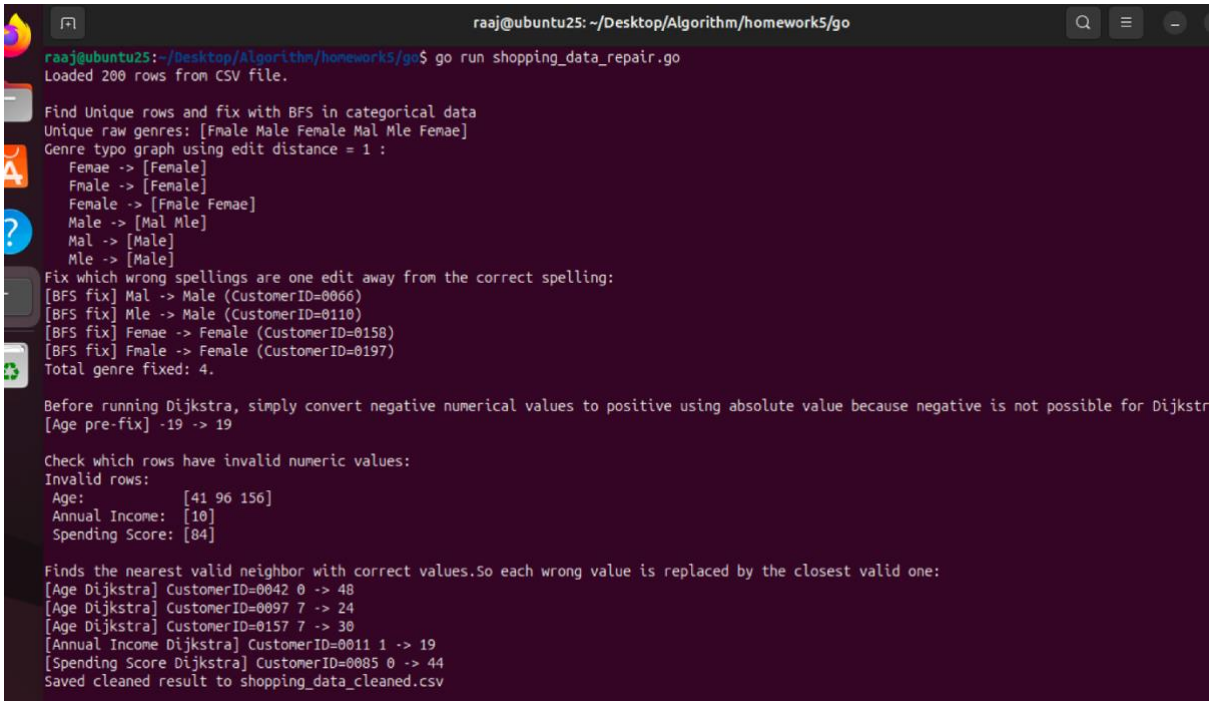


Figure 17 – Terminal Output Showing Numeric Error Detection and Dijkstra-Based Repairs in Go

III. Comparison of Original vs. Cleaned Data

The combined figure presents a clear visual comparison of the dataset before and after applying the graph-based cleaning algorithms. In the top-left plot, the Genre distribution shows how the original dataset contained several misspelled categories such as *Femae*, *Fmale*, *Mal*, and *Mle*. After correction, the cleaned version contains only the valid labels *Male* and *Female*, demonstrating that the BFS edit-distance approach successfully removed all typographical inconsistencies. The top-right plot compares Age distributions, where the original histogram includes invalid values such as –19 and 0. In the cleaned dataset, these unrealistic ages are replaced with appropriate values using Dijkstra's nearest-neighbour repair, causing the gold "invalid spikes" to disappear while preserving the natural shape of the age distribution. The bottom-left histogram shows Annual Income values before and after cleaning. The original data includes extreme outliers, such as an income of 1 k$, whereas the cleaned data eliminates these anomalies and aligns all entries within the valid 10–200 k$ range. Finally, the bottom-right histogram illustrates the Spending Score distribution. The original data includes a score of 0, which lies outside the valid 1–100 range. After repair, the cleaned distribution removes these invalid points but still follows the overall pattern of the original dataset. Together, these four visualizations confirm that the cleaning process corrected errors without distorting the underlying trends in the data.

