Data Structures and Algorithms

[SKY VOYAGE]

Course Project Report

School of Computer Science and Engineering

2023-24

Contents

1. Course and Team Details

1.1 Course details

Course Name	Data Structures and Algorithms
Course Code	23ECSC205
Semester	III
Division	B
Year	2023-24
Instructor	Dr. Priyanka Gawade

1.2 Team Details

Si.No.	Roll No.	Name
1.		DHANUSH KAMATAR (02FE22BCS034)
2.		MONISHYA KAMBLE (02FE22BCS054)
3.		SACHIN KIRAGI (02FE22BCS090)
4.	37	SANIKA UTTARKAR (02FE22BCS105)

1.3 Report Owner

Roll No.	Name
37	SANIKA UTTARKAR (02FE22BCS105)

2. Introduction

In a world increasingly interconnected by air travel, the need for an efficient and robust Flight Network Analysis System is paramount. Our project aims to address this imperative by developing a comprehensive system that caters to both user and managerial requirements within the dynamic landscape of a flight network. User convenience and data security are at the forefront of our design philosophy. The system boasts user-friendly features such as secure registration and login processes, facilitated by the Knuth-Morris-Pratt (KMP) string matching algorithm. Managers are equipped with tools to seamlessly manage flights, incorporating the creation and deletion of routes in the underlying graph data structure. Users, in turn, benefit from intuitive booking and cancellation functionalities. The system also encompasses advanced features, including path and cost analysis, alphabetical sorting of cities, and the ability to add reviews and ratings, ensuring a holistic and user-centric approach. Persistent storage in files guarantees data integrity, allowing for seamless retrieval and continuous improvement of the flight network analysis experience. As our world continues to traverse global airspace, this Flight Network Analysis System stands as an indispensable tool for efficient, secure, and user-friendly air travel management.

3. Problem Statement

3.1 Domain

Problem Statement: Sky Voyage - Flight Network Analysis System
Sky Voyage aims to revolutionize the management of flight networks through the development of a comprehensive Flight Network Analysis System. The system is designed to cater to both user and manager interactions, providing essential functionalities that encompass user registration and login, manager registration and login, graph creation for flight connections, flight management, booking and cancellation, path and cost analysis, alphabetical sorting of cities, and the ability to add reviews and ratings. In addition to its user-centric features, the system prioritizes data integrity by ensuring persistent storage of user, manager, and travel experience details in appropriate files.

Key Features:
1. Secure User and Manager Registration/Login.
2. Dynamic Graph Creation for Flight Connections.
3. Flight Management with Booking and Cancellation.
4. Path and Cost Analysis for User Convenience.
5. Alphabetical Sorting of Cities for Easy Reference.
6. User-Generated Reviews and Ratings.
7. Persistent Storage for User, Manager, and Travel Experience Details.

3.2 Module Description

1.KMP String Matching():
The provided C code defines functions for user registration and authentication using a file-based storage system. The program employs the Knuth-Morris-Pratt algorithm for efficient string matching during username existence checks. The registerUser function prompts the user for a unique username and a password, storing the credentials in a file. The loginUser function verifies entered credentials against stored records, facilitating user authentication. The code incorporates password input without display, enhancing security. Error handling for file operations is included. Overall, the module provides a basic user management system with file-based storage, employing string matching and password security measures.

2. assignPathToEachFlight():
This function iterates through all pairs of places, applying a depth-first search to find all possible paths between them and assigning those paths to respective flights.

3.sortTempFlightsFromSourceToDestByCost():
It sorts an array of temporary flights from source to destination based on their total cost, facilitating efficient selection of the optimal flight.

4.findFirstOccuOfReqFlight():
This function searches for the index of the first flight with a given source and destination, aiding in retrieving specific flight information.

5. searchForFlight():
The function searches for flights from a given source to destination, utilizing the index of the first occurrence of the required flight. It dynamically allocates memory for temporary flights, populates them, sorts them by cost, and displays the available flights, enhancing user experience in flight selection.

6. viewAllFlightsFromSourceToDestination():
The function prompts the user to input source and destination places, converts them to lowercase, and finds corresponding keys. It then checks for flight availability between the locations, displays the available flights, and provides an option to return to the main menu, enhancing user interaction in exploring flight options.

