EAST WEST UNIVERSITY

Mini Project

Bus Route Optimization and Planning System

Course Code : CSE246

Course Title : Algorithm

Sec : 10

Submitted to:

Amit Modal

Lecturer

Department of Computer Science and Engineering

Submitted by:

Name : Aditya Debnath

ID : 2022-2-60-124

Name : Umme Habiba

ID: 2022-1-60-362

Name : Abdul Wadud Priyo

ID : 2022-2-60-133

Name : Mushfida Ferdous Maisha

ID : 2022-3-60-264

Date: 02/02/2025

Problem Statement:

The problem involves optimizing and planning bus routes using various algorithms. The system needs to handle the following tasks:

1. Sorting Bus Routes: Sort bus routes based on travel time or distance.
2. Finding the Shortest Path: Determine the shortest path between bus stops using Dijkstra's algorithm.
3. Selecting Buses: Select buses within a given budget to maximize passenger capacity using the 0/1 Knapsack problem.

Objectives:

1. Efficient Sorting: Implement a quick sort algorithm to sort bus routes based on travel time or distance.
2. Shortest Path Calculation: Use Dijkstra's algorithm to find the shortest path between bus stops.
3. Optimal Bus Selection: Use the 0/1 Knapsack algorithm to select buses within a given budget to maximize passenger capacity.
4. User Interaction: Allow users to input data or use default data for the system.
5. Performance Measurement: Measure and display the runtime of each algorithm.

Methodology:

Structures:

- **BusRoute:** Represents a bus route with attributes like route ID, start city, end city, travel time, and distance.
- **GraphEdge:** Represents an edge in the bus stop graph with attributes like neighbor stop and weight.
- **Bus:** Represents a bus with attributes like cost, capacity, and index.
- **BusStopGraph:** Represents the graph of bus stops with attributes like stops, adjacency list, number of edges, and number of stops.
- **PQItem:** Represents an item in the priority queue used in Dijkstra's algorithm.
- **PriorityQueue:** Represents the priority queue used in Dijkstra's algorithm.

Functions:

Quick Sort Implementation

- **compareRoutesByTime:** Compares two bus routes based on travel time.
- **compareRoutesByDistance:** Compares two bus routes based on distance.
- **quickSortRoutes:** Sorts bus routes using the Quick Sort algorithm.
- **partitionRoutes:** Partitions the array of bus routes for Quick Sort.
- **swapRoutes:** Swaps two bus routes.

Dijkstra's Algorithm Implementation

- **getStopIndex:** Gets the index of a stop in the bus stop graph.
- **addStop:** Adds a stop to the bus stop graph.
- **addEdge:** Adds an edge to the bus stop graph.
- **pq_init:** Initializes the priority queue.
- **pq_push:** Pushes an item into the priority queue.
- **pq_pop:** Pops an item from the priority queue.
- **pq_is_empty:** Checks if the priority queue is empty.
- **pq_heapify_up:** Heapifies the priority queue upwards.
- **pq_heapify_down:** Heapifies the priority queue downwards.
- **pq_swap_items:** Swaps two items in the priority queue.
- **dijkstraShortestPath:** Finds the shortest path between two stops using Dijkstra's algorithm.

0/1 Knapsack Problem Implementation

- **knapsackBusSelection:** Selects buses within a given budget to maximize passenger capacity using the 0/1 Knapsack algorithm.

Data Parsing Functions

- **parseBusRoutes:** Parses bus routes from a file.
- **parseBusStops:** Parses bus stops from a file.
- **parseBusEdges:** Parses bus edges from a file.
- **parseBuses:** Parses buses from a file.
- **parseBudget:** Parses the budget from a file.

- **parseDijkstraStops:** Parses the start and end stops for Dijkstra's algorithm from a file.
- **parseSortBy:** Parses the sorting option from a file.

File Handling Functions

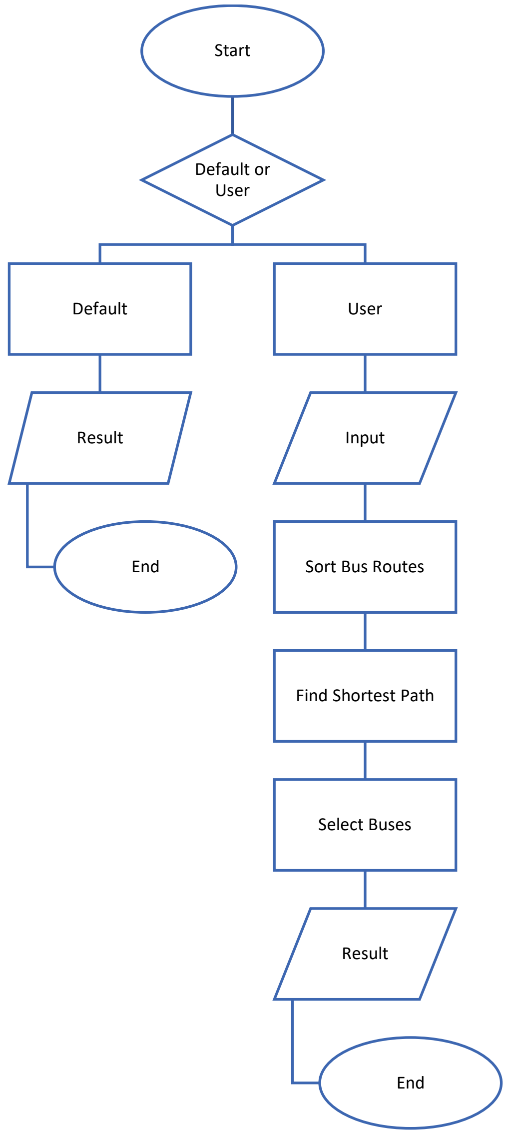
- **writeDefaultDataToFile:** Writes default data from a file to another file.
- **getUserInputToFile:** Gets user input and writes it to a file.

Main Function:

The main function integrates all the functionalities:

- **Data Input Mode:** Allows the user to choose between default data and user input.
- **File Handling:** Handles file operations to read and write data.
- **Quick Sort:** Sorts bus routes based on the selected option (time or distance).
- **Dijkstra's Algorithm:** Finds the shortest path between two bus stops.
- **0/1 Knapsack Problem:** Selects buses within the given budget to maximize passenger capacity.
- **Runtime Measurement:** Measures the runtime of each algorithm using gettimeofday.

Flowchart:



Start
Default or User
Default
User
Result
Input
End
Sort Bus Routes
Find Shortest Path
Select Buses
Result
End

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <ctype.h> // Required for tolower()
#include <time.h>   // Required for clock()
#include <sys/time.h> // Required for gettimeofday()

// --- Structures ---
typedef struct
{
   int route_id;
   char start_city[50];
   char end_city[50];
   int travel_time;
   int distance;
} BusRoute;

typedef struct
{
   char neighbor_stop[50];
   int weight;
} GraphEdge;

typedef struct
```

```c
{
    int cost;
    int capacity;
    int index;
} Bus;

// --- 1. Quick Sort Implementation for Sorting Bus Routes ---
int compareRoutesByTime(const void *a, const void *b);
int compareRoutesByDistance(const void *a, const void *b);
void quickSortRoutes(BusRoute routes[], int low, int high, const char *sort_by);
int partitionRoutes(BusRoute routes[], int low, int high, int (*compare)(const void *,
const void *));
void swapRoutes(BusRoute *a, BusRoute *b);

