### East West University

**Course Title : Algorithm**

**Course Code : CSE246**

**Section : 07**

**Semester : Fall2024**

**Project Report**

### ****Optimizing Road Travel in a City Network****

**Submitted To:**

Amit Mandal

Lecturer, department of Comp. Sci. and Eng.

**Submitted By**

|  |  |
| --- | --- |
| **Name** | **ID no.** |
| Swabirah Iffat Binte Ameer | 2022-3-60-067 |
| Tasnia Farin Ifa | 2022-3-60-096 |
| Sajid Ahmed | 2022-3-60-191 |
| Sadia Reza | 2022-3-60-073 |

Department of Computer Science And Engineering

**Date of Submission:**

2 Feb 2025

### ****Problem Statement****

We are a city planner responsible for improving travel efficiency between multiple cities in a region. The cities are connected by a network of roads, and each road has a specific **capacity** (maximum number of cars that can travel at once) and **cost** (construction or maintenance cost). Our goal is to: **Select the most important roads** for optimization based on their capacity and cost and **calculate the shortest travel times** between all pairs of cities to help commuters and logistics companies plan their routes.

**Problem Objective**

**Creating Roads**: We need data about the roads connecting the cities, including their capacities and costs.

The program randomly generates roads between cities. Each road has:

* + A starting city (city1).
  + A destination city (city2).
  + A weight representing its capacity or cost (randomly generated between 1 and 10).

**Getting Valuable Roads**: We have a limited budget (or capacity) for optimizing roads. We need to select the most important roads without exceeding the budget.

The program uses the **0/1 Knapsack algorithm** to select roads. Roads are chosen based on their weights, ensuring that the total cost (or capacity) does not exceed the maximum budget.

**Connecting Roads**: We need a way to represent the selected roads and their connections between cities.

The program creates a **graph** (adjacency matrix) where:

* + Each node represents a city.
  + Each edge represents a selected road, with its weight as the travel time or cost.

**Roads Connecting Cities**: We need to find the shortest travel times between all pairs of cities, considering the selected roads.

The program uses the **Floyd-Warshall algorithm** to compute the shortest paths between all cities. This algorithm updates the graph to reflect the most efficient travel routes.

**Getting Shortest Path**: We want to find the shortest paths from a specific city (e.g., City 0) in a directed acyclic road network.

The program uses **topological sort** to process cities in a specific order and compute the shortest paths from the starting city.

### ****Limitations****

### 1. ****Random Road Generation****

**Lack of Real-World Data**: The roads are generated randomly, which may not reflect real-world road networks. Real-world road networks have specific patterns, constraints, and geographical considerations that random generation cannot capture.

**No Consideration for Geographical Constraints**: The random generation does not account for geographical barriers like rivers, mountains, or urban layouts, which are crucial in real-world road planning.

### 2. ****0/1 Knapsack Problem****

**Simplistic Selection Criteria**: The 0/1 Knapsack algorithm selects roads based solely on their weights (capacities or costs). In reality, road selection criteria are more complex and may include factors like traffic volume, economic impact, and environmental considerations.

### 3. ****Floyd-Warshall Algorithm****

**Computational Complexity**: The Floyd-Warshall algorithm has a time complexity of O(n3)O(n3), where nn is the number of cities. This can be computationally expensive for large networks with many cities.

### 4. ****Topological Sort****

**Applicability**: Topological sort is only applicable to Directed Acyclic Graphs (DAGs). Real-world road networks may contain cycles, making this approach unsuitable.

**Single Source**: The topological sort approach only computes shortest paths from a single source city. For comprehensive planning, shortest paths from all cities might be necessary.