7. bookFlight():
The function facilitates user interaction for flight booking. It prompts users to input source and destination, converts them to lowercase, and finds corresponding keys. After displaying available flights, the user selects a flight by entering the corresponding id. The function handles seat availability, confirms user booking preferences, and updates seat availability and booking details. It provides feedback on successful bookings or prompts the user to try again. The process enhances user experience with a user-friendly interface, ensuring a smooth booking process and updating relevant flight details while maintaining user security.

8.cancel_booking()
It is used to cancel flight bookings for a user. It displays the user's booked flights, prompts for cancellation, and updates the user's file accordingly. It also checks for valid input and updates the available seats for the cancelled flight. The function uses file handling and basic console I/O for user interaction.

4. Functionality Selection

Si. No.	Functionality Name	Known	Unknown	Principles applicable	Algorithms	Data Structures
	Name the functionality within the module	What information do you already know about the module? What kind of data you already have? How much of process information is known?	What are the pain points? What information needs to be explored and understood? What are challenges?	What are the supporting principles and design techniques?	List all the algorithms you will use	What are the supporting data structures?
1	The KMP String Matching module performs pattern matching in a text using the Knuth-Morris-Pratt algorithm.	The module searches for occurrences of a pattern in a text and returns the index of the first occurrence. The data includes the text and pattern strings. Used for matching Login details and passwords. The algorithm iterates through the text, utilizing the Longest Prefix Suffix (LPS) array for efficient pattern matching.	Potential challenges include memory allocation errors and handling large text or pattern inputs. Understanding the efficiency trade-offs and memory usage during large-scale pattern matching. Memory management and handling edge cases where the pattern or text is extensive	Efficient pattern matching using the KMP algorithm for improved time complexity. Utilizing dynamic memory allocation for the LPS array and smart indexing for pattern matching efficiency.	Knuth-Morris-Pratt (KMP) algorithm for string matching.	Dynamic arrays for the LPS array, integer variables for indices, and character arrays for text and pattern strings.
2	Assigns paths to each flight between all pairs of places using a depth-first search.	Utilizes a depth-first search and involves places and flights. Iterates through pairs of places, applying a depth-first search to find	Potential issues may arise with complex path assignments and optimization. Managing multiple paths for each flight	Efficient path assignment using a depth-first search. Iterative depth-first search for path assignment.	Depth First Search algorithm.	Graph Data Structure.

		paths for each flight.	and optimizing the process.			
3	Sorts temporary flights from source to destination based on cost.	Involves temporary flights and sorting by cost. Iterative sorting algorithm based on total cost.	Potential issues with sorting efficiency and algorithmic complexity. Understanding the sorting algorithm's efficiency for large datasets. Efficiently handling a large number of temporary flights.	Sorting algorithm principles, possibly using a comparison-based sort. Bubble sort or similar sorting algorithms for simplicity.		
4.	Finds the index of the first occurrence of a flight with given source and destination .	Involves searching for flights with specified source and destination. Iterates through flights, searching for the first occurrence.	Challenges with optimizing search algorithms for large datasets. Evaluating the efficiency of search algorithms for various scenarios. Optimizing search for large flight datasets.	Efficient search algorithms, possibly linear search. Linear search for simplicity unless optimized search is necessary.		
5.	Searches and displays available flights between a given source and destination.	Utilizes source and destination data, checks flight availability, and displays relevant information. Iterates through flights, validating and displaying available options .	Managing user inputs, handling invalid choices, and efficient display. Optimizing display and user interaction. Efficiently handling large datasets of flights	User-friendly interface design, input validation. Linear search for available flights, interactive user prompts .		
6	Displays all available					

	flights between a given source and destination.	Utilizes source and destination data, checks and displays flight availability. Involves user input, source-destination conversion, and flight display.	User input validation, conversion handling, and display optimization. Efficient handling of user inputs and conversion. User-friendly handling of inputs, ensuring correct conversions.	User-friendly interface design, source-destination conversion. Interactive user prompts, efficient display.		
7.	Facilitates the user in booking a flight, including seat selection and confirmation.	Involves source-destination input, flight selection, and booking details. Iterative user prompts, seat availability check, and booking confirmation.	Seat availability management, user confirmation. Efficient seat booking process. Optimizing seat availability updates and user interaction.	User-friendly interface design, seat availability checks. Interactive user prompts, efficient seat availability updates.		Arrays (for booked flight validation), file structures for user and temporary files
8.	The module cancels flight bookings, displaying user bookings, updating files, and adjusting seat availability.	It reads and writes user files, manages flight data, and handles console I/O. The cancellation process involves user input validation, file manipulation, and seat availability updates.	Error handling, user input validation, and file manipulation robustness are potential challenges. Understanding file handling nuances, enhancing error handling, and validating user inputs thoroughly are crucial. Robust error handling, user input validation, and ensuring file integrity pose challenges.	File I/O, error handling, input validation, modular design. File manipulation, conditional statements, loops, modularizatio n	Sequential search for booked flights validation.	Arrays (for booked flight validation), file structures for user and temporary files.