// --- 2. Dijkstra’s Algorithm Implementation for Shortest Path ---
#define MAX_STOPS 50
#define MAX_GRAPH_SIZE 50
typedef struct
{
    char stops[MAX_STOPS][50];
    GraphEdge adj_list[MAX_GRAPH_SIZE][MAX_STOPS];
    int num_edges[MAX_GRAPH_SIZE];
    int num_stops;
} BusStopGraph;

int getStopIndex(BusStopGraph *graph, const char *stop_name);
```

```
void addStop(BusStopGraph *graph, const char *stop_name);

void addEdge(BusStopGraph *graph, const char *from_stop, const char *to_stop, int weight);


typedef struct

{

   char stop_name[50];

   int distance;

} PQItem;

typedef struct

{

   PQItem items[MAX_STOPS];

   int size;

} PriorityQueue;

void pq_init(PriorityQueue *pq);

void pq_push(PriorityQueue *pq, const char *stop_name, int distance);

PQItem pq_pop(PriorityQueue *pq);

int pq_is_empty(PriorityQueue *pq);

void pq_heapify_up(PriorityQueue *pq, int index);

void pq_heapify_down(PriorityQueue *pq, int index);

void pq_swap_items(PriorityQueue *pq, int i, int j);


void dijkstraShortestPath(BusStopGraph *graph, const char *start_stop, const char *end_stop, int distances[], char previous_stops[][50]);


// --- 3. 0/1 Knapsack Problem Implementation for Bus Selection ---
```

```c
int knapsackBusSelection(Bus buses[], int n, int budget, int dp[][budget + 1], int keep[][budget + 1], int *selected_bus_indices, int max_buses);


// --- Data Parsing Functions ---
int parseBusRoutes(FILE *fp, BusRoute *bus_routes)
{
    char line[256];
    int num_routes = 0;
    while (fgets(line, sizeof(line), fp) != NULL)
    {
        if (strstr(line, "BUS_ROUTES_END")) break;
        if (strstr(line, "ROUTE"))
        {
            sscanf(line, "ROUTE %d %s %s %d %d", &bus_routes[num_routes].route_id, bus_routes[num_routes].start_city, bus_routes[num_routes].end_city, &bus_routes[num_routes].travel_time, &bus_routes[num_routes].distance);
            num_routes++;
        }
    }
    return num_routes;
}


int parseBusStops(FILE *fp, BusStopGraph *graph)
{
    char line[256];
    int num_stops = 0;
    graph->num_stops = 0;
```

```c
    while (fgets(line, sizeof(line), fp) != NULL)
    {
        if (strstr(line, "BUS_STOPS_END")) break;
        if (strstr(line, "STOP"))
        {
            char stop_name[50];
            sscanf(line, "STOP %s", stop_name);
            addStop(graph, stop_name);
            num_stops++;
        }
    }
    return num_stops;
}

int parseBusEdges(FILE *fp, BusStopGraph *graph)
{
    char line[256];
    int num_edges_parsed = 0;
    while (fgets(line, sizeof(line), fp) != NULL)
    {
        if (strstr(line, "BUS_EDGES_END")) break;
        if (strstr(line, "EDGE"))
        {
            char from_stop[50], to_stop[50];
            int weight;
            sscanf(line, "EDGE %s %s %d", from_stop, to_stop, &weight);
```

```
            addEdge(graph, from_stop, to_stop, weight);
            num_edges_parsed++;
        }
    }
    return num_edges_parsed;
}

int parseBuses(FILE *fp, Bus *buses)
{
    char line[256];
    int num_buses = 0;
    while (fgets(line, sizeof(line), fp) != NULL)
    {
        if (strstr(line, "BUSES_END")) break;
        if (strstr(line, "BUS"))
        {
            sscanf(line, "BUS %d %d", &buses[num_buses].cost,
&buses[num_buses].capacity);
            buses[num_buses].index = num_buses;
            num_buses++;
        }
    }
    return num_buses;
}

int parseBudget(FILE *fp)
```

```
{
    char line[256];
    int budget = 0;
    while (fgets(line, sizeof(line), fp) != NULL)
    {
        if (strstr(line, "BUDGET"))
        {
            sscanf(line, "BUDGET %d", &budget);
            break;
        }
    }
    return budget;
}

void parseDijkstraStops(FILE *fp, char *start_stop, char *end_stop)
{
    char line[256];
    while (fgets(line, sizeof(line), fp) != NULL)
    {
        if (strstr(line, "DIJKSTRA_STOPS"))
        {
            sscanf(line, "DIJKSTRA_STOPS %s %s", start_stop, end_stop);
            break;
        }
    }
}
```

```
void parseSortBy(FILE *fp, char *sort_by)
{
    char line[256];
    while (fgets(line, sizeof(line), fp) != NULL)
    {
        if (strstr(line, "SORT_BY"))
        {
            sscanf(line, "SORT_BY %s", sort_by);
            break;
        }
    }
}

// --- Function to write default data from code.txt to bus_data.txt ---
void writeDefaultDataToFile(const char *input_file, const char *output_file)
{
    FILE *in_fp = fopen(input_file, "r");
    if (in_fp == NULL)
    {
        perror("Error opening input file");
        return;
    }

    FILE *out_fp = fopen(output_file, "w");
    if (out_fp == NULL)
```

```
    {
        perror("Error opening output file");
        fclose(in_fp);
        return;
    }

    char line[256];
    while (fgets(line, sizeof(line), in_fp) != NULL)
    {
        fprintf(out_fp, "%s", line);
    }

    fclose(in_fp);
    fclose(out_fp);
    printf("Default data from '%s' written to '%s'.\n", input_file, output_file);
}

// --- Function to get user input and write to bus_data.txt ---
void getUserInputToFile(const char *output_file)
{
    FILE *fp = fopen(output_file, "w");
    if (fp == NULL)
    {
        perror("Error opening output file for writing user input");
        return;
    }
```

```
// --- Bus Routes ---
    fprintf(fp, "# Bus Routes Data\n");
    fprintf(fp, "BUS_ROUTES_START\n");
    int num_routes;
    printf("Enter the number of bus routes: ");
    scanf("%d", &num_routes);
    for (int i = 0; i < num_routes; i++)
    {
        BusRoute route;
        printf("\nEnter details for Route %d:\n", i + 1);
        printf("Route ID: ");
        scanf("%d", &route.route_id);
        printf("Start City: ");
        scanf("%s", route.start_city);
        printf("End City: ");
        scanf("%s", route.end_city);
        printf("Travel Time: ");
        scanf("%d", &route.travel_time);
        printf("Distance: ");
        scanf("%d", &route.distance);
        fprintf(fp, "ROUTE %d %s %s %d %d\n", route.route_id, route.start_city,
route.end_city, route.travel_time, route.distance);
    }
    fprintf(fp, "BUS_ROUTES_END\n\n");
```

```
// --- Bus Stops ---
    fprintf(fp, "# Bus Stop Graph Data\n");
    fprintf(fp, "BUS_STOPS_START\n");
    int num_stops_input;
    printf("\nEnter the number of bus stops: ");
    scanf("%d", &num_stops_input);
    for (int i = 0; i < num_stops_input; i++)
    {
        char stop_name[50];
        printf("Enter stop name %d: ", i + 1);
        scanf("%s", stop_name);
        fprintf(fp, "STOP %s\n", stop_name);
    }
    fprintf(fp, "BUS_STOPS_END\n\n");