### ****5. Error Handling****

**Limited Error Handling**: The code has minimal error handling. For example, it does not handle cases where no roads are selected by the Knapsack algorithm, which could lead to incorrect or incomplete results.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <time.h>

#define MAX\_CITIES 100

#define MAX\_ROADS 1000

// Generate random roads between cities

void generate\_roads(int num\_cities, int num\_roads, int roads[][3]) {

srand(time(NULL));

for (int i = 0; i < num\_roads; i++) {

int city1 = rand() % num\_cities;

int city2 = rand() % num\_cities;

if (city1 == city2) {

city2 = (city2 + 1) % num\_cities;

int weight = rand() % 10 + 1;

roads[i][0] = city1;

roads[i][1] = city2;

roads[i][2] = weight;

}

}

// Solve the 0/1 Knapsack Problem

void knapsack(int num\_roads, int road\_weights[], int max\_capacity, int selected\_roads[]) {

int dp[max\_capacity + 1];

for (int i = 0; i <= max\_capacity; i++) {

dp[i] = 0;

}

for (int i = 0; i < num\_roads; i++) {

for (int j = max\_capacity; j >= road\_weights[i]; j--) {

if (dp[j - road\_weights[i]] + 1 > dp[j]) {

dp[j] = dp[j - road\_weights[i]] + 1;

selected\_roads[i] = 1;

}

}

}

}

// Create a graph from selected roads

void create\_graph(int num\_cities, int num\_roads, int roads[][3], int selected\_roads[], int graph[MAX\_CITIES][MAX\_CITIES]) {

for (int i = 0; i < num\_cities; i++) {

for (int j = 0; j < num\_cities; j++) {

if (i == j) {

graph[i][j] = 0;

} else {

graph[i][j] = INT\_MAX;

}

}

for (int i = 0; i < num\_roads; i++) {

if (selected\_roads[i]) {

int city1 = roads[i][0];

int city2 = roads[i][1];

int weight = roads[i][2];

graph[city1][city2] = weight;

graph[city2][city1] = weight;

}

}

}

// Apply Floyd-Warshall to find shortest paths

void floyd\_warshall(int num\_cities, int graph[MAX\_CITIES][MAX\_CITIES]) {

for (int k = 0; k < num\_cities; k++) {

for (int i = 0; i < num\_cities; i++) {

for (int j = 0; j < num\_cities; j++) {

if (graph[i][k] != INT\_MAX && graph[k][j] != INT\_MAX &&

graph[i][k] + graph[k][j] < graph[i][j]) {

graph[i][j] = graph[i][k] + graph[k][j];

}

}

}

}

}

// Function to perform topological sort using DFS

void topologicalSortUtil(int city, int num\_cities, int graph[MAX\_CITIES][MAX\_CITIES], int visited[], int stack[], int\* stack\_index) {

visited[city] = 1;

for (int i = 0; i < num\_cities; i++) {

if (graph[city][i] != INT\_MAX && !visited[i]) {

topologicalSortUtil(i, num\_cities, graph, visited, stack, stack\_index);

}

}

stack[(\*stack\_index)++] = city;

}

void topologicalSort(int num\_cities, int graph[MAX\_CITIES][MAX\_CITIES], int stack[]) {

int visited[MAX\_CITIES] = {0};

int stack\_index = 0;

for (int i = 0; i < num\_cities; i++) {

if (!visited[i]) {

topologicalSortUtil(i, num\_cities, graph, visited, stack, &stack\_index);

}

}

}

// Function to compute shortest paths using topological order

void shortestPathsTopological(int num\_cities, int graph[MAX\_CITIES][MAX\_CITIES], int start\_city, int dist[]) {

int stack[MAX\_CITIES];

topologicalSort(num\_cities, graph, stack);

// Initialize distances

for (int i = 0; i < num\_cities; i++) {

dist[i] = INT\_MAX;

}

dist[start\_city] = 0;

// Process cities in topological order

for (int i = num\_cities - 1; i >= 0; i--) {

int city = stack[i];

if (dist[city] != INT\_MAX) {

for (int j = 0; j < num\_cities; j++) {

if (graph[city][j] != INT\_MAX && dist[city] + graph[city][j] < dist[j]) {

dist[j] = dist[city] + graph[city][j];

