

Course: Algorithms Report

Topic

Journey Economizer, fly high spend low

Table of Contents:

| SL NO | Topics | Page NO. |
|--------------|--------------------------|-----------------|
| 01 | Introduction | 03 |
| 02 | Objective | 03 |
| 03 | Technology specification | 03 |
| 04 | Data flow diagram | 03 |
| 05 | Features List | 06 |
| 06 | Implementation | 17 |
| 07 | Complexity Analysis | 30 |
| 08 | Results | 31 |
| 09 | Limitations | 31 |
| 10 | Conclusion | 31 |

Project Report: Journey Economizer

1. Introduction

The Journey Economizer program is designed to help users plan trips efficiently and manage their finances effectively. By using advanced algorithms like Dijkstra's Algorithm and the Knapsack Algorithm it identifies the cheapest travel routes and provides practical financial advice. This program makes travel planning and financial management simple, efficient, and accessible to users. It addresses the challenges of high travel costs and budget constraints, offering solutions that are both practical and user-friendly.

2. Objective

The primary objective of the Journey Economizer program is to assist users in saving money by planning cost-effective and efficient travel routes and also focuses on financial management, providing practical solutions for reducing expenses and ensuring a smooth and budget-friendly journey. The program focuses on solving the challenges of rising travel expenses and inefficient budgeting practices. It is tailored for travelers and individuals who want to manage their finances better. By delivering actionable insights, it helps users make informed decisions, improve financial planning, and ensure cost-effective travel experiences.

3. Technology Specification

Programming Language: C++

Development Framework: Standard Template Library (STL)

Tools: g++ (GNU Compiler Collection), Visual Studio Code, Code Block

Libraries: iostream, fstream, vector, map, list, queue, unordered_map

Hardware Requirements:

CPU: Dual-core processor

RAM: Minimum 4 GB

Storage: Minimum 100 MB free space

Security Considerations: Input validation to prevent injection attacks.

Scalability: Designed to handle multiple users and large datasets..

4. Data Flow Diagram

The program processes user inputs such as travel preferences, budgets, and destinations. These inputs go through the following stages:

1. Input Validation: Ensures the data entered by users is correct and valid.

2. Algorithm Execution: Uses Dijkstra's algorithm for finding the shortest routes and the Knapsack algorithm for financial planning.
3. Output Display: Presents optimized travel routes and financial plans in a clear, userfriendly manner.

Journey Economizer and Financial Advisor



5. Features List

Travel Cost Optimization: Identifies the minimum cost of routes for travel.

Algorithm: Dijkstra

Input:

- Source vertex (src)
- Adjacency list of the graph (adj) where each entry contains a list of pairs (cost, neighbor)
- Distance map (dist) initialized to infinity for all vertices except the source
- Previous vertex map (prev) initialized to empty for all vertices

Output:

- Updated distance map (dist) with the shortest distances from the source to all vertices
- Updated previous vertex map (prev) to reconstruct the shortest path

1. Initialize:

- Create a priority queue (pq) to store pairs of (cost, vertex).
- Set **dist[src] = 0** (the cost from the source to itself is 0).
- Push the pair (0, src) into the priority queue.

2. While the priority queue is not empty: a. Extract the vertex with the smallest cost:

- Let **u** be the vertex corresponding to the smallest cost in the priority queue.
- Remove **u** from the priority queue.

b. For each neighbor of u:

- Let **v** be the neighbor vertex and **cost** be the travel cost from **u** to **v**.

i. Relaxation step: - If **dist[u] + cost < dist[v]**:

ii. Update the distance: **dist[v] = dist[u] + cost**.

iii. Update the previous vertex: **prev[v] = u**.

iv. Push the updated pair (dist[v], v) into the priority queue.

3. End of algorithm:

- The distance map (dist) now contains the shortest distances from the source to all vertices.
- The previous vertex map (prev) can be used to reconstruct the shortest paths