// --- Bus Edges ---
    fprintf(fp, "BUS_EDGES_START\n");
    int num_edges_input;
    printf("\nEnter the number of connections (edges): ");
    scanf("%d", &num_edges_input);
    for (int i = 0; i < num_edges_input; i++)
    {
        char from_stop[50], to_stop[50];
        int weight;
        printf("\nEnter details for connection %d:\n", i + 1);
        printf("From Stop: ");
```

```
        scanf("%s", from_stop);
        printf("To Stop: ");
        scanf("%s", to_stop);
        printf("Travel Time (weight): ");
        scanf("%d", &weight);
        fprintf(fp, "EDGE %s %s %d\n", from_stop, to_stop, weight);
    }
    fprintf(fp, "BUS_EDGES_END\n\n");

// --- Buses Data for Knapsack ---
    fprintf(fp, "# Buses Data for Knapsack\n");
    fprintf(fp, "BUSES_START\n");
    int num_buses_knapsack;
    printf("\nEnter the number of bus types available: ");
    scanf("%d", &num_buses_knapsack);
    for (int i = 0; i < num_buses_knapsack; i++)
    {
        Bus bus;
        printf("\nEnter details for Bus Type %d:\n", i + 1);
        printf("Cost: ");
        scanf("%d", &bus.cost);
        printf("Capacity: ");
        scanf("%d", &bus.capacity);
        fprintf(fp, "BUS %d %d\n", bus.cost, bus.capacity);
    }
    fprintf(fp, "BUSES_END\n\n");
```

```
// --- Knapsack Budget ---
    fprintf(fp, "# Knapsack Budget\n");
    printf("\nEnter the budget for bus selection: ");
    int budget;
    scanf("%d", &budget);
    fprintf(fp, "BUDGET %d\n\n", budget);

// --- Dijkstra Start and End Stops ---
    fprintf(fp, "# Dijkstra Start and End Stops\n");
    char start_stop[50], end_stop[50];
    printf("\nEnter start stop for Dijkstra's: ");
    scanf("%s", start_stop);
    printf("Enter end stop for Dijkstra's: ");
    scanf("%s", end_stop);
    fprintf(fp, "DIJKSTRA_STOPS %s %s\n\n", start_stop, end_stop);

// --- Sort By ---
    fprintf(fp, "# Sort By (time or distance)\n");
    char sort_option[20];
    printf("\nSort routes by 'time' or 'distance'? ");
    scanf("%s", sort_option);
    fprintf(fp, "SORT_BY %s\n", sort_option);

    fclose(fp);
    printf("User input data written to '%s'.\n", output_file);
```

```
}

// --- Integration and Example Usage ---

int main()
{
   printf("--- Bus Route Optimization and Planning System ---\n\n");

   char data_input_mode[20];
   printf("Choose data input mode ('default' or 'user'): ");
   scanf("%s", data_input_mode);

   if (strcmp(data_input_mode, "default") == 0)
   {
      writeDefaultDataToFile("code.txt", "bus_data.txt");
   }
   else if (strcmp(data_input_mode, "user") == 0)
   {
      getUserInputToFile("bus_data.txt");
   }
   else
   {
      printf("Invalid data input mode. Using default data from 'code.txt'.\n");
      writeDefaultDataToFile("code.txt", "bus_data.txt");
   }
```

```
    FILE *fp = fopen("bus_data.txt", "r");
    if (fp == NULL)
    {
        char create_file_option;
        printf("bus_data.txt not found. Do you want to create it and enter data? (y/n): ");
        scanf(" %c", &create_file_option); // Note the space before %c to consume any
leading whitespace

        if (tolower(create_file_option) == 'y')
        {
            getUserInputToFile("bus_data.txt");
            fp = fopen("bus_data.txt", "r"); // Re-open in read mode after user input
            if (fp == NULL)
            {
                perror("Error re-opening bus_data.txt after user input");
                return 1;
            }
        }
        else
        {
            printf("Terminating program as bus_data.txt is needed and not created.\n");
            return 1; // Exit if user chooses not to create the file
        }
    }

    struct timeval start_time, end_time;
```

```
    double cpu_time_used_sort = 0.0;
    double cpu_time_used_dijkstra = 0.0;
    double cpu_time_used_knapsack = 0.0;
    double total_cpu_time_used = 0.0;

// --- 1. Quick Sort: Sorting Bus Routes ---
    printf("\n--- 1. Quick Sort: Sorting Bus Routes ---\n");
    BusRoute bus_routes[50]; // Assuming max 50 routes
    char bus_routes_buffer[256];
    while(fgets(bus_routes_buffer, sizeof(bus_routes_buffer), fp) != NULL)
    {
        if(strstr(bus_routes_buffer, "BUS_ROUTES_START")) break;
    }
    int num_routes = parseBusRoutes(fp, bus_routes);

    printf("\nOriginal Bus Routes:\n");
    for (int i = 0; i < num_routes; i++)
    {
        printf("Route ID: %d, Start: %s, End: %s, Time: %d, Distance: %d\n",
               bus_routes[i].route_id, bus_routes[i].start_city, bus_routes[i].end_city,
               bus_routes[i].travel_time, bus_routes[i].distance);
    }

    char sort_option[20] = "time"; // Default sort option
    rewind(fp);
    parseSortBy(fp, sort_option);
```

```
    BusRoute sorted_routes[num_routes];
    memcpy(sorted_routes, bus_routes, sizeof(BusRoute) * num_routes);

    gettimeofday(&start_time, NULL);
    for (int i = 0; i < 100000; i++) { // Increase the number of iterations to ensure
measurable time
        quickSortRoutes(sorted_routes, 0, num_routes - 1, sort_option);
    }
    gettimeofday(&end_time, NULL);
    cpu_time_used_sort = (end_time.tv_sec - start_time.tv_sec) + (end_time.tv_usec -
start_time.tv_usec) / 1e6;

    printf("\nSorted by %s:\n", sort_option);
    for (int i = 0; i < num_routes; i++)
    {
        printf("Route ID: %d, Start: %s, End: %s, Time: %d, Distance: %d\n",
               sorted_routes[i].route_id, sorted_routes[i].start_city, sorted_routes[i].end_city,
               sorted_routes[i].travel_time, sorted_routes[i].distance);
    }
    printf("Quick Sort Runtime: %f seconds\n", cpu_time_used_sort);

// --- 2. Dijkstra's Algorithm: Shortest Path between Bus Stops ---
    printf("\n--- 2. Dijkstra's Algorithm: Shortest Path between Bus Stops ---\n");
    BusStopGraph bus_stop_graph;
    bus_stop_graph.num_stops = 0;
```