}

}

}

}

}

// Report results

void print\_results(int num\_cities, int num\_roads, int roads[][3], int selected\_roads[], int graph[MAX\_CITIES][MAX\_CITIES]) {

printf("Selected Roads:\n");

for (int i = 0; i < num\_roads; i++) {

if (selected\_roads[i]) {

printf("City %d -> City %d (Weight: %d)\n", roads[i][0], roads[i][1], roads[i][2]);

}

}

printf("\nShortest Paths Between All Cities (Floyd-Warshall):\n");

for (int i = 0; i < num\_cities; i++) {

for (int j = 0; j < num\_cities; j++) {

if (graph[i][j] == INT\_MAX) {

printf("INF\t");

} else {

printf("%d\t", graph[i][j]);

}

}

printf("\n");

}

printf("\nShortest Paths from City 0 (Topological Sort):\n");

int dist[MAX\_CITIES];

shortestPathsTopological(num\_cities, graph, 0, dist);

for (int i = 0; i < num\_cities; i++) {

if (dist[i] == INT\_MAX) {

printf("City 0 -> City %d: INF\n", i);

} else {

printf("City 0 -> City %d: %d\n", i, dist[i]);

}

}

}

int main() {

int num\_cities, num\_roads;

int max\_capacity = 50;

// User input for number of cities and roads

printf("Enter the number of cities: ");

scanf("%d", &num\_cities);

printf("Enter the number of roads: ");

scanf("%d", &num\_roads);

if (num\_cities <= 0 || num\_roads <= 0 || num\_cities > MAX\_CITIES || num\_roads > MAX\_ROADS) {

printf("Invalid input size. Please try again.\n");

return 1;

}

int roads[MAX\_ROADS][3];

int road\_weights[MAX\_ROADS];

int selected\_roads[MAX\_ROADS] = {0};

int graph[MAX\_CITIES][MAX\_CITIES];

// Step 1: Generate Road Information

clock\_t start = clock();

generate\_roads(num\_cities, num\_roads, roads);

clock\_t end = clock();

double time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken to generate roads: %f seconds\n", time\_taken);

// Step 2: Extract road weights

for (int i = 0; i < num\_roads; i++) {

road\_weights[i] = roads[i][2];

}

// Step 3: Solve the 0/1 Knapsack Problem

start = clock();

knapsack(num\_roads, road\_weights, max\_capacity, selected\_roads);

end = clock();

time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken to solve the 0/1 Knapsack problem: %f seconds\n", time\_taken);

// Step 4: Create a Road Network

start = clock();

create\_graph(num\_cities, num\_roads, roads, selected\_roads, graph);

end = clock();

time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken to create the road network: %f seconds\n", time\_taken);

// Step 5: Apply Floyd-Warshall Algorithm

start = clock();

floyd\_warshall(num\_cities, graph);

end = clock();

time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken to apply Floyd-Warshall algorithm: %f seconds\n", time\_taken);

// Step 6: Apply Topological Sort (Optional)

start = clock();

int dist[MAX\_CITIES];

shortestPathsTopological(num\_cities, graph, 0, dist);

end = clock();

time\_taken = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Time taken to apply Topological Sort: %f seconds\n", time\_taken);

// Step 7: Report Results

print\_results(num\_cities, num\_roads, roads, selected\_roads, graph);

return 0;}

### ****Output:****

### 

### ****Graphs****

### **For Knapsack:**

### 

### **For Floyd-Warshall:**

### 

### **For Topological-sort:**

### 

### ****Conclusion****

This scenario demonstrates how the code can be applied to optimize road travel in a city network. By selecting the most important roads and calculating the shortest paths, city planners can make data-driven decisions to improve transportation efficiency and reduce costs. The program combines **0/1 Knapsack, Floyd-Warshall,** and **Topological Sort** to provide a comprehensive solution for real-world road optimization problems.