5. Functionality Analysis

KMP Search():

- Implements the Knuth-Morris-Pratt pattern matching algorithm to find a pattern (pattern) in a text (text).
- Dynamically allocates memory for the Longest Prefix Suffix (LPS) array.
- Builds the LPS array and performs pattern matching.
- Frees the allocated memory and returns the index where the pattern is found or -1 if not found.

isUsernameExists:

- Checks if a given username already exists in a file ("user1_details.txt").
- Uses the kmpSearch function to perform pattern matching efficiently.
- Returns 1 if the username exists, 0 if it doesn't.

getPassword:

- Allows the user to input a password without displaying it on the screen.
- Uses getch to get each character, masking it with '*' on the screen.
- Handles backspace to erase characters.

registerUser:

- Prompts the user to enter a username and password.
- Checks if the username already exists using isUsernameExists.
- If not, takes a password using getPassword and appends the username-password pair to "user1_details.txt".
- Prints a success message and calls the normaluser function.

5. loginUser:

- Prompts the user to enter a username and password.
- Checks if the username exists using isUsernameExists.
- If yes, verifies the password using getPassword.
- If the password is correct, prints a login success message and calls the normaluser function.
- If the password is incorrect, prints an error message and returns to the usermenu.

Efficiency Analysis:

kmpSearch:

- Time Complexity: O(m + n) where m is the pattern length and n is the text length.
- Space Complexity: O(m) for the LPS array.

Overall, the efficiency is reasonable, especially considering that the dominant factor is the pattern matching step. The use of KMP enhances pattern matching efficiency.

assignPathToEachFlight():

The function iterates through all pairs of places using nested loops, excluding cases where the source and destination are the same. For each pair, it initializes a visited array, performs

depth-first search (DFS) to find all paths between the places, and updates flight information accordingly.

Efficiency Analysis:
The function's time complexity is influenced by the nested loops and the DFS operation. In the worst case, where all places are interconnected, the time complexity is O(N^3), where N is the number of places. The DFS operation itself has a time complexity of O(V + E), where V is the number of vertices and E is the number of edges in the graph. The overall efficiency depends on the density of connections between places.

sortTempFlightsFromSourceToDestByCost():
The function implements the Bubble Sort algorithm to sort an array of temporary flights from source to destination based on their total cost. It compares adjacent elements in the array and swaps them if they are in the wrong order. This process is repeated until the entire array is sorted.

Efficiency Analysis:
The time complexity of the Bubble Sort algorithm is O(n^2), where n is the number of elements in the array. In this case, the array size is 'noOfFlights.' While Bubble Sort is simple, its efficiency decreases for large datasets.

findFirstOccuOfReqFlight():
The function iterates through the array of flights to find the index of the first occurrence where both the source and destination match the provided parameters. It returns the index if a match is found; otherwise, it returns -1.

Efficiency Analysis:
The time complexity of this linear search algorithm is O(n), where n is the number of flights. In the worst case, the function iterates through the entire array to find the required flight.

searchForFlight():
1. The function calls `findFirstOccuOfReqFlight` to find the index of the first occurrence of a flight matching the given source and destination.
2. If no matching flight is found, it prints a message, clears the screen, and returns 0.
3. It dynamically allocates memory for an array `tempFlightsFromSourceToDest` to store temporary flights.
4. It iterates through the flights starting from the found index, collecting flights with the specified source and destination, and populates `tempFlightsFromSourceToDest`.
5. Calls `sortTempFlightsFromSourceToDestByCost` to sort the temporary flights based on their total cost.
6. Displays the sorted flights.
7. Returns 1.