```
rewind(fp);
char bus_stops_buffer_read[256];
while(fgets(bus_stops_buffer_read, sizeof(bus_stops_buffer_read), fp) != NULL)
{
   if(strstr(bus_stops_buffer_read, "BUS_STOPS_START")) break;
}
parseBusStops(fp, &bus_stop_graph);

rewind(fp);
char bus_edges_buffer_read[256];
while(fgets(bus_edges_buffer_read, sizeof(bus_edges_buffer_read), fp) != NULL)
{
   if(strstr(bus_edges_buffer_read, "BUS_EDGES_START")) break;
}
parseBusEdges(fp, &bus_stop_graph);

char start_stop[50] = "StopA", end_stop[50] = "StopE"; // Default stops
rewind(fp);
parseDijkstraStops(fp, start_stop, end_stop);

int distances[MAX_STOPS];
char previous_stops[MAX_STOPS][50];
printf("\nFinding shortest path from %s to %s...\n", start_stop, end_stop);

gettimeofday(&start_time, NULL);
```

```
    for (int i = 0; i < 100000; i++) { // Increase the number of iterations to ensure
measurable time
        dijkstraShortestPath(&bus_stop_graph, start_stop, end_stop, distances,
previous_stops);
    }
    gettimeofday(&end_time, NULL);
    cpu_time_used_dijkstra = (end_time.tv_sec - start_time.tv_sec) + (end_time.tv_usec -
start_time.tv_usec) / 1e6;

    int end_stop_index = getStopIndex(&bus_stop_graph, end_stop);
    if (distances[end_stop_index] != INT_MAX)
    {
        printf("Shortest Path Distance from %s to %s: %d\n", start_stop, end_stop,
distances[end_stop_index]);
        printf("All distances from Start Stop:\n");
        for(int i = 0; i < bus_stop_graph.num_stops; ++i)
        {
            printf("%s: %d\n", bus_stop_graph.stops[i], distances[i] == INT_MAX ? -1 :
distances[i]);
        }

        printf("Shortest Path: ");
        char path[MAX_STOPS * 50] = "";
        char current_stop_name[50];
        strcpy(current_stop_name, end_stop);

        while(strcmp(current_stop_name, "") != 0 && strcmp(current_stop_name,
start_stop) != 0)
```

```
        {
            strcat(path, current_stop_name);
            strcat(path, " <- ");
            strcpy(current_stop_name, previous_stops[getStopIndex(&bus_stop_graph,
current_stop_name)]);
        }
        strcat(path, start_stop);

        char reversed_path[MAX_STOPS * 50] = "";
        char *token = strtok(path, " <- ");
        char path_stops[MAX_STOPS][50];
        int num_path_stops = 0;
        while(token != NULL)
        {
            strcpy(path_stops[num_path_stops++], token);
            token = strtok(NULL, " <- ");
        }
        for(int i = num_path_stops - 1; i >= 0; --i)
        {
            strcat(reversed_path, path_stops[i]);
            if(i > 0) strcat(reversed_path, " -> ");
        }
        printf("%s\n", reversed_path);

    }
    else
```

```
    {
        printf("No path found from %s to %s.\n", start_stop, end_stop);
    }
    printf("Dijkstra's Algorithm Runtime: %f seconds\n", cpu_time_used_dijkstra);

// --- 3. 0/1 Knapsack Problem: Selecting Buses ---
    printf("\n--- 3. 0/1 Knapsack Problem: Selecting Buses ---\n");
    Bus buses[50]; // Assuming max 50 bus types
    rewind(fp);
    char buses_buffer_read[256];
    while(fgets(buses_buffer_read, sizeof(buses_buffer_read), fp) != NULL)
    {
        if(strstr(buses_buffer_read, "BUSES_START")) break;
    }
    int num_buses_knapsack = parseBuses(fp, buses);

    int budget = 350; // Default budget
    rewind(fp);
    budget = parseBudget(fp);

    printf("\nBus Costs: ");
    for (int i = 0; i < num_buses_knapsack; i++) printf("%d ", buses[i].cost);
    printf("\nBus Capacities: ");
    for (int i = 0; i < num_buses_knapsack; i++) printf("%d ", buses[i].capacity);
    printf("\nBudget: %d\n", budget);
```

```c
    int dp_knapsack[num_buses_knapsack + 1][budget + 1];
    int keep_knapsack[num_buses_knapsack + 1][budget + 1];
    int selected_bus_indices[num_buses_knapsack];
    int max_selected_buses = num_buses_knapsack;

    gettimeofday(&start_time, NULL);
    for (int i = 0; i < 100000; i++) { // Increase the number of iterations to ensure
measurable time
        knapsackBusSelection(buses, num_buses_knapsack, budget, dp_knapsack,
keep_knapsack, selected_bus_indices, max_selected_buses);
    }
    gettimeofday(&end_time, NULL);
    cpu_time_used_knapsack = (end_time.tv_sec - start_time.tv_sec) +
(end_time.tv_usec - start_time.tv_usec) / 1e6;

    printf("Maximum Passenger Capacity within Budget: %d\n",
dp_knapsack[num_buses_knapsack][budget]);
    printf("Selected Bus Indices (0-indexed): ");
    for (int i = 0; i < num_buses_knapsack; i++) selected_bus_indices[i] = -1; // Reset
indices
    knapsackBusSelection(buses, num_buses_knapsack, budget, dp_knapsack,
keep_knapsack, selected_bus_indices, max_selected_buses);

    int count_selected = 0;
    for(int i = 0; i < num_buses_knapsack; ++i)
    {
        if(selected_bus_indices[i] != -1) count_selected++;
    }
```

```
    printf("[");
    for (int i = 0; i < count_selected; i++)
    {
        if (selected_bus_indices[i] != -1)
        {
            printf("%d", selected_bus_indices[i]);
            if(i < count_selected -1) printf(", ");
        }
    }
    printf("]\n");

    printf("Selected Buses (Cost, Capacity):\n");
    for (int i = 0; i < count_selected; i++)
    {
        int index = selected_bus_indices[i];
        if (index != -1)
        {
            printf("  - Bus %d: Cost=%d, Capacity=%d\n", index + 1, buses[index].cost,
buses[index].capacity);
        }
    }
    printf("Knapsack Problem Runtime: %f seconds\n", cpu_time_used_knapsack);

    total_cpu_time_used = cpu_time_used_sort + cpu_time_used_dijkstra +
cpu_time_used_knapsack;
    printf("\n--- Total Runtime Summary ---\n");
```

```
    printf("Total Runtime for all algorithms: %f seconds\n", total_cpu_time_used);

    fclose(fp);
    return 0;
}

// --- Function Implementations (Quick Sort) ---
int compareRoutesByTime(const void *a, const void *b)
{
    return ((BusRoute *)a)->travel_time - ((BusRoute *)b)->travel_time;
}

int compareRoutesByDistance(const void *a, const void *b)
{
    return ((BusRoute *)a)->distance - ((BusRoute *)b)->distance;
}

void quickSortRoutes(BusRoute routes[], int low, int high, const char *sort_by)
{
    if (low < high)
    {
        int pi;
        if (strcmp(sort_by, "time") == 0)
        {
            pi = partitionRoutes(routes, low, high, compareRoutesByTime);
        }
```

```c
        else if (strcmp(sort_by, "distance") == 0)
        {
            pi = partitionRoutes(routes, low, high, compareRoutesByDistance);
        }
        else
        {
            return; // Invalid sort_by
        }

        quickSortRoutes(routes, low, pi - 1, sort_by);
        quickSortRoutes(routes, pi + 1, high, sort_by);
    }
}

int partitionRoutes(BusRoute routes[], int low, int high, int (*compare)(const void *,
const void *))
{
    BusRoute pivot = routes[high];
    int i = (low - 1);

    for (int j = low; j < high; j++)
    {
        if (compare(&routes[j], &pivot) < 0)
        {
            i++;
            swapRoutes(&routes[i], &routes[j]);
```

```
        }
    }
    swapRoutes(&routes[i + 1], &routes[high]);
    return (i + 1);
}

void swapRoutes(BusRoute *a, BusRoute *b)
{
    BusRoute temp = *a;
    *a = *b;
    *b = temp;
}

// --- Function Implementations (Dijkstra's) ---
int getStopIndex(BusStopGraph *graph, const char *stop_name)
{
    for (int i = 0; i < graph->num_stops; i++)
    {
        if (strcmp(graph->stops[i], stop_name) == 0)
        {
            return i;
        }
    }
    return -1; // Not found
}
```