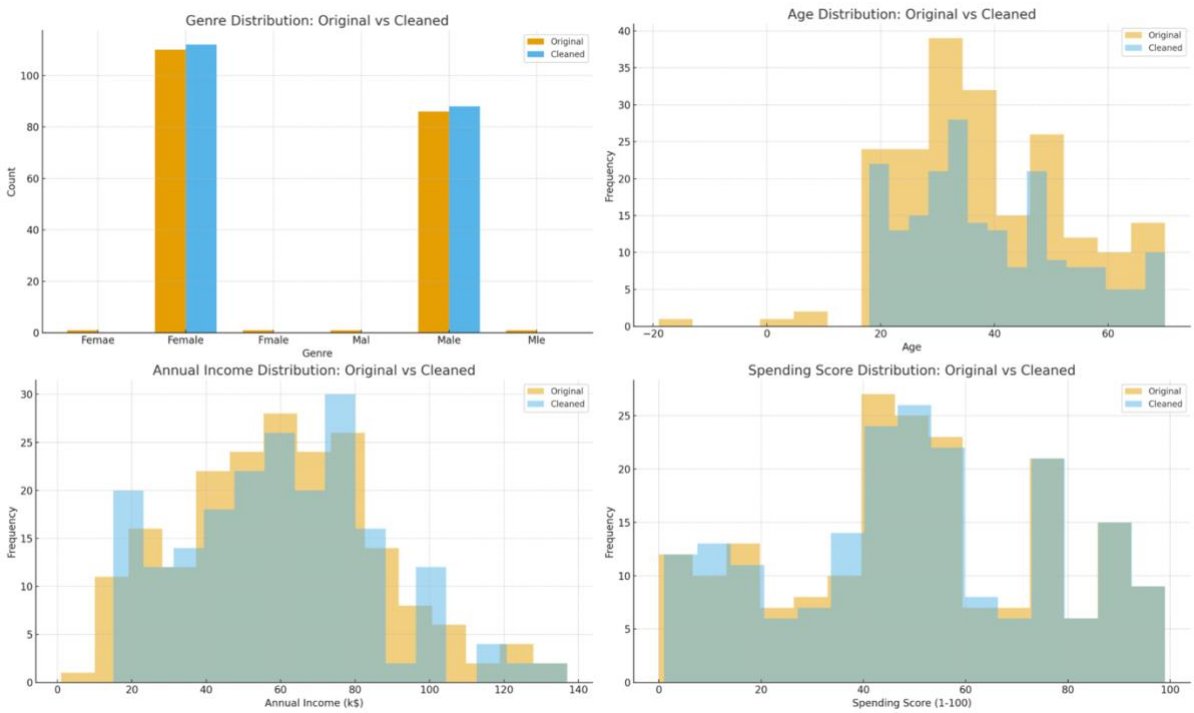


Figure 18 – Original data vs After fix data for all attributes.

HYPERLINK TO ONLINE GDB

The complete source code for both the Python and Go implementations has been uploaded to OnlineGDB, which allows to run the code directly in a browser. A free account may be required for smooth execution.

Python code : https://onlinegdb.com/d_ILHeose0

Go code : https://onlinegdb.com/zhZCa7nr_

DISCUSSION

This homework brought together several concepts from graph algorithms, error detection, and data compression, and applying them to a real dataset highlighted how these techniques complement each other. The overall cleaning process showed that even small inconsistencies such as misspelled categorical values or slightly incorrect numeric entries can have a noticeable impact on later analysis. Using a systematic approach allowed every correction to be justified instead of relying on manual guesses or arbitrary adjustments.

One important observation was how well simple graph structures can model relationships inside a dataset. The typo-correction graph for the Genre column was tiny, but BFS handled the problem effectively by always finding the closest correct spelling. This demonstrated how graph traversal can be used in data-cleaning tasks that at first glance seem unrelated to graph theory.

The graph used for numeric repair showed a different kind of strength. Instead of treating each attribute in isolation, the weighted edges captured the combined similarity of Age, Income, and Spending Score. Dijkstra's algorithm then used this structure to make realistic corrections based on the nearest valid row. This approach produced repairs that were not only valid but also consistent with the dataset's natural patterns. Comparing this to a simple unweighted graph helped reinforce why distances matter when cleaning numeric data.

Taken together, these tasks demonstrated how multiple algorithms BFS, Dijkstra, graph construction, can work together to enhance the quality and efficiency of data processing. Beyond completing the assignment, this workflow reflects real-world practices in data

engineering, where data must often be cleaned, corrected and validated, before it can be used for meaningful analysis.

