|  |  |
| --- | --- |
|  | **DEPARTMENT OF COMPUTER ENGINEERING** |

**PBLE Report**

|  |  |
| --- | --- |
| Semester | S.E. Semester III – Computer Engineering |
| Subject | Analysis of Algorithm |
| Subject Professor In-charge | Prof. Swapnil S. Sonawane |

|  |  |
| --- | --- |
| Roll Number | Name of Students |
| 24102A0014 | Arpit Singh |
| 24102A0032 | Yash Ghadge |
| 24102A0028 | Parth Kumbhar |
|  |  |

**Name of the Project: College Bus Route Planner (TSP Solver)**

**Project Description:** The College Bus Route Planner is a console-based C program that calculates the optimal bus route for a set of college stops using the Traveling Salesperson Problem (TSP) algorithms. It supports exact and heuristic solutions, dynamic stop management, and file-based configuration.

The program allows users to:

Display all bus stops and the distance matrix.

Calculate shortest bus routes using:

Brute-Force (Exact)

Greedy / Nearest Neighbor (Heuristic)

Branch & Bound (Exact - Optimized)

Dynamically add new stops and modify distances.

Save and load configurations to/from a file.

Compare exact and heuristic algorithms**.**

**Project Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#include <stdbool.h>

#include <time.h>

#define MAX\_STOPS 20

#define INF 999999

#define CONFIG\_FILE "bus\_route\_config.txt"

// Structure to store route information

typedef struct {

int path[MAX\_STOPS];

int distance;

int numStops;

} Route;

// Structure for Branch and Bound node

typedef struct Node {

int level;

int path[MAX\_STOPS];

bool visited[MAX\_STOPS];

int cost;

int bound;

} Node;

char stopNames[MAX\_STOPS][50];

int distanceMatrix[MAX\_STOPS][MAX\_STOPS];

int numStops = 0;

int startStop = 0;

void initializeDefaultStops();

void displayMenu();

void displayStops();

void displayDistanceMatrix();

void findShortestRoute();

void handleAddedStop();

void compareAlgorithms();

void bruteForceTSP(Route \*bestRoute);

void greedyTSP(Route \*bestRoute);

void branchAndBoundTSP(Route \*bestRoute);

void nearestNeighborHeuristic(Route \*bestRoute);

void permute(int \*arr, int start, int end, Route \*bestRoute, int \*visited);

int calculateRouteDistance(int \*path, int length);

void printRoute(Route \*route);

void addNewStop();

void modifyDistances();

void saveConfiguration();

void loadConfiguration();

void fileIOMenu();

// Branch and Bound helper functions

int calculateInitialBound();

int calculateBound(Node \*node);

void copyNode(Node \*dest, Node \*src);

int findMinEdge(int vertex, bool \*visited);

Node\* createNode(int level);

int firstMin(int v);

int secondMin(int v);

int getIntInput(const char \*prompt, int \*out);

int main() {

int choice;

printf("\n╔════════════════════════════════════════════════╗\n");

printf("║ COLLEGE BUS ROUTE PLANNER (TSP SOLVER) ║\n");

printf("╚════════════════════════════════════════════════╝\n");

if(getIntInput("\nLoad previous configuration? (1=Yes, 0=No): ", &choice) != 0) {

// fallback

choice = 0;

}

if(choice == 1) {

loadConfiguration();

if(numStops == 0) {

printf(" No saved configuration found. Loading defaults...\n");

initializeDefaultStops();

}

} else {

initializeDefaultStops();

}

while(1) {

displayMenu();

if(getIntInput("\nEnter your choice: ", &choice) != 0) {

printf("\n Invalid input. Please enter a number.\n");

continue;

}

switch(choice) {

case 1:

displayStops();

break;

case 2:

displayDistanceMatrix();

break;

case 3:

findShortestRoute();

break;

case 4:

handleAddedStop();

break;

case 5:

compareAlgorithms();

break;

case 6:

addNewStop();

break;

case 7:

modifyDistances();

break;

case 8:

fileIOMenu();

break;

case 9:

printf("\n Thank you for using College Bus Route Planner!\n");

printf(" Safe travels! \n\n");

exit(0);

default:

printf("\n Invalid choice! Please try again.\n");

}

}

return 0;

}

void initializeDefaultStops() {

numStops = 6;

strcpy(stopNames[0], "Bus Depot (Start)");

strcpy(stopNames[1], "Main Gate");

strcpy(stopNames[2], "Library");

strcpy(stopNames[3], "Hostel Block A");

strcpy(stopNames[4], "Cafeteria");

strcpy(stopNames[5], "Sports Complex");

int distances[6][6] = {

{0, 5, 10, 15, 12, 20}, // Bus Depot

{5, 0, 8, 12, 7, 15}, // Main Gate

{10, 8, 0, 6, 9, 18}, // Library

{15, 12, 6, 0, 11, 8}, // Hostel Block A

{12, 7, 9, 11, 0, 13}, // Cafeteria

{20, 15, 18, 8, 13, 0} // Sports Complex

};

for(int i = 0; i < numStops; i++) {

for(int j = 0; j < numStops; j++) {

distanceMatrix[i][j] = distances[i][j];