Output:

```
Enter the source country: Argentina
Enter your budget: 5000
Paths within budget of 5000:
Cost from Argentina to Spain = 1700 Path: Argentina -> France -> Spain
Cost from Argentina to Germany = 1100 Path: Argentina -> France -> Germany
Cost from Argentina to Norway = 3500 Path: Argentina -> Canada -> England -> Norway
Cost from Argentina to Bangladesh = 1100 Path: Argentina -> Canada -> Bangladesh
Cost from Argentina to Sweden = 2300 Path: Argentina -> France -> Germany -> Denmark -> Sweden
Cost from Argentina to France = 700 Path: Argentina -> France
Cost from Argentina to Italy = 1200 Path: Argentina -> France -> Italy
Cost from Argentina to Denmark = 1400 Path: Argentina -> France -> Germany -> Denmark
Cost from Argentina to Portugal = 2400 Path: Argentina -> France -> Spain -> Portugal
Cost from Argentina to Canada = 900 Path: Argentina -> Canada
Cost from Argentina to England = 2000 Path: Argentina -> Canada -> England
Enter Destination:
```

Complexity Analysis:

Time Complexity Analysis:

Using an Adjacency List with a Min-Heap (Priority Queue)

- **Initialization:** $O(V)$ for initializing the distance array and priority queue.
- **Main Loop:** The algorithm processes each vertex once. Each vertex is extracted from the priority queue, which takes $O(\log V)$ time. For each vertex, we relax its edges:
 - If a vertex has E edges, the relaxation step takes $O(E)$ time in total.
- **Total Complexity:**
 - The extraction of vertices from the priority queue contributes $O(V \log V)$.
 - The relaxation of edges contributes $O(E \log V)$ because each edge relaxation may involve a priority queue update.
 - Thus, the total time complexity is: $[O((V + E) \log V)]$

Space Complexity Analysis:

The space complexity of Dijkstra's algorithm is determined by the storage requirements for the graph representation and the auxiliary data structures:

Graph Representation: Adjacency List: $O(V + E)$ for storing the list of edges.

Auxiliary Structures:

- The distance array requires $O(V)$ space.
- The previous vertex array requires $O(V)$ space.

- The priority queue requires $O(V)$ space in the worst case.

Total Space Complexity: $O(V + E)$

Financial Planning: Offers savings plans and budget optimization.

Part-1:

Here's the algorithm that calculates financial planning, including income, expenses, savings, and goal projections.

Algorithm:

*Input: Monthly Income (I), Fixed Expenses (F), Variable Expenses (V), Financial Goals (G),
Time Period (T in months), Savings Interest Rate (r)*

1. Initialize Savings (S) = 0
2. For each month from 1 to T :
 - a. Calculate Total Expenses (E) = $F + V$
 - b. Calculate Monthly Savings (M) = $I - E$
 - c. If $M > 0$:
 - i. Update Savings: $S = S + M + (S * r / 12)$
 - d. For each Goal g in G :
 - i. Check if Savings $\geq g.amount$
 - ii. If true, allocate $g.amount$ from S
 - e. Record monthly savings and goals progress
3. Return final Savings (S) and Goal Progress (G)

Output:

| ----- Spending Breakdown ----- | |
|--|--------------|
| Category | Amount (\$) |
| ----- | |
| Monthly Income | 1000.00 |
| Fixed Costs | 400.00 |
| bills | 90.00 |
| food | 200.00 |
| medicine | 180.00 |
| ----- | |
| Total Spending | 870.00 |
| Savings | 130.00 |
| ----- | |
| ----- Projected Cash Flow ----- | |
| Month | Savings (\$) |
| ----- | |
| 1 | 130.00 |
| 2 | 132.47 |
| 3 | 134.94 |
| 4 | 137.42 |
| 5 | 139.90 |
| 6 | 142.39 |
| ----- | |
| ----- Maximum Achievable Reduction ----- | |
| Vacation Goal: \$1100.00 in 6 months. | |
| Required Monthly Savings: \$183.33 | |
| Current Monthly Savings: \$130.00 | |
| Additional Reduction Needed: \$53.33 | |

Complexity Analysis

Time Complexity Analysis

1. Expense Calculation: $O(1)$ (Simple arithmetic).
2. Savings Update: $O(1)$ (Adding and interest calculation).
3. Goal Management $O(|G|)$, where $|G|$ is the number of financial goals.
4. Monthly Iteration: Runs T iterations for T months.

Total Time Complexity:

$O(T \cdot (1 + |G|)) = O(T \cdot |G|)$ where T is the time period in months and $|G|$ is the number of goals.

Space Complexity Analysis

1. Savings Tracking: $O(1)$ for cumulative savings.
2. Goals Progress: $O(|G|)$ to store progress for each goal.
3. Monthly Records (Optional): $O(T)$, if detailed tracking per month is required.

Total Space Complexity:

- Without detailed tracking: $O(|G|)$.
- With detailed tracking: $O(T + |G|)$.