LIMITATIONS AND FUTUREWORK

Although the cleaning pipeline worked reliably for this dataset, there are several limitations that are worth noting. The graph used for numeric repair is based on a simple chain structure, where each row is only connected to its immediate neighbors. While this approach is efficient and easy to implement, it may not fully capture deeper relationships in larger or more complex datasets. If two customers who are very similar happen to be far apart in the CSV ordering, the algorithm cannot find that connection, which may lead to suboptimal corrections. A more expressive graph such as connecting each row to its k-nearest neighbors could address this limitation but would require additional computation.

Another limitation comes from the use of Dijkstra's algorithm with Euclidean distance. Although this technique produces realistic replacements, it assumes that all three numeric features contribute equally to similarity. In practice, Age, Income, and Spending Score may not have the same importance. A weighted distance metric or a normalization step could give more balanced results. Additionally, the algorithm directly copies values from the nearest valid row. While this works well here, in real applications, interpolating or averaging values might create smoother adjustments, especially when multiple columns are invalid simultaneously.

The typo-correction method, based on edit distance and BFS, also has its constraints. It works perfectly for a small set of categorical labels like "Male" and "Female," but it may not scale as effectively if a column contains many categories or if typos are more severe than a single-character difference. For such cases, a dictionary-based spell-checker or a machine-learning model trained on valid labels could improve accuracy.

In future work, the entire pipeline could be extended into a more general data-cleaning framework. Possible enhancements include adding anomaly detection, integrating outlier analysis, supporting multiple graph structures, and incorporating visual dashboards to show which rows were corrected and why. Automating hyperparameter choices such as the distance metric, validation ranges, or graph construction rules would make the system more adaptable to new datasets without manual tuning. Finally, applying the same methodology to a larger or noisier real-world dataset would provide a stronger evaluation of its robustness and scalability.

CONCLUSION

This homework demonstrated how graph-based methods can be used to clean and repair real-world datasets in a structured and explainable way. Even though the errors in the shopping_data.csv file were relatively small mostly misspelled Genre labels and a handful of unrealistic numeric values they clearly showed how incorrect data can affect downstream analysis if not handled properly. By approaching the problem through graph algorithms, each correction could be justified using clear relationships instead of arbitrary rules.

The use of a typo graph combined with Breadth-First Search provided a reliable way to correct categorical mistakes. Because the graph was built using edit distance, the algorithm naturally corrected values like “Mle” and “Fmale” to their proper forms without relying on manual mapping. This step highlighted how even a simple graph can capture meaningful structure in categorical data. For numeric attributes, constructing a feature graph allowed the cleaning process to consider the similarity between customer records. Applying Dijkstra’s algorithm ensured that invalid values were replaced using the nearest valid row in the dataset, based on a distance measure that reflects how similar two customers are in terms of Age, Annual Income, and Spending Score. This produced corrections that were not only valid but also aligned with the natural pattern of the data.

Overall, the assignment made it clear that graph algorithms are effective tools for data cleaning. especially when the goal is to make repairs that are transparent, reproducible, and grounded in the structure of the dataset. The workflow developed here can be extended to more complex datasets and additional types of errors, and it provides a strong foundation for building more advanced data-quality systems in the future.

ACKNOWLEDGE

I would like to sincerely thank Prof. Dr. David Dai for his guidance and support throughout this assignment. His clear explanations of BFS, Dijkstra's algorithm, and graph-based thinking made a huge difference in my understanding of how these techniques can be applied to real-world data cleaning problems. The examples he demonstrated in class especially the way he connected small typos or numeric mistakes to graph traversal strategies helped me build the foundation for my own implementation in this homework.

I am also grateful for the insights he shared during the class. His breakdown of how to detect errors, construct meaningful graphs, and choose the correct algorithm for each type of data issue directly influenced the structure of my solution. The approach I used in this homework was strongly inspired by his teaching, and it helped me gain a deeper appreciation for how powerful and flexible graph algorithms can be when used creatively. This assignment not only strengthened my algorithmic thinking but also gave me practical experience that I will carry into future coursework and projects.