}

}

}

void displayMenu() {

printf("\n╔════════════════════════════════════════════════╗\n");

printf("║ MAIN MENU ║\n");

printf("╠════════════════════════════════════════════════╣\n");

printf("║ 1. Display All Bus Stops ║\n");

printf("║ 2. Display Distance Matrix ║\n");

printf("║ 3. Find Shortest Route (Choose Algorithm) ║\n");

printf("║ 4. Handle Added Stop (Dynamic Route Update) ║\n");

printf("║ 5. Compare Exact vs Heuristic Solutions ║\n");

printf("║ 6. Add New Bus Stop ║\n");

printf("║ 7. Modify Distance Between Stops ║\n");

printf("║ 8. File I/O (Save/Load Configuration) ║\n");

printf("║ 9. Exit ║\n");

printf("╚════════════════════════════════════════════════╝\n");

}

void displayStops() {

printf("\n COLLEGE BUS STOPS:\n");

printf("═══════════════════════════════════════\n");

for(int i = 0; i < numStops; i++) {

printf(" [%d] %s\n", i, stopNames[i]);

}

printf("═══════════════════════════════════════\n");

}

void displayDistanceMatrix() {

printf("\n DISTANCE MATRIX (in 100m units):\n");

printf("═══════════════════════════════════════════════════\n");

printf("%-20s", "");

for(int i = 0; i < numStops; i++) {

printf("%4d ", i);

}

printf("\n");

for(int i = 0; i < numStops; i++) {

printf("%-20s", stopNames[i]);

for(int j = 0; j < numStops; j++) {

if(i == j)

printf("%4s ", "-");

else

printf("%4d ", distanceMatrix[i][j]);

}

printf("\n");

}

printf("═══════════════════════════════════════════════════\n");

}

void findShortestRoute() {

int choice;

Route bestRoute;

printf("\n SELECT ALGORITHM:\n");

printf("════════════════════════════════════════\n");

printf(" 1. Brute-Force TSP (Exact - Slow)\n");

printf(" 2. Greedy/Nearest Neighbor (Heuristic - Fast)\n");

printf(" 3. Branch & Bound (Exact - Optimized)\n");

printf("════════════════════════════════════════\n");

if(getIntInput("Enter choice: ", &choice) != 0) {

printf(" Invalid input!\n");

return;

}

printf("\n Computing optimal route...\n");

switch(choice) {

case 1:

if(numStops > 10) {

printf(" Warning: Brute-force may take long time for %d stops!\n", numStops);

printf(" Continue? (1=Yes, 0=No): ");

int cont;

if(getIntInput("", &cont) != 0) return;

if(!cont) return;

}

bruteForceTSP(&bestRoute);

printf("\n Brute-Force Algorithm Result:\n");

break;

case 2:

nearestNeighborHeuristic(&bestRoute);

printf("\n Greedy/Nearest Neighbor Result:\n");

break;

case 3:

branchAndBoundTSP(&bestRoute);

printf("\n Branch & Bound Algorithm Result:\n");

break;

default:

printf(" Invalid choice!\n");

return;

}

printRoute(&bestRoute);

}

void bruteForceTSP(Route \*bestRoute) {

int vertices[MAX\_STOPS];

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

int count = 0;

for(int i = 0; i < numStops; i++) {

if(i != startStop) {

vertices[count++] = i;

}

}

bestRoute->distance = INF;

bestRoute->numStops = numStops;

permute(vertices, 0, count - 1, bestRoute, visited);

}

void permute(int \*arr, int start, int end, Route \*bestRoute, int \*visited) {

if(start == end) {

// Create complete route: start -> permutation -> start

int route[MAX\_STOPS];

route[0] = startStop;

for(int i = 0; i <= end; i++) {

route[i + 1] = arr[i];

}

route[end + 2] = startStop;

int dist = calculateRouteDistance(route, end + 3);

if(dist < bestRoute->distance) {

bestRoute->distance = dist;

for(int i = 0; i < end + 3; i++) {

bestRoute->path[i] = route[i];

}

bestRoute->numStops = end + 3;

}

return;

}

for(int i = start; i <= end; i++) {

// Swap

int temp = arr[start];

arr[start] = arr[i];

arr[i] = temp;

permute(arr, start + 1, end, bestRoute, visited);

// Backtrack

temp = arr[start];

arr[start] = arr[i];

arr[i] = temp;

}

}

void nearestNeighborHeuristic(Route \*bestRoute) {

bool visited[MAX\_STOPS] = {false};

int currentStop = startStop;

int pathIndex = 0;

bestRoute->path[pathIndex++] = currentStop;

visited[currentStop] = true;

bestRoute->distance = 0;

for(int i = 1; i < numStops; i++) {

int nearestStop = -1;

int minDist = INF;

for(int j = 0; j < numStops; j++) {

if(!visited[j] && distanceMatrix[currentStop][j] < minDist) {

minDist = distanceMatrix