Part 2:

Cost Reduction Algorithm

Algorithm

Input: Initial Fixed Costs ($F_{initial}$), Initial Variable Costs ($V_{initial}$), Monthly Costs (

1. Initialize Total Cost Reduction = 0

2. For each month i from 1 to T :

a. Calculate Monthly Fixed Cost Reduction ($F_{reduction}$) = $F_{initial} - F[i]$

b. Calculate Monthly Variable Cost Reduction ($V_{reduction}$) = $V_{initial} - V[i]$

c. Calculate Total Monthly Reduction (R) = $F_{reduction} + V_{reduction}$

d. Accumulate Total Reduction: Total Cost Reduction += R

3. Calculate Percentage Reduction:

a. Initial Total Cost = $F_{initial} + V_{initial}$

b. Final Total Cost = $F[T] + V[T]$

c. Percentage Reduction

= $((Initial\ Total\ Cost - Final\ Total\ Cost) / Initial\ Total\ Cost) * 100$

4. Return Total Cost Reduction and Percentage Reduction

Output:

```
Maximum Achievable Reduction: $53.33
The goal can be met by reducing the following costs proportionally:
- food: Reduce by $22.70 (42.6% of the total reduction)
- medicine: Reduce by $20.43 (38.3% of the total reduction)
- bills: Reduce by $10.21 (19.1% of the total reduction)
```

Complexity Analysis

Time Complexity

1. **Monthly Reductions Calculation:** $O(1)$ per month for fixed and variable costs.
2. **Iteration:** Runs T iterations for T months.

Total Time Complexity:

$O(T)$, where T is the number of months.

Space Complexity

1. **Cost Data Storage:** $O(T)$ for monthly costs if stored.
2. **Tracking Variables:** $O(1)$ for reductions and percentages.

Total Space Complexity:

- **Without storing monthly details:** $O(1)$.
- **With monthly details:** $O(T)$.

User-Friendly Interface: Provides easy-to-use menu navigation.

The code implements a journey cost calculator that allows users to find the cost of traveling from a source country to a destination country using Dijkstra's algorithm. It also allows users to find all reachable countries within a specified budget. The main components of the algorithm are as follows:

1. **User Input:**

- The user is prompted to choose between two options:
 1. Find the cost from a source country to a destination country.
 2. Find all countries reachable from a source country within a specified budget.

2. **Option 1: Find Cost from Source to Destination:**

- The user inputs the source and destination countries.
- The program calculates the trip cost using Dijkstra's algorithm.
- If a path exists, it prompts the user for financial details and saves the financial advice to a file.

3. Option 2: Find All Destinations Within Budget:

- The user inputs the source country and budget.
- Dijkstra's algorithm is run to find the shortest paths from the source to all other countries.
- The program prints all countries reachable within the budget and their respective costs.
- The user is then prompted for a destination country, and the trip cost is calculated again.

4. Financial Advisor:

- The program collects financial details from the user and uses a **FinancialAdvisor** class to visualize spending, project cash flow, and calculate maximum achievable reductions.

5. File Handling:

- The financial advice is saved to a text file, and the user is given the option to open the file.

Output:

```
*****-> JOURNEY ECONOMIZER <-*****
1.Find source to a destination country cost
2.Find source to all destination country within a budget
Enter your Choice:

```

Giving the choice: 1

```
Enter the source country for trip planning: Bangladesh
Enter the destination country for trip planning: Spain|
```

```

Shortest path from Bangladesh to Spain is: Bangladesh -> Canada -> France -> Spain
Minimum cost is: $2200
Enter your monthly income: 10000
Enter annual income growth rate (e.g., 0.05 for 5%): 0.03
Enter your fixed costs (e.g., rent, utilities): 8000
Enter the number of variable expense categories: 2
Enter each category name and monthly cost:
Snacks
500
Bill
1000
Enter the number of months for planning: 3

Financial advice saved to 'financial_advice.txt'.
Would you like to open the file now? (yes/no): yes

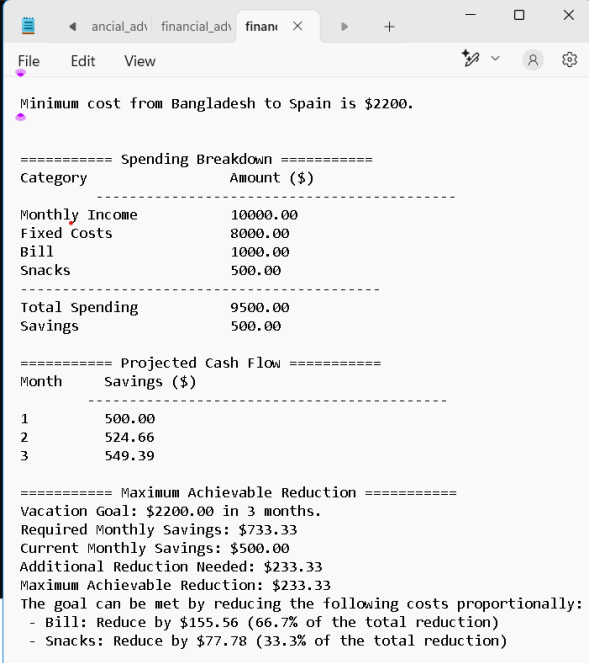
```

```

*****
*****-> THANK YOU :) *****
*****

Process returned 0 (0x0)   execution time : 240.907 s
Press any key to continue.

```



Minimum cost from Bangladesh to Spain is \$2200.

===== Spending Breakdown =====

| Category | Amount (\$) |
|----------------|-------------|
| Monthly Income | 10000.00 |
| Fixed Costs | 8000.00 |
| Bill | 1000.00 |
| Snacks | 500.00 |
| Total Spending | 9500.00 |
| Savings | 500.00 |

===== Projected Cash Flow =====

| Month | Savings (\$) |
|-------|--------------|
| 1 | 500.00 |
| 2 | 524.66 |
| 3 | 549.39 |

===== Maximum Achievable Reduction =====

Vacation Goal: \$2200.00 in 3 months.
 Required Monthly Savings: \$733.33
 Current Monthly Savings: \$500.00
 Additional Reduction Needed: \$233.33
 Maximum Achievable Reduction: \$233.33

The goal can be met by reducing the following costs proportionally:

- Bill: Reduce by \$155.56 (66.7% of the total reduction)
- Snacks: Reduce by \$77.78 (33.3% of the total reduction)

Giving the choice:2

```

Enter the source country: Bangladesh
Enter your budget: 5000
Paths within budget of 5000:
Cost from Bangladesh to Spain = 2200 Path: Bangladesh -> Canada -> France -> Spain
Cost from Bangladesh to Germany = 1100 Path: Bangladesh -> Denmark -> Germany
Cost from Bangladesh to Norway = 2800 Path: Bangladesh -> Canada -> England -> Norway
Cost from Bangladesh to Sweden = 1700 Path: Bangladesh -> Denmark -> Sweden
Cost from Bangladesh to France = 1200 Path: Bangladesh -> Canada -> France
Cost from Bangladesh to Italy = 1700 Path: Bangladesh -> Canada -> France -> Italy
Cost from Bangladesh to Argentina = 1100 Path: Bangladesh -> Canada -> Argentina
Cost from Bangladesh to Denmark = 800 Path: Bangladesh -> Denmark
Cost from Bangladesh to Portugal = 2900 Path: Bangladesh -> Canada -> France -> Spain -> Portugal
Cost from Bangladesh to Canada = 200 Path: Bangladesh -> Canada
Cost from Bangladesh to England = 1300 Path: Bangladesh -> Canada -> England
Enter Destination:Spain

```

```

Shortest path from Bangladesh to Spain is: Bangladesh -> Canada -> France -> Spain
Minimum cost is: $2200
Enter your monthly income: 10000
Enter annual income growth rate (e.g., 0.05 for 5%): 0.05
Enter your fixed costs (e.g., rent, utilities): 9000
Enter the number of variable expense categories: 1
Enter each category name and monthly cost:
med
800
Enter the number of months for planning: 6

Financial advice saved to 'financial_advice.txt'.
Would you like to open the file now? (yes/no): yes

```

The screenshot shows a C++ IDE with two windows. The left window displays the program's execution output, including a 'THANK YOU :)' message and execution time. The right window shows a report titled 'financial_advice.txt' with the following content:

```

Minimum cost from Bangladesh to Spain is $2200.

===== Spending Breakdown =====
Category          Amount ($)
-----
Monthly Income    10000.00
Fixed Costs       9000.00
med              800.00
-----
Total Spending    9800.00
Savings           200.00

===== Projected Cash Flow =====
Month    Savings ($)
-----
1         200.00
2         240.74
3         281.65
4         322.72
5         363.96
6         405.37

===== Maximum Achievable Reduction =====
Vacation Goal: $2200.00 in 6 months.
Required Monthly Savings: $366.67
Current Monthly Savings: $200.00
Additional Reduction Needed: $166.67
Maximum Achievable Reduction: $166.67
The goal can be met by reducing the following costs proportionally:
- med: Reduce by $166.67 (100.0% of the total reduction)