REFERENCES

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. — Used for understanding BFS and Dijkstra's shortest path algorithm.
2. Kleinberg, J., & Tardos, É. (2005). *Algorithm Design*. Pearson/Addison-Wesley. — Provided conceptual background on graph modelling and weighted shortest paths.
3. Wagner, R. A., & Fischer, M. J. (1974). "The String-to-String Correction Problem." *Journal of the ACM, 21*(1), 168–173. — Source of the edit-distance concept used for Genre typo correction.
4. Python Software Foundation. *Python 3 Standard Library Documentation*. https://docs.python.org/3/ — Used for CSV handling, data structures (dict, list), and heap-based priority queue.
5. Go Authors. *The Go Programming Language Specification*. https://go.dev/doc/ — Referenced for implementing the Go version of the assignment.
6. Class Lecture Notes – CS51510 (November 17, 2025). — Provided the initial guidelines for identifying errors, constructing graphs, and choosing BFS and Dijkstra's algorithm for correction tasks.
7. OnlineGDB — Used as the integrated environment for compiling and testing both Python and Go implementations.

APPENDIX

I. Python code

```
import csv
import math
import heapq
from collections import defaultdict, Counter, deque

# file names (just to use below)
INPUT_CSV = "shopping_data.csv"
CLEANED_CSV = "shopping_data_cleaned.csv"

# load rows from csv
def load_rows(path):
    # read csv and convert some cols to int
    rows = []
    with open(path, newline='') as f:
        reader = csv.DictReader(f)
        for row in reader:
            # converting here coz otherwise string mess later
            row["Age"] = int(row["Age"])
            row["Annual Income (k$)"] = int(row["Annual Income (k$)"])
            row["Spending Score (1-100)"] = int(row["Spending Score (1-100)"])
            rows.append(row)
    return rows

# save final cleaned csv
def save_rows(path, rows):
    if not rows:
        return
    fieldnames = ["CustomerID", "Genre", "Age",
                  "Annual Income (k$)", "Spending Score (1-100)"]
    with open(path, "w", newline='') as f:
        w = csv.DictWriter(f, fieldnames=fieldnames)
```

```
        w.writeheader()
        for r in rows:
            # writing cleaned data back
            w.writerow({
                "CustomerID": r["CustomerID"],
                "Genre": r["Genre"],
                "Age": int(r["Age"]),
                "Annual Income (k$)": int(r["Annual Income (k$)"]),
                "Spending Score (1-100)": int(r["Spending Score (1-100)"])
            })



# BFS + edit distance to fix wrong spelling in Genre
def levenshtein(a, b):
    # simple edit dist fn, used to check typo diff
    dp = [[0] * (len(b) + 1) for _ in range(len(a) + 1)]
    for i in range(len(a) + 1):
        dp[i][0] = i
    for j in range(len(b) + 1):
        dp[0][j] = j
    for i in range(1, len(a) + 1):
        for j in range(1, len(b) + 1):
            cost = 0 if a[i - 1] == b[j - 1] else 1
            dp[i][j] = min(
                dp[i - 1][j] + 1,
                dp[i][j - 1] + 1,
                dp[i - 1][j - 1] + cost
            )
    return dp[-1][-1]

def build_genre_graph(genres):
    # build small graph where node=string, and edge if edit dist is 1
    genres = list(genres)
    g = defaultdict(list)
```

```
    for i in range(len(genres)):
        for j in range(i + 1, len(genres)):
            a, b = genres[i], genres[j]
            if levenshtein(a, b) == 1:
                g[a].append(b)
                g[b].append(a)
    return g

def bfs_fix_genre(cur, graph, valid):
    # try to find closest valid male/female by BFS
    if cur in valid:
        return cur
    seen = {cur}
    q = deque([cur])
    while q:
        u = q.popleft()
        if u in valid:
            return u
        for v in graph.get(u, []):
            if v not in seen:
                seen.add(v)
                q.append(v)
    # nothing found so maybe no path so just keep same
    return cur

def fix_genres(rows):
    # collect all weird gender values
    all_g = {r["Genre"] for r in rows}
    valid = {"Male", "Female"}
    graph = build_genre_graph(all_g)

    print("Unique raw genres:", all_g)
    print("Genre typo graph using edit distance = 1 :")
    for k, vs in graph.items():
```

```
        print("  ", k, "->", vs)

    print(f"Fix which wrong spellings are one edit away from the correct spelling:")
    changes = 0
    for r in rows:
        old = r["Genre"]
        new = bfs_fix_genre(old, graph, valid)
        if new != old:
            changes += 1
            if(old != "Female" or "Male"):
                print(f"[BFS fix] {old} -> {new} (CustomerID={r['CustomerID']})")