Efficiency Analysis:
1. The `findFirstOccuOfReqFlight` has a linear search time complexity of O(n), where n is the number of flights.
2. Memory allocation for `tempFlightsFromSourceToDest` has a constant time complexity.
3. The loop populating `tempFlightsFromSourceToDest` has a time complexity of O(m), where m is the number of flights with the specified source and destination.
4.Sorting`tempFlightsFromSourceToDest`using `sortTempFlightsFromSourceToDestByCost` has a time complexity of O(m^2) in the worst case.
5. Displaying the sorted flights has a time complexity of O(m).

viewAllFlightsFromSourceToDestination():
1. Displays available places using `displayPlaces()`.
2. Prompts the user to enter the source and destination places.
3. Converts numeric IDs to place names if necessary.
4. Converts source and destination names to lowercase.
5. Finds keys for source and destination using `findKeyForString`.
6. Checks if the source and destination are valid, displaying an error message if not.
7. Calls `searchForFlight` to display available flights between the specified source and destination.
8. Waits for user input and clears the screen upon pressing any key.

Efficiency Analysis:
1. Displaying places using `displayPlaces()` has a constant time complexity.
2. User input and conversion have constant time complexities.
3. `findKeyForString` has a time complexity of O(n), where n is the number of places.
4. The overall efficiency depends on the number of places and the validity of user inputs. The operations are generally efficient for moderate-sized datasets, but user input handling may introduce variability.

bookFlight():
1. Displays available places using `displayPlaces()`.
2. Prompts the user to enter the source and destination places.
3. Converts numeric IDs to place names if necessary.
4. Converts source and destination names to lowercase.
5. Finds keys for source and destination using `findKeyForString`.
6. Checks if the source and destination are valid, displaying an error message if not.
7. Calls `searchForFlight` to display available flights between the specified source and destination.
8. Prompts the user to enter the flight ID or -1 to cancel.
9. Validates the entered flight ID and prompts the user again if invalid.
10. Displays details of the chosen flight.
11. Prompts the user to enter the number of seats to be booked.
12. Checks seat availability and prompts the user to confirm the booking.

13. Updates seat availability, displays booking details, and writes them to a file.
14. Displays a thank you message and clears the screen.

Efficiency Analysis:
1. Displaying places using `displayPlaces()` has a constant time complexity.
2. User input and conversion have constant time complexities.
3. `findKeyForString` has a time complexity of O(n), where n is the number of places.
4. The efficiency depends on user input validity and the number of available flights, affecting the performance of `searchForFlight`.
5. Booking operations involve constant time complexities, but repeated invalid inputs may lead to inefficiencies.
6. Writing booking details to a file has a constant time complexity.
7. Overall efficiency depends on the number of available flights and user input handling. Optimization can be achieved by improving input validation and potentially using more efficient data structures for large datasets.

cancelFlight():
1. The function starts by calling the `display_booked_flights` function to show the user's booked flights. If there are no booked flights, it waits for 3 seconds and returns.
2. If there are booked flights, the user is prompted to enter a flight ID to cancel the booking or enter -1 to go back.
3. The function then opens the user's file for reading and writing, and a temporary file is created for writing.
4. It reads the user's file line by line, parsing the flight details and checking if the flight ID matches the one the user wants to cancel. If it matches, the line is skipped to effectively remove the booking.
5. The non-matching lines are written to the temporary file.
6. The original user's file is closed, and the temporary file is closed and replaces the original file.
7. The function checks if the entered flight ID was actually booked by the user. If not, it prompts the user to enter a valid ID.
8. If the entered flight ID is valid, it updates the available seats for the corresponding flight (if the flight ID is within the range of the available flights).
9. A success message is displayed, and the screen is cleared.

Efficiency Analysis:
1. The function uses a simple file-based approach to store user booking information, which is reasonable for small-scale applications.
2. It iterates through the user's file line by line, which may not be efficient for large files, but considering the typical size of user booking files, this approach is generally acceptable.
3. The use of a temporary file for writing and then replacing the original file helps in maintaining data integrity.
4. The function uses an array (`flightsBooked`) to keep track of flight IDs that the user has booked, which aids in checking the validity of the entered flight ID.

5. The function uses a goto statement (`goto flag`) to handle input validation and looping back to the input prompt, which might be considered less readable, but it serves the purpose in this case.
6. The function clears the screen using `system("cls")`, which is a platform-dependent command and may not work on all systems.
7. The function lacks proper error handling in case of file operations or invalid input, and it uses `perror` to print error messages, which may not provide the best user experience.
8. The function uses the `Sleep` and `sleep` functions for waiting, which might be platform-dependent and could be replaced with more portable alternatives.

6. Conclusion

This project highlights the importance of efficient algorithms and data structures in handling flight information and user interactions. Learning includes the implementation of search and sorting algorithms, user-friendly interfaces, and error handling. Key takeaways involve enhancing code efficiency, managing user inputs, and improving overall system responsiveness.

~*~*~*~*~*~*~*~