```

Complexity Analysis

Time Complexity Analysis

1. Dijkstra's Algorithm:

- **Initialization:** $O(V)$ for initializing the distance and previous vertex maps.
- **Main Loop:** The algorithm processes each vertex once. For each vertex, it examines its neighbors:
 - Using an adjacency list with a priority queue (min-heap):
 - Extracting the minimum distance vertex takes $O(\log V)$.
 - Relaxing edges takes $O(E)$ in total.

- **Total Time Complexity for Dijkstra's:** $[O((V + E) \log V)]$
2. **User Input and File Operations:**
 - Collecting user input is $O(1)$ for each input prompt.
 - Writing to a file is $O(N)$ where N is the number of lines written, but this is generally considered negligible compared to the Dijkstra's algorithm complexity.
 3. **Overall Time Complexity:**
 - The overall time complexity for the entire program is dominated by the Dijkstra's algorithm calls: $[O((V + E) \log V)]$

Space Complexity Analysis

1. **Graph Representation:**
 - Using an adjacency list: $O(V + E)$ for storing the graph.
2. **Auxiliary Structures:**
 - Distance map: $O(V)$
 - Previous vertex map: $O(V)$
 - Priority queue: $O(V)$ in the worst case.
3. **Financial Advisor Data:**
 - The space used by the **FinancialAdvisor** class depends on its implementation, but it generally requires $O(1)$ additional space for calculations.
4. **Overall Space Complexity:**
 - The overall space complexity is: $[O(V + E)]$

Summary

- **Time Complexity:**
 - Dijkstra's algorithm: $[O((V + E) \log V)]$
 - Overall: $[O((V + E) \log V)]$
- **Space Complexity:**
 - Overall: $[O(V + E)]$

Report Generation: Creates detailed financial and travel reports

Input: File containing Monthly Income (I), Fixed Expenses (F), Variable Expenses (V), Financial Goals (G), Savings Interest Rate (r)

Algorithm:

1. Open the file and read its contents
2. Parse the file to extract:
 - a. Monthly Income (I) for each month
 - b. Fixed Expenses (F) and Variable Expenses (V) for each month
 - c. Financial Goals (G) as a list of {goal_name, goal_amount}
 - d. Annual Savings Interest Rate (r)
3. Initialize Savings (S) = 0
4. For each month m in the data:
 - a. Calculate Total Monthly Expenses (E) = $F[m] + V[m]$
 - b. Calculate Monthly Savings (M) = $I[m] - E$
 - c. If $M > 0$:
 - i. Update Savings: $S = S + M + (S * r / 12)$
 - d. For each goal g in G:
 - i. If Savings (S) \geq g.goal_amount:
 - Allocate g.goal_amount from S
 - Mark g as "Achieved"
5. Record the following:
 - a. Monthly savings for each month
 - b. Status of each goal (achieved or pending)
6. Output:

- a. Final Savings (S)
- b. List of goals with their status (achieved/pending)

Output:

Minimum cost from Bangladesh to Spain is \$2200.

===== Spending Breakdown =====

| Category | Amount (\$) |
|----------------|-------------|
| Monthly Income | 10000.00 |
| Fixed Costs | 9000.00 |
| med | 800.00 |
| Total Spending | 9800.00 |
| Savings | 200.00 |

===== Projected Cash Flow =====

| Month | Savings (\$) |
|-------|--------------|
| 1 | 200.00 |
| 2 | 240.74 |
| 3 | 281.65 |
| 4 | 322.72 |
| 5 | 363.96 |
| 6 | 405.37 |

===== Maximum Achievable Reduction =====

Vacation Goal: \$2200.00 in 6 months.