            r["Genre"] = new

    print(f"Total genre fixed: {changes}. \n")



#  Dijkstra for numeric repair for Age / Income / Score ====

def feat_vec(row):
    # convert row to int tuple for dist calculation
    return (
        int(row["Age"]),
        int(row["Annual Income (k$)"]),
        int(row["Spending Score (1-100)"])
    )

def num_dist(r1, r2):
    # distance in 3D space
    a1, i1, s1 = feat_vec(r1)
    a2, i2, s2 = feat_vec(r2)
    return math.sqrt((a1 - a2)**2 + (i1 - i2)**2 + (s1 - s2)**2)

def build_graph(rows):
    # making simple chain graph: each row connected to next
```

```
    n = len(rows)
    adj = [[] for _ in range(n)]
    for i in range(n - 1):
        w = num_dist(rows[i], rows[i+1])
        adj[i].append((i+1, w))
        adj[i+1].append((i, w))
    return adj

def dijkstra(adj, start):
    # shortest path for positive weight
    n = len(adj)
    dist = [math.inf] * n
    dist[start] = 0.0
    pq = [(0.0, start)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue
        for v, w in adj[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                heapq.heappush(pq, (nd, v))
    return dist

# small validation fn
def valid_age(a):
    return 18 <= a <= 70

def valid_inc(i):
    return 10 <= i <= 200

def valid_score(s):
    return 1 <= s <= 100
```

```
def repair_numeric(rows, adj):
    print("Before running Dijkstra, simply convert negative numerical values to positive using absolute value because negative is not possible for Dijkstra:")
    # fix negative ages first (simple rule)
    for r in rows:
        if r["Age"] < 0:
            print(f"[Age pre-fix] {r['Age']} -> {abs(r['Age'])}")
            r["Age"] = abs(r["Age"])

    # collect invalid positions
    bad_age = []
    bad_inc = []
    bad_score = []
    for i, r in enumerate(rows):
        if not valid_age(r["Age"]):
            bad_age.append(i)
        if not valid_inc(r["Annual Income (k$)"]):
            bad_inc.append(i)
        if not valid_score(r["Spending Score (1-100)"]):
            bad_score.append(i)
    print("\nCheck which rows have invalid numeric values:")
    print("Invalid rows:")
    print(" Age:           ", bad_age)
    print(" Annual Income: ", bad_inc)
    print(" Spending Score:", bad_score)
    print("\nFinds the nearest valid neighbor with correct values.So each wrong value is replaced by the closest valid one:")

    # fix ages
    for idx in bad_age:
        dist = dijkstra(adj, idx)
        best_j = None
        best_d = math.inf
```

```
        for j, r in enumerate(rows):
            if j != idx and valid_age(r["Age"]):
                if dist[j] < best_d:
                    best_d = dist[j]
                    best_j = j
        if best_j is not None:
            old = rows[idx]["Age"]
            rows[idx]["Age"] = rows[best_j]["Age"]
            print(f"[Age Dijkstra] CustomerID={rows[idx]['CustomerID']} {old} ->
{rows[idx]['Age']}")



    # fix incomes
    for idx in bad_inc:
        dist = dijkstra(adj, idx)
        best_j = None
        best_d = math.inf
        for j, r in enumerate(rows):
            if j != idx and valid_inc(r["Annual Income (k$)"]):
                if dist[j] < best_d:
                    best_d = dist[j]
                    best_j = j
        if best_j is not None:
            old = rows[idx]["Annual Income (k$)"]
            rows[idx]["Annual Income (k$)"] = rows[best_j]["Annual Income (k$)"]
            print(f"[Annual Income Dijkstra] CustomerID={rows[idx]['CustomerID']} {old} ->
{rows[idx]['Annual Income (k$)']}")

    # fix spending score
    for idx in bad_score:
        dist = dijkstra(adj, idx)
        best_j = None
        best_d = math.inf
        for j, r in enumerate(rows):
```

```
            if j != idx and valid_score(r["Spending Score (1-100)"]):
                if dist[j] < best_d:
                    best_d = dist[j]
                    best_j = j
        if best_j is not None:
            old = rows[idx]["Spending Score (1-100)"]
            rows[idx]["Spending Score (1-100)"] = rows[best_j]["Spending Score (1-100)"]
            print(f"[Spending Score Dijkstra] CustomerID={rows[idx]['CustomerID']} {old} ->
{rows[idx]['Spending Score (1-100)']}")



# main

def main():
    rows = load_rows(INPUT_CSV)
    print(f"Loaded {len(rows)} rows from CSV file.\n")


    # BFS
    print(f"Find Unique rows and fix with BFS in categorical data")
    fix_genres(rows)
    # graph for dijkstra
    adj = build_graph(rows)
    # Dijkstra fixes
    repair_numeric(rows, adj)

    save_rows(CLEANED_CSV, rows)
    print(f"Saved cleaned result to {CLEANED_CSV}\n")


if __name__ == "__main__":
    main()
```