```
void addStop(BusStopGraph *graph, const char *stop_name)
{
    if (graph->num_stops < MAX_STOPS)
    {
        strcpy(graph->stops[graph->num_stops], stop_name);
        graph->num_edges[graph->num_stops] = 0;
        graph->num_stops++;
    }
}

void addEdge(BusStopGraph *graph, const char *from_stop, const char *to_stop, int
weight)
{
    int from_index = getStopIndex(graph, from_stop);
    int to_index = getStopIndex(graph, to_stop);

    if (from_index != -1 && to_index != -1 && graph->num_edges[from_index] <
MAX_STOPS)
    {
        strcpy(graph->adj_list[from_index][graph-
>num_edges[from_index]].neighbor_stop, to_stop);
        graph->adj_list[from_index][graph->num_edges[from_index]].weight = weight;
        graph->num_edges[from_index]++;
    }
}

void pq_init(PriorityQueue *pq)
```

```
{
    pq->size = 0;
}

void pq_push(PriorityQueue *pq, const char *stop_name, int distance)
{
    if (pq->size < MAX_STOPS)
    {
        strcpy(pq->items[pq->size].stop_name, stop_name);
        pq->items[pq->size].distance = distance;
        pq->size++;
        pq_heapify_up(pq, pq->size - 1);
    }
}

PQItem pq_pop(PriorityQueue *pq)
{
    PQItem item = pq->items[0];
    pq->items[0] = pq->items[pq->size - 1];
    pq->size--;
    pq_heapify_down(pq, 0);
    return item;
}

int pq_is_empty(PriorityQueue *pq)
{
```

```
    return pq->size == 0;
}

void pq_heapify_up(PriorityQueue *pq, int index)
{
    int parent_index = (index - 1) / 2;
    while (index > 0 && pq->items[index].distance < pq->items[parent_index].distance)
    {
        pq_swap_items(pq, index, parent_index);
        index = parent_index;
        parent_index = (index - 1) / 2;
    }
}

void pq_heapify_down(PriorityQueue *pq, int index)
{
    int min_index = index;
    int left_child_index = 2 * index + 1;
    int right_child_index = 2 * index + 2;

    if (left_child_index < pq->size && pq->items[left_child_index].distance < pq-
>items[min_index].distance)
    {
        min_index = left_child_index;
    }
```

```
    if (right_child_index < pq->size && pq->items[right_child_index].distance < pq->items[min_index].distance)
    {
        min_index = right_child_index;
    }
    if (min_index != index)
    {
        pq_swap_items(pq, index, min_index);
        pq_heapify_down(pq, min_index);
    }
}

void pq_swap_items(PriorityQueue *pq, int i, int j)
{
    PQItem temp = pq->items[i];
    pq->items[i] = pq->items[j];
    pq->items[j] = temp;
}

void dijkstraShortestPath(BusStopGraph *graph, const char *start_stop, const char *end_stop, int distances[], char previous_stops[][50])
{
    for (int i = 0; i < graph->num_stops; i++)
    {
        distances[i] = INT_MAX;
        previous_stops[i][0] = '\0';
```

```
}

int start_index = getStopIndex(graph, start_stop);
int end_index = getStopIndex(graph, end_stop);

if (start_index == -1 || end_index == -1) return;

distances[start_index] = 0;
PriorityQueue pq;
pq_init(&pq);
pq_push(&pq, start_stop, 0);

while (!pq_is_empty(&pq))
{
    PQItem current_item = pq_pop(&pq);
    char current_stop_name[50];
    strcpy(current_stop_name, current_item.stop_name);
    int current_distance = current_item.distance;

    int current_index = getStopIndex(graph, current_stop_name);
    if (current_distance > distances[current_index])
    {
        continue;
    }

    for (int i = 0; i < graph->num_edges[current_index]; i++)
```

```
        {
            GraphEdge *edge = &graph->adj_list[current_index][i];
            int neighbor_index = getStopIndex(graph, edge->neighbor_stop);
            int distance = current_distance + edge->weight;

            if (distance < distances[neighbor_index])
            {
                distances[neighbor_index] = distance;
                strcpy(previous_stops[neighbor_index], current_stop_name);
                pq_push(&pq, edge->neighbor_stop, distance);
            }
        }
    }
}

// --- Function Implementations (Knapsack) ---
int knapsackBusSelection(Bus buses[], int n, int budget, int dp[][budget + 1], int
keep[][budget + 1], int *selected_bus_indices, int max_buses)
{
    for (int i = 0; i <= n; i++)
    {
        for (int w = 0; w <= budget; w++)
        {
            if (i == 0 || w == 0)
            {
                dp[i][w] = 0;
```

```
        }
        else if (buses[i - 1].cost <= w)
        {
            if (buses[i - 1].capacity + dp[i - 1][w - buses[i - 1].cost] > dp[i - 1][w])
            {
                dp[i][w] = buses[i - 1].capacity + dp[i - 1][w - buses[i - 1].cost];
                keep[i][w] = 1;
            }
            else
            {
                dp[i][w] = dp[i - 1][w];
            }
        }
        else
        {
            dp[i][w] = dp[i - 1][w];
        }
    }
}

int w = budget;
int selected_count = 0;
for (int i = n; i > 0 && w > 0 && selected_count < max_buses; i--)
{
    if (keep[i][w])
    {
```