Required Monthly Savings: \$366.67

Current Monthly Savings: \$200.00

Additional Reduction Needed: \$166.67

Maximum Achievable Reduction: \$166.67

The goal can be met by reducing the following costs proportionally:

- med: Reduce by \$166.67 (100.0% of the total reduction)

6. Implementation

Feature: Travel Cost Optimization

Code :

```
// Function to perform Dijkstra's algorithm

void dijkstra(const string& src, const unordered_map<string, list<ipair>>& adj,
unordered_map<string, int>& dist, unordered_map<string, string>& prev) {
priority_queue<ipair, vector<ipair>, greater<ipair>> pq; // Min-heap priority queue
dist[src] = 0; // Cost from source to itself is 0

pq.push({0, src}); // Push the source country into the priority queue

while (!pq.empty()) {    string u = pq.top().second; // Get the country with
the smallest cost    pq.pop();

    // Explore all neighbors of u    for (const auto&
neighbor : adj.at(u)) {        string v = neighbor.second; //
Neighbor country        int cost = neighbor.first; // Travel
cost

        // Relaxation step        if (dist[u] + cost <
dist[v]) {            dist[v] = dist[u] + cost; // Update
cost            prev[v] = u; // Update previous country
pq.push({dist[v], v}); // Push the updated cost into
the priority queue
        }
    }
}
}
```

```
// Function to print the shortest path from source to destination
void printPath(const string& target, const unordered_map<string, string>& prev) {
    if (prev.find(target) == prev.end() || prev.at(target) == "") {
        cout << target << " ";
        return;
    }
    printPath(prev.at(target), prev); // Recursive call to print the path
    cout << " -> " << target << " "; // Print the current country
}

```

Feature: Financial planning

Code:

Part one ->

```
void projectCashFlow(ofstream& output) {
    output << "\n===== Projected Cash Flow =====\n";
    output << setw(10) << "Month" << setw(20) << "Savings ($)\n";
    output << "----- \n";

    for (int t = 1; t <= planning_months; ++t) {
        double income = monthly_income * pow(1 + income_growth_rate, (t - 1) / 12.0);
        double total_variable_cost = 0;

        for (auto& cost : variable_costs) {
            total_variable_cost += cost.second;
        }

        double monthly_savings = income - (fixed_costs + total_variable_cost);
        output << setw(10) << t << setw(20) << fixed << setprecision(2) << monthly_savings << "\n";
    }
}

```

```
}
```

Part two ->

```
void calculateMaximumAchievableReduction(ofstream& output) {  
    // Convert variable costs to a vector for easier processing  
    vector<pair<string, double>> costs(variable_costs.begin(), variable_costs.end());  
  
    // Calculate required monthly savings  
    double monthly_vacation_savings = vacation_budget / planning_months;  
    double monthly_savings = monthly_income - fixed_costs -  
        accumulate(costs.begin(), costs.end(), 0.0,  
            [](double sum, const pair<string, double>& cost) { return sum + cost.second;  
        });  
  
    output << "\n===== Maximum Achievable Reduction =====\n";  
    output << "Vacation Goal: $" << fixed << setprecision(2) << vacation_budget << " in " <<  
    planning_months << " months.\n";  
    output << "Required Monthly Savings: $" << fixed << setprecision(2) << monthly_vacation_savings  
    << "\n";  
    output << "Current Monthly Savings: $" << fixed << setprecision(2) << monthly_savings << "\n";  
  
    if (monthly_savings >= monthly_vacation_savings) {  
        output << "Congratulations! Your current savings are sufficient to meet your vacation goal.\n";  
        return;  
    }  
  
    double required_reduction = monthly_vacation_savings - monthly_savings;  
    output << "Additional Reduction Needed: $" << fixed << setprecision(2) << required_reduction <<  
    "\n";
```

```

// Sort costs by their amount (descending order of cost for max reduction)
sort(costs.begin(), costs.end(), [](const pair<string, double>& a, const pair<string, double>& b) {
    return a.second > b.second;
});

// Apply proportional reduction to each cost
double total_cost = accumulate(costs.begin(), costs.end(), 0.0,
    [](double sum, const pair<string, double>& cost) { return sum + cost.second; });
vector<pair<string, double>> selected_reductions;

// Calculate proportional reduction for each category
for (auto& cost : costs) {
    double proportion = cost.second / total_cost; // Proportion of total cost
    double reduction = proportion * required_reduction; // Proportional reduction
    selected_reductions.push_back({cost.first, reduction});
}

// Calculate total reduction
double max_reduction = accumulate(selected_reductions.begin(), selected_reductions.end(), 0.0,
    [](double sum, const pair<string, double>& reduction) { return sum +
reduction.second; });

output << "Maximum Achievable Reduction: $" << fixed << setprecision(2) << max_reduction <<
"\n";

if (max_reduction < required_reduction) {