II. Go code

```
package main

import (
	"container/heap"
	"encoding/csv"
	"fmt"
	"log"
	"math"
	"os"
	"strings"
)

// file names (just to use below)
const (
	INPUT_CSV       = "shopping_data.csv"
	CLEANED_CSV     = "shopping_data_cleaned.csv"
)

// Row struct similar to python dict row
type Row struct {
	CustomerID     string
	Genre          string
	Age            int
	AnnualIncome   int
	SpendingScore  int
}

// load rows from csv
func load_rows(path string) []Row {
	// read csv and convert some cols to int
	f, err := os.Open(path)
	if err != nil {
```

```
		log.Fatalf("cannot open %s: %v", path, err)
	}
	defer f.Close()

	r := csv.NewReader(f)
	records, err := r.ReadAll()
	if err != nil {
		log.Fatalf("csv read error: %v", err)
	}

	if len(records) < 1 {
		return []Row{}
	}

	// header indices
	header := records[0]
	index := make(map[string]int)
	for i, h := range header {
		index[h] = i
	}

	var rows []Row
	for _, rec := range records[1:] {
		if len(rec) < len(header) {
			continue
		}
		age := atoi(rec[index["Age"]])
		inc := atoi(rec[index["Annual Income (k$)"]])
		score := atoi(rec[index["Spending Score (1-100)"]])
		rows = append(rows, Row{
			CustomerID:    rec[index["CustomerID"]],
			Genre:         rec[index["Genre"]],
			Age:           age,
			AnnualIncome:  inc,
```

```
                        SpendingScore: score,
                })
        }
        return rows
}

// save final cleaned csv
func save_rows(path string, rows []Row) {
        if len(rows) == 0 {
                return
        }
        f, err := os.Create(path)
        if err != nil {
                log.Fatalf("cannot create %s: %v", path, err)
        }
        defer f.Close()

        w := csv.NewWriter(f)
        defer w.Flush()

        header := []string{"CustomerID", "Genre", "Age", "Annual Income (k$)", "Spending
Score (1-100)"}
        if err := w.Write(header); err != nil {
                log.Fatalf("csv write err: %v", err)
        }

        for _, r := range rows {
                rec := []string{
                        r.CustomerID,
                        r.Genre,
                        fmt.Sprintf("%d", r.Age),
                        fmt.Sprintf("%d", r.AnnualIncome),
                        fmt.Sprintf("%d", r.SpendingScore),
                }
```

```
                if err := w.Write(rec); err != nil {
                        log.Fatalf("csv write err: %v", err)
                }
        }
}

// ==== BFS + edit distance to fix wrong spelling in Genre ====

func levenshtein(a, b string) int {
        // simple edit dist fn, used to check typo diff
        la := len(a)
        lb := len(b)
        dp := make([][]int, la+1)
        for i := range dp {
                dp[i] = make([]int, lb+1)
        }
        for i := 0; i <= la; i++ {
                dp[i][0] = i
        }
        for j := 0; j <= lb; j++ {
                dp[0][j] = j
        }
        for i := 1; i <= la; i++ {
                for j := 1; j <= lb; j++ {
                        cost := 1
                        if a[i-1] == b[j-1] {
                                cost = 0
                        }
                        dp[i][j] = min3(
                                dp[i-1][j]+1,
                                dp[i][j-1]+1,
                                dp[i-1][j-1]+cost,
                        )
                }
```

```
	}
	return dp[la][lb]
}

func build_genre_graph(genres []string) map[string][]string {
	// build small graph where node=string, and edge if edit dist is 1
	g := make(map[string][]string)
	for i := 0; i < len(genres); i++ {
		for j := i + 1; j < len(genres); j++ {
			a := genres[i]
			b := genres[j]
			if levenshtein(a, b) == 1 {
				g[a] = append(g[a], b)
				g[b] = append(g[b], a)
			}
		}
	}
	return g
}