```
            selected_bus_indices[selected_count++] = buses[i-1].index;
            w -= buses[i - 1].cost;
        }
    }
    return dp[n][budget];
}
```

Performance Analysis:

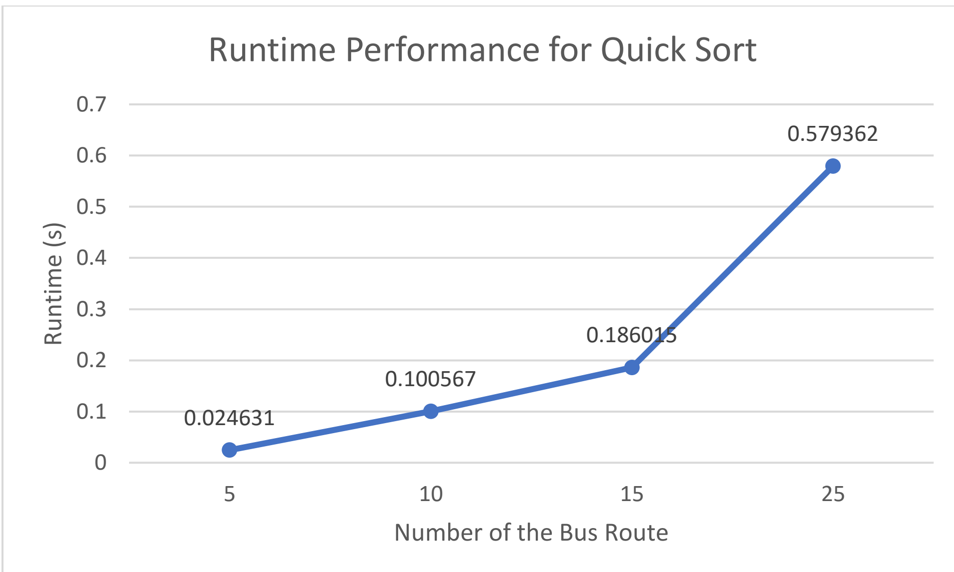


Fig 1 : Runtime Performance for Quick Sort

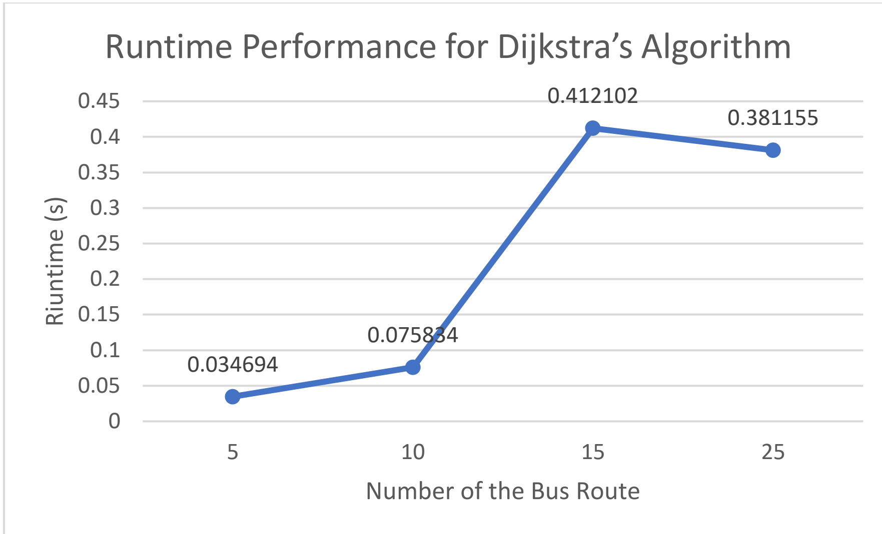


Fig 2 : Runtime Performance for Dijkstra’s Algorithm

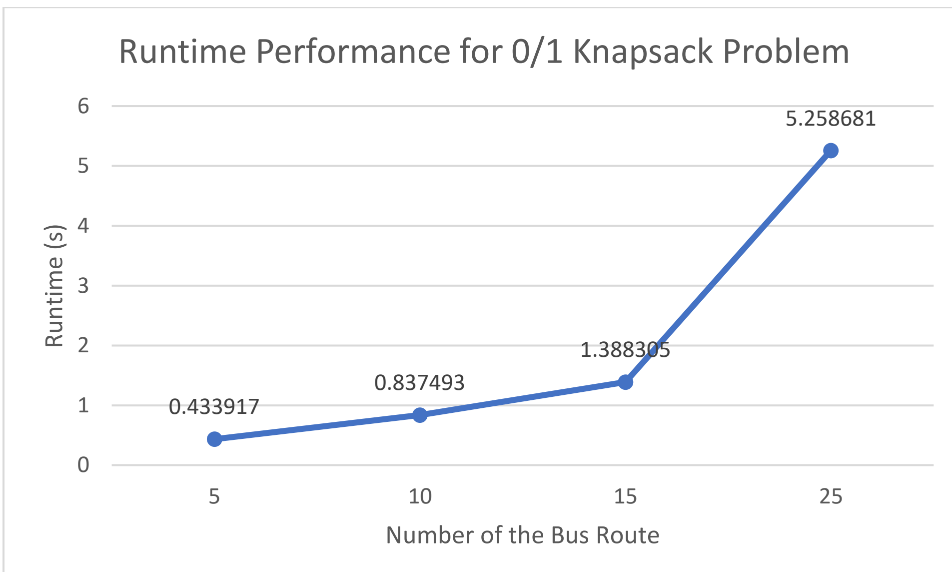


Fig 3 : Runtime Performance for 0/1 Knapsack Problem

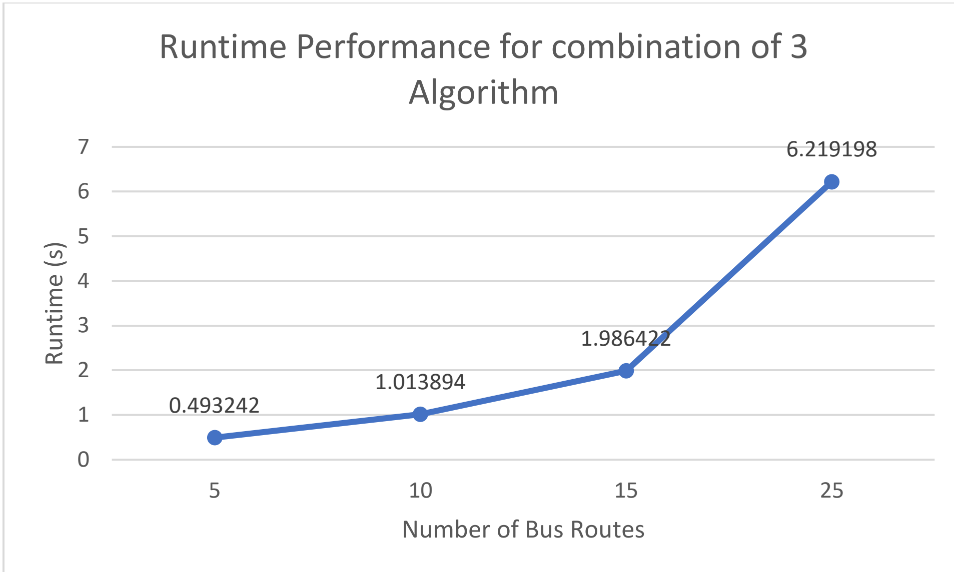


Fig 4 : Runtime Performance for 3 Algorithm

Output:

Output for 15 Bus Routes

```
--- 1. Quick Sort: Sorting Bus Routes ---

Original Bus Routes:
Route ID: 1, Start: A, End: B, Time: 60, Distance: 100
Route ID: 2, Start: C, End: D, Time: 120, Distance: 200
Route ID: 3, Start: B, End: E, Time: 45, Distance: 75
Route ID: 4, Start: D, End: F, Time: 90, Distance: 150
Route ID: 5, Start: E, End: G, Time: 75, Distance: 125
Route ID: 6, Start: F, End: H, Time: 150, Distance: 250
Route ID: 7, Start: G, End: I, Time: 60, Distance: 100
Route ID: 8, Start: H, End: J, Time: 180, Distance: 300
Route ID: 9, Start: I, End: K, Time: 45, Distance: 75
Route ID: 10, Start: J, End: L, Time: 120, Distance: 200
Route ID: 11, Start: K, End: M, Time: 90, Distance: 150
Route ID: 12, Start: L, End: N, Time: 75, Distance: 125
Route ID: 13, Start: M, End: O, Time: 60, Distance: 100
Route ID: 14, Start: N, End: P, Time: 150, Distance: 250
Route ID: 15, Start: O, End: A, Time: 210, Distance: 350

Sorted by time:
Route ID: 3, Start: B, End: E, Time: 45, Distance: 75
Route ID: 9, Start: I, End: K, Time: 45, Distance: 75
Route ID: 13, Start: M, End: O, Time: 60, Distance: 100
Route ID: 1, Start: A, End: B, Time: 60, Distance: 100
Route ID: 7, Start: G, End: I, Time: 60, Distance: 100
Route ID: 5, Start: E, End: G, Time: 75, Distance: 125
Route ID: 12, Start: L, End: N, Time: 75, Distance: 125
Route ID: 11, Start: K, End: M, Time: 90, Distance: 150
Route ID: 4, Start: D, End: F, Time: 90, Distance: 150
Route ID: 2, Start: C, End: D, Time: 120, Distance: 200
Route ID: 10, Start: J, End: L, Time: 120, Distance: 200
Route ID: 6, Start: F, End: H, Time: 150, Distance: 250
Route ID: 14, Start: N, End: P, Time: 150, Distance: 250
Route ID: 8, Start: H, End: J, Time: 180, Distance: 300
Route ID: 15, Start: O, End: A, Time: 210, Distance: 350
Quick Sort Runtime: 0.186015 seconds