```

```

    output << "Warning: Maximum achievable reduction is insufficient to meet your goal.\n";

    output << "Consider reducing fixed costs or increasing income.\n";

    return;
}

// Print selected reductions and their proportional values

output << "The goal can be met by reducing the following costs proportionally:\n";

for (const auto& reduction : selected_reductions) {

    double percentage = (reduction.second / cost.second) * 100; // Percentage reduction for the
category

    output << " - " << reduction.first << ": Reduce by $"

        << fixed << setprecision(2) << reduction.second << " (" << fixed << setprecision(1)

        << percentage << "% of the original cost)\n";

}

}

```

Feature: User-Friendly Interface

Code:

```

cout<<"\n\n*****-> JOURNEY ECONOMIZER <-
*****\n"<<endl;

cout<<"1.Find source to a destination country cost\n";

cout<<"2.Find source to all destination country within a budget\n";

cout<<"Enter your Choice:\n";

cin>>c;

clearScreen();

if(c==1){

// Prompt user for trip planning details

```

```

string source, destination;

cout << "Enter the source country for trip planning: ";

cin >> source;

cout << "Enter the destination country for trip planning: ";

cin >> destination;

clearScreen();

    // Open file for saving output
string filename = "financial_advice.txt";
ofstream output(filename);

if (!output.is_open()) {
    cerr << "Error: Unable to open file for saving output.\n";
    return 1;
}

// Calculate trip cost using Dijkstra's algorithm
double vacation_budget = 0;

FinancialAdvisor advisor(0, 0, 0, {}, vacation_budget, 0);

int trip_cost = advisor.calculateTripCost(source, destination, adj, output, dist, prev);

if (trip_cost == -1) {
    cout << "No path exists between the specified countries.\n";
    cout << "Enter a minimum amount you want to save for your trip: ";
    cin >> vacation_budget;
} else {
    vacation_budget = trip_cost;
}

```

```

// Get financial details from the user
double income, growth_rate, fixed_costs;
map<string, double> variable_costs;
int months;

getUserInputs(income, growth_rate, fixed_costs, variable_costs, vacation_budget, months);


// Update advisor with the trip cost as the vacation budget
FinancialAdvisor updated_advisor(income, growth_rate, fixed_costs, variable_costs,
vacation_budget, months);


// Display financial advice
updated_advisor.visualizeSpending(output);
updated_advisor.projectCashFlow(output);
updated_advisor.calculateMaximumAchievableReduction(output);


output.close();


// Ask the user if they want to open the file
cout << "\nFinancial advice saved to " << filename << ".\n";
string userChoice;
while (true) {
    cout << "Would you like to open the file now? (yes/no): ";
    cin >> userChoice;

    // Convert user input to lowercase for consistent comparison
    transform(userChoice.begin(), userChoice.end(), userChoice.begin(), ::tolower);

```



```

if (userChoice == "yes" || userChoice == "y") {
    #if defined(_WIN32) || defined(_WIN64)
        system(("start " + filename).c_str());
    #elif defined(__APPLE__)
        system(("open " + filename).c_str());
    #else
        system(("xdg-open " + filename).c_str());
    #endif
    break;
} else if (userChoice == "no" || userChoice == "n") {
    cout << "Okay, not opening the file.\n";
    break;
} else {
    cout << "Invalid input. Please enter 'yes', 'y', 'no', or 'n'. \n";
}
}
}

else if(c==2)
{
    // User input for source country and budget
    string source,destination;
    double budget;
    cout << "Enter the source country: ";
    cin >> source;

```

```

cout << "Enter your budget: ";
cin >> budget;

// Run Dijkstra's algorithm from the user-defined source
dijkstra(source, adj, dist, prev);

// Print the shortest path and cost from the source to each country within the budget
cout << "Paths within budget of " << budget << ":\n";
for (const auto& country : adj) {
    if (dist[country.first] && dist[country.first] <= budget) {
        cout << "Cost from " << source << " to " << country.first << " = " << dist[country.first] << " ";
        cout << "Path: ";
        printPath(country.first, prev);
        cout << endl;
    }
}
cout << "Enter Destination:";
cin >> destination;
clearScreen();
string filename = "financial_advice.txt";
ofstream output(filename);
if (!output.is_open()) {
    cerr << "Error: Unable to open file for saving output.\n";
    return 1;
}