func bfs_fix_genre(cur string, graph map[string][]string, valid map[string]bool) string {
	// try to find closest valid male/female by BFS
	if valid[cur] {
		return cur
	}
	seen := map[string]bool{cur: true}
	q := []string{cur}
	for len(q) > 0 {
		u := q[0]
		q = q[1:]
		if valid[u] {
			return u
		}
		for _, v := range graph[u] {
```

```
                        if !seen[v] {
                                seen[v] = true
                                q = append(q, v)
                        }
                }
        }
        // nothing found so maybe no path so just keep same
        return cur
}

func fix_genres(rows []Row) {
        // collect all weird gender values
        uniqSet := make(map[string]bool)
        for _, r := range rows {
                uniqSet[r.Genre] = true
        }
        var all_g []string
        for k := range uniqSet {
                all_g = append(all_g, k)
        }

        valid := map[string]bool{"Male": true, "Female": true}
        graph := build_genre_graph(all_g)

        fmt.Println("Unique raw genres:", all_g)
        fmt.Println("Genre typo graph using edit distance = 1 :")
        for k, vs := range graph {
                fmt.Println("  ", k, "->", vs)
        }

        fmt.Println("Fix which wrong spellings are one edit away from the correct spelling:")
        changes := 0
        for i := range rows {
                old := rows[i].Genre
```

```
                newVal := bfs_fix_genre(old, graph, valid)
                if newVal != old {
                        changes++
                        fmt.Printf("[BFS fix] %s -> %s (CustomerID=%s)\n", old, newVal,
rows[i].CustomerID)
                        rows[i].Genre = newVal
                }
        }
        fmt.Printf("Total genre fixed: %d.\n\n", changes)
}

// ==== Dijkstra for numeric repair for Age / Income / Score ====

func feat_vec(row Row) (int, int, int) {
        // convert row to int tuple for dist calculation
        return row.Age, row.AnnualIncome, row.SpendingScore
}

func num_dist(r1, r2 Row) float64 {
        // distance in 3D space
        a1, i1, s1 := feat_vec(r1)
        a2, i2, s2 := feat_vec(r2)
        return math.Sqrt(float64((a1-a2)*(a1-a2) + (i1-i2)*(i1-i2) + (s1-s2)*(s1-s2)))
}

type Edge struct {
        to int
        w  float64
}

func build_graph(rows []Row) [][]Edge {
        // making simple chain graph: each row connected to next
        n := len(rows)
        adj := make([][]Edge, n)
```

```
	for i := 0; i < n-1; i++ {
		w := num_dist(rows[i], rows[i+1])
		adj[i] = append(adj[i], Edge{to: i + 1, w: w})
		adj[i+1] = append(adj[i+1], Edge{to: i, w: w})
	}
	return adj
}

func dijkstra(adj [][]Edge, start int) []float64 {
	// shortest path for positive weight
	n := len(adj)
	dist := make([]float64, n)
	for i := range dist {
		dist[i] = math.Inf(1)
	}
	dist[start] = 0.0
	pq := &MinHeap{}
	heap.Init(pq)
	heap.Push(pq, HeapItem{node: start, dist: 0.0})

	for pq.Len() > 0 {
		item := heap.Pop(pq).(HeapItem)
		u := item.node
		d := item.dist
		if d > dist[u] {
			continue
		}
		for _, e := range adj[u] {
			nd := d + e.w
			if nd < dist[e.to] {
				dist[e.to] = nd
				heap.Push(pq, HeapItem{node: e.to, dist: nd})
			}
		}
```

```
	}
	return dist
}

// small validation fn
func valid_age(a int) bool {
	return 18 <= a && a <= 70
}
func valid_inc(i int) bool {
	return 10 <= i && i <= 200
}
func valid_score(s int) bool {
	return 1 <= s && s <= 100
}

func repair_numeric(rows []Row, adj [][]Edge) {
	fmt.Println("Before running Dijkstra, simply convert negative numerical values to positive using absolute value because negative is not possible for Dijkstra:")

	// fix negative ages first (simple rule)
	for i := range rows {
		if rows[i].Age < 0 {
			fmt.Printf("[Age pre-fix] %d -> %d\n", rows[i].Age, -rows[i].Age)
			rows[i].Age = -rows[i].Age