--- 2. DijkstraÆs Algorithm: Shortest Path between Bus Stops ---

Finding shortest path from StopA to StopP...
Shortest Path Distance from StopA to StopP: 995
All distances from Start Stop:
StopA: 0
StopB: 60
StopC: 110
StopD: 230
StopE: 105
StopF: 320
```

```
Route ID: 14, Start: N, End: P, Time: 150, Distance: 250
Route ID: 8, Start: H, End: J, Time: 180, Distance: 300
Route ID: 15, Start: O, End: A, Time: 210, Distance: 350
Quick Sort Runtime: 0.186015 seconds

--- 2. Dijkstraâs Algorithm: Shortest Path between Bus Stops ---

Finding shortest path from StopA to StopP...
Shortest Path Distance from StopA to StopP: 995
All distances from Start Stop:
StopA: 0
StopB: 60
StopC: 110
StopD: 230
StopE: 105
StopF: 320
StopG: 180
StopH: 470
StopI: 240
StopJ: 650
StopK: 285
StopL: 770
Stopm: 375
StopN: 845
StopO: 435
StopP: 995
Shortest Path: StopA -> StopB -> StopC -> StopD -> StopF -> StopH -> StopJ -> StopL -> StopN -> StopP
Dijkstra's Algorithm Runtime: 0.412102 seconds

--- 3. 0/1 Knapsack Problem: Selecting Buses ---

Bus Costs: 50 80 120 30
Bus Capacities: 30 45 70 20
Budget: 300
Maximum Passenger Capacity within Budget: 165
Selected Bus Indices (0-indexed): [3, 2, 1, 0]
Selected Buses (Cost, Capacity):
  - Bus 4: Cost=30, Capacity=20
  - Bus 3: Cost=120, Capacity=70
  - Bus 2: Cost=80, Capacity=45
  - Bus 1: Cost=50, Capacity=30
Knapsack Problem Runtime: 1.388305 seconds

--- Total Runtime Summary ---
Total Runtime for all algorithms: 1.986422 seconds

Process returned 0 (0x0)   execution time : 16.142 s
Press any key to continue.
```

Output for 5 Bus Routes

```
--- Bus Route Optimization and Planning System ---

Choose data input mode ('default' or 'user'): default
Error opening input file: No such file or directory

--- 1. Quick Sort: Sorting Bus Routes ---

Original Bus Routes:
Route ID: 101, Start: CityA, End: CityB, Time: 60, Distance: 100
Route ID: 102, Start: CityB, End: CityC, Time: 45, Distance: 80
Route ID: 103, Start: CityC, End: CityD, Time: 75, Distance: 120
Route ID: 104, Start: CityD, End: CityA, Time: 90, Distance: 150
Route ID: 105, Start: CityA, End: CityC, Time: 50, Distance: 90

Sorted by time:
Route ID: 102, Start: CityB, End: CityC, Time: 45, Distance: 80
Route ID: 105, Start: CityA, End: CityC, Time: 50, Distance: 90
Route ID: 101, Start: CityA, End: CityB, Time: 60, Distance: 100
Route ID: 103, Start: CityC, End: CityD, Time: 75, Distance: 120
Route ID: 104, Start: CityD, End: CityA, Time: 90, Distance: 150
Quick Sort Runtime: 0.024631 seconds

--- 2. Dijkstraâs Algorithm: Shortest Path between Bus Stops ---

Finding shortest path from StopA to StopC...
Shortest Path Distance from StopA to StopC: 105
All distances from Start Stop:
StopA: 0
StopB: 60
StopC: 105
Stopd: 180
Shortest Path: StopA -> StopB -> StopC
Dijkstra's Algorithm Runtime: 0.034694 seconds

--- 3. 0/1 Knapsack Problem: Selecting Buses ---

Bus Costs: 50 80 120
Bus Capacities: 20 35 50
Budget: 150
Maximum Passenger Capacity within Budget: 55
Selected Bus Indices (0-indexed): [1, 0]
Selected Buses (Cost, Capacity):
  - Bus 2: Cost=80, Capacity=35
  - Bus 1: Cost=50, Capacity=20
Knapsack Problem Runtime: 0.433917 seconds

--- Total Runtime Summary ---
Total Runtime for all algorithms: 0.493242 seconds
```

Output for 10 Bus Routes

```
--- 1. Quick Sort: Sorting Bus Routes ---

Original Bus Routes:
Route ID: 101, Start: CityA, End: CityB, Time: 60, Distance: 100
Route ID: 102, Start: CityB, End: CityC, Time: 45, Distance: 80
Route ID: 103, Start: CityC, End: CityD, Time: 75, Distance: 120
Route ID: 104, Start: CityD, End: CityA, Time: 90, Distance: 150
Route ID: 105, Start: CityA, End: CityC, Time: 50, Distance: 90
Route ID: 106, Start: CityB, End: CityD, Time: 70, Distance: 110
Route ID: 107, Start: CityA, End: CityE, Time: 85, Distance: 140
Route ID: 108, Start: CityE, End: CityC, Time: 55, Distance: 95
Route ID: 109, Start: CityD, End: CityE, Time: 65, Distance: 105
Route ID: 110, Start: CityE, End: CityB, Time: 80, Distance: 130

Sorted by distance:
Route ID: 102, Start: CityB, End: CityC, Time: 45, Distance: 80
Route ID: 105, Start: CityA, End: CityC, Time: 50, Distance: 90
Route ID: 108, Start: CityE, End: CityC, Time: 55, Distance: 95
Route ID: 101, Start: CityA, End: CityB, Time: 60, Distance: 100
Route ID: 109, Start: CityD, End: CityE, Time: 65, Distance: 105
Route ID: 106, Start: CityB, End: CityD, Time: 70, Distance: 110
Route ID: 103, Start: CityC, End: CityD, Time: 75, Distance: 120
Route ID: 110, Start: CityE, End: CityB, Time: 80, Distance: 130
Route ID: 107, Start: CityA, End: CityE, Time: 85, Distance: 140
Route ID: 104, Start: CityD, End: CityA, Time: 90, Distance: 150
Quick Sort Runtime: 0.095393 seconds

--- 2. DijkstraÆs Algorithm: Shortest Path between Bus Stops ---

Finding shortest path from StopA to StopH...
Shortest Path Distance from StopA to StopH: 420
All distances from Start Stop:
StopA: 0
StopB: 60
StopC: 105
StopD: 180
StopE: 230
StopF: 285
StopG: 350
StopH: 420
Shortest Path: StopA -> StopB -> StopC -> StopD -> StopE -> StopF -> StopG -> StopH
Dijkstra's Algorithm Runtime: 0.069822 seconds

--- 3. 0/1 Knapsack Problem: Selecting Buses ---

Bus Costs: 50 80 120 70 90
Bus Capacities: 20 35 50 30 40
Budget: 250
```

```
--- 3. 0/1 Knapsack Problem: Selecting Buses ---

Bus Costs: 50 80 120 70 90
Bus Capacities: 20 35 50 30 40
Budget: 250
Maximum Passenger Capacity within Budget: 105
Selected Bus Indices (0-indexed): [2, 1, 0]
Selected Buses (Cost, Capacity):
  - Bus 3: Cost=120, Capacity=50
  - Bus 2: Cost=80, Capacity=35
  - Bus 1: Cost=50, Capacity=20
Knapsack Problem Runtime: 0.811747 seconds

--- Total Runtime Summary ---
Total Runtime for all algorithms: 0.976962 seconds

Process returned 0 (0x0)   execution time : 6.100 s
Press any key to continue.
```

Output for 25 Bus Routes

```
--- 1. Quick Sort: Sorting Bus Routes ---

Original Bus Routes:
Route ID: 101, Start: CityA, End: CityB, Time: 60, Distance: 100
Route ID: 102, Start: CityB, End: CityC, Time: 45, Distance: 80
Route ID: 103, Start: CityC, End: CityD, Time: 75, Distance: 120
Route ID: 104, Start: CityD, End: CityA, Time: 90, Distance: 150
Route ID: 105, Start: CityA, End: CityC, Time: 50, Distance: 90
Route ID: 106, Start: CityB, End: CityD, Time: 70, Distance: 110
Route ID: 107, Start: CityA, End: CityE, Time: 85, Distance: 140
Route ID: 108, Start: CityE, End: CityC, Time: 55, Distance: 95
Route ID: 109, Start: CityD, End: CityE, Time: 65, Distance: 105
Route ID: 110, Start: CityE, End: CityB, Time: 80, Distance: 130
Route ID: 111, Start: CityF, End: CityG, Time: 40, Distance: 70
Route ID: 112, Start: CityG, End: CityH, Time: 55, Distance: 90
Route ID: 113, Start: CityH, End: CityI, Time: 60, Distance: 100
Route ID: 114, Start: CityI, End: CityF, Time: 70, Distance: 115
Route ID: 115, Start: CityF, End: CityH, Time: 50, Distance: 85
Route ID: 116, Start: CityG, End: CityI, Time: 65, Distance: 105
Route ID: 117, Start: CityF, End: CityJ, Time: 75, Distance: 125
Route ID: 118, Start: CityJ, End: CityH, Time: 80, Distance: 130
Route ID: 119, Start: CityI, End: CityJ, Time: 45, Distance: 75
Route ID: 120, Start: CityJ, End: CityG, Time: 90, Distance: 150
Route ID: 121, Start: CityA, End: CityF, Time: 100, Distance: 170
Route ID: 122, Start: CityB, End: CityG, Time: 95, Distance: 160
Route ID: 123, Start: CityC, End: CityH, Time: 110, Distance: 185
Route ID: 124, Start: CityD, End: CityI, Time: 105, Distance: 180
Route ID: 125, Start: CityE, End: CityJ, Time: 120, Distance: 200

Sorted by time:
Route ID: 111, Start: CityF, End: CityG, Time: 40, Distance: 70
Route ID: 102, Start: CityB, End: CityC, Time: 45, Distance: 80
Route ID: 119, Start: CityI, End: CityJ, Time: 45, Distance: 75
Route ID: 105, Start: CityA, End: CityC, Time: 50, Distance: 90
Route ID: 115, Start: CityF, End: CityH, Time: 50, Distance: 85
Route ID: 112, Start: CityG, End: CityH, Time: 55, Distance: 90
Route ID: 108, Start: CityE, End: CityC, Time: 55, Distance: 95
Route ID: 113, Start: CityH, End: CityI, Time: 60, Distance: 100
Route ID: 101, Start: CityA, End: CityB, Time: 60, Distance: 100
Route ID: 109, Start: CityD, End: CityE, Time: 65, Distance: 105
Route ID: 116, Start: CityG, End: CityI, Time: 65, Distance: 105
Route ID: 114, Start: CityI, End: CityF, Time: 70, Distance: 115
Route ID: 106, Start: CityB, End: CityD, Time: 70, Distance: 110
Route ID: 117, Start: CityF, End: CityJ, Time: 75, Distance: 125
Route ID: 103, Start: CityC, End: CityD, Time: 75, Distance: 120
Route ID: 110, Start: CityE, End: CityB, Time: 80, Distance: 130
Route ID: 118, Start: CityJ, End: CityH, Time: 80, Distance: 130
Route ID: 107, Start: CityA, End: CityE, Time: 85, Distance: 140
Route ID: 104, Start: CityD, End: CityA, Time: 90, Distance: 150
```

```
Route ID: 104, Start: CityD, End: CityA, Time: 90, Distance: 150
Route ID: 120, Start: CityJ, End: CityG, Time: 90, Distance: 150
Route ID: 122, Start: CityB, End: CityG, Time: 95, Distance: 160
Route ID: 121, Start: CityA, End: CityF, Time: 100, Distance: 170
Route ID: 124, Start: CityD, End: CityI, Time: 105, Distance: 180
Route ID: 123, Start: CityC, End: CityH, Time: 110, Distance: 185
Route ID: 125, Start: CityE, End: CityJ, Time: 120, Distance: 200
Quick Sort Runtime: 0.579362 seconds

--- 2. DijkstraÆs Algorithm: Shortest Path between Bus Stops ---

Finding shortest path from StopA to StopO...
Shortest Path Distance from StopA to StopO: 635
All distances from Start Stop:
StopA: 0
StopB: 60
StopC: 50
StopD: 125
StopE: 130
StopF: 165
StopG: 190
StopH: 260
StopI: 300
StopJ: 355
StopK: 415
StopL: 485
StopM: 530
Stopn: 580
StopO: 635
Shortest Path: StopA -> StopC -> StopE -> StopG -> StopH -> StopI -> StopJ -> StopK -> StopL -> StopM -> Stopn -> StopO
Dijkstra's Algorithm Runtime: 0.381155 seconds

--- 3. 0/1 Knapsack Problem: Selecting Buses ---

Bus Costs: 50 80 120 70 90 150 110 60 130 100
Bus Capacities: 20 35 50 30 40 60 45 25 55 42
Budget: 500
Maximum Passenger Capacity within Budget: 212
Selected Bus Indices (0-indexed): [9, 6, 4, 2, 1]
Selected Buses (Cost, Capacity):
  - Bus 10: Cost=100, Capacity=42
  - Bus 7: Cost=110, Capacity=45
  - Bus 5: Cost=90, Capacity=40
  - Bus 3: Cost=120, Capacity=50
  - Bus 2: Cost=80, Capacity=35
Knapsack Problem Runtime: 5.258681 seconds

--- Total Runtime Summary ---
Total Runtime for all algorithms: 6.219198 seconds
```

Limitations:

File Size: The system assumes that the input file bus_data.txt will not exceed a reasonable size for in-memory processing.

Data Validation: The system does not perform extensive validation of user input data.

Performance: The system may not perform optimally for very large datasets due to the complexity of the algorithms used.

Concurrency: The system does not handle concurrent access to the data file, which may lead to data inconsistencies if multiple instances are run simultaneously.

Conclusion:

The Bus Route Optimization and Planning System provides an effective and well-structured solution for managing and optimizing bus routes. It streamlines key processes, such as sorting bus routes, identifying the shortest paths between stops, and selecting buses within a specified budget to maximize passenger capacity. Additionally, it efficiently handles file operations, user input, and runtime measurement to ensure smooth and reliable performance.

By incorporating Quick Sort for route sorting, Dijkstra's Algorithm for shortest path calculations, and the 0/1 Knapsack Problem for optimal bus selection, the system effectively tackles the core challenges of bus route optimization. The use of fprintf for formatted data storage ensures clarity and structure in data management, while runtime measurement using gettimeofday provides valuable insights into performance, helping to identify and address any inefficiencies.

Overall, this system presents a well-organized and efficient approach to bus route optimization. It follows a structured methodology, integrates key algorithms for enhanced decision-making, and accounts for potential limitations, making it a practical and scalable solution for improving public transportation planning.