```

```

// Calculate trip cost using Dijkstra's algorithm
double vacation_budget = 0;
FinancialAdvisor advisor(0, 0, 0, {}, vacation_budget, 0);

int trip_cost=advisor.calculateTripCost(source, destination, adj, output,dist,prev);
if (trip_cost == -1) {
    cout << "No path exists between the specified countries.\n";
    cout << "Enter a minimum amount you want to save for your trip: ";
    cin >> budget;
}
else{
    budget = trip_cost;
}

// Get financial details from the user
double income, growth_rate, fixed_costs;
map<string, double> variable_costs;
int months;

getUserInputs(income, growth_rate, fixed_costs, variable_costs, budget, months);

// Update advisor with the trip cost as the vacation budget
FinancialAdvisor updated_advisor(income, growth_rate, fixed_costs, variable_costs, budget,
months);

// Display financial advice

```

```

updated_advisor.visualizeSpending(output);
updated_advisor.projectCashFlow(output);
updated_advisor.calculateMaximumAchievableReduction(output);

output.close();

// Ask the user if they want to open the file
cout << "\nFinancial advice saved to " << filename << ".\n";
string userChoice;
while (true) {
    cout << "Would you like to open the file now? (yes/no): ";
    cin >> userChoice;

    // Convert user input to lowercase for consistent comparison
    transform(userChoice.begin(), userChoice.end(), userChoice.begin(), ::tolower);

    if (userChoice == "yes" || userChoice == "y") {
        #if defined(_WIN32) || defined(_WIN64)
            system(("start " + filename).c_str());
        #elif defined(__APPLE__)
            system(("open " + filename).c_str());
        #else
            system(("xdg-open " + filename).c_str());
        #endif
        break;
    } else if (userChoice == "no" || userChoice == "n") {

```

```

        cout << "Okay, not opening the file.\n";

        break;

    } else {

        cout << "Invalid input. Please enter 'yes', 'y', 'no', or 'n'. \n";

    }

}

}

else

{

    cout<<"Kindly choose 1 or 2\n";

    goto i;

}

clearScreen();

cout<<"\n\n*****\n*****-> THANK
YOU :) *****\n*****\n\n\n";

return 0;

}

```

Feature: Report Generation

Code:

```

cout << "\nFinancial advice saved to '" << filename << "'.\n";

string userChoice;

while (true) {

    cout << "Would you like to open the file now? (yes/no): ";

    cin >> userChoice;

    // Convert user input to lowercase for consistent comparison

```

```

transform(userChoice.begin(), userChoice.end(), userChoice.begin(), ::tolower);

if (userChoice == "yes" || userChoice == "y") {
    #if defined(_WIN32) || defined(_WIN64)
        system(("start " + filename).c_str());
    #elif defined(__APPLE__)
        system(("open " + filename).c_str());
    #else
        system(("xdg-open " + filename).c_str());
    #endif
    break;
} else if (userChoice == "no" || userChoice == "n") {
    cout << "Okay, not opening the file.\n";
    break;
} else {
    cout << "Invalid input. Please enter 'yes', 'y', 'no', or 'n'. \n";
}
}
}
else
{
    cout<<"Kindly choose 1 or 2\n";
    goto i;
}

```

7. Total Complexity Analysis

The program's overall complexity is a combination of its features:

- Route Optimization: $O((V + E) \log V)$
- Financial Planning: $O(n * \text{capacity})$

Despite these complexities, the program ensures quick execution and efficient resource usage, making it reliable for users.

8. Results

The Journey Economizer program successfully achieves its objectives by:

- Calculating the most affordable travel routes.
- Providing actionable financial advice and savings plans.
- Offering a user-friendly experience with clear and concise outputs.

9. Limitations

- The program currently lacks a graphical interface, which could enhance usability.
- Limited to predefined travel routes and destinations.
- Does not include real-time travel data or currency exchange rates.
- Requires additional customization for advanced budgeting needs.

10. Conclusion

The Journey Economizer program combines travel and financial planning into a single tool. By using Dijkstra's Algorithm for travel optimization and the Knapsack Algorithm for budgeting, it delivers accurate, practical results. Future enhancements, such as a graphical interface, real-time data integration, and advanced financial tools, will further improve its functionality. These developments will make the program even more useful and appealing to a wider range of users.