		}
	}

	// collect invalid positions
	var bad_age, bad_inc, bad_score []int
	for i, r := range rows {
		if !valid_age(r.Age) {
			bad_age = append(bad_age, i)
		}
		if !valid_inc(r.AnnualIncome) {
```

```
                bad_inc = append(bad_inc, i)
            }
            if !valid_score(r.SpendingScore) {
                bad_score = append(bad_score, i)
            }
        }

        fmt.Println("\nCheck which rows have invalid numeric values:")
        fmt.Println("Invalid rows:")
        fmt.Println(" Age:           ", bad_age)
        fmt.Println(" Annual Income: ", bad_inc)
        fmt.Println(" Spending Score:", bad_score)
        fmt.Println("\nFinds the nearest valid neighbor with correct values.So each wrong value is replaced by the closest valid one:")

        // fix ages
        for _, idx := range bad_age {
            dist := dijkstra(adj, idx)
            best_j := -1
            best_d := math.Inf(1)
            for j, r := range rows {
                if j != idx && valid_age(r.Age) {
                    if dist[j] < best_d {
                        best_d = dist[j]
                        best_j = j
                    }
                }
            }
            if best_j != -1 {
                old := rows[idx].Age
                rows[idx].Age = rows[best_j].Age
                fmt.Printf("[Age Dijkstra] CustomerID=%s %d -> %d\n", rows[idx].CustomerID, old, rows[idx].Age)
            }
```

```
	}

	// fix incomes
	for _, idx := range bad_inc {
		dist := dijkstra(adj, idx)
		best_j := -1
		best_d := math.Inf(1)
		for j, r := range rows {
			if j != idx && valid_inc(r.AnnualIncome) {
				if dist[j] < best_d {
					best_d = dist[j]
					best_j = j
				}
			}
		}
		if best_j != -1 {
			old := rows[idx].AnnualIncome
			rows[idx].AnnualIncome = rows[best_j].AnnualIncome
			fmt.Printf("[Annual Income Dijkstra] CustomerID=%s %d -> %d\n",
rows[idx].CustomerID, old, rows[idx].AnnualIncome)
		}
	}

	// fix spending score
	for _, idx := range bad_score {
		dist := dijkstra(adj, idx)
		best_j := -1
		best_d := math.Inf(1)
		for j, r := range rows {
			if j != idx && valid_score(r.SpendingScore) {
				if dist[j] < best_d {
					best_d = dist[j]
					best_j = j
				}
```

```
                        }
                }
                if best_j != -1 {
                        old := rows[idx].SpendingScore
                        rows[idx].SpendingScore = rows[best_j].SpendingScore
                        fmt.Printf("[Spending Score Dijkstra] CustomerID=%s %d -> %d\n",
rows[idx].CustomerID, old, rows[idx].SpendingScore)
                }
        }
}


// ===== helpers for Go =====

type HeapItem struct {
        node int
        dist float64
}

type MinHeap []HeapItem

func (h MinHeap) Len() int           { return len(h) }
func (h MinHeap) Less(i, j int) bool { return h[i].dist < h[j].dist }
func (h MinHeap) Swap(i, j int)      { h[i], h[j] = h[j], h[i] }
func (h *MinHeap) Push(x interface{}) {
        *h = append(*h, x.(HeapItem))
}
func (h *MinHeap) Pop() interface{} {
        old := *h
        n := len(old)
        x := old[n-1]
        *h = old[:n-1]
        return x
}
```

```
func atoi(s string) int {
	var x int
	fmt.Sscanf(strings.TrimSpace(s), "%d", &x)
	return x
}

func min3(a, b, c int) int {
	if a < b {
		if a < c {
			return a
		}
		return c
	}
	if b < c {
		return b
	}
	return c
}

// main

func main() {
	rows := load_rows(INPUT_CSV)
	fmt.Printf("Loaded %d rows from CSV file.\n\n", len(rows))

	// BFS
	fmt.Println("Find Unique rows and fix with BFS in categorical data")
	fix_genres(rows)

	// graph for dijkstra
	adj := build_graph(rows)

	// Dijkstra fixes
```

```
        repair_numeric(rows, adj)

        save_rows(CLEANED_CSV, rows)
        fmt.Printf("Saved cleaned result to %s\n\n", CLEANED_CSV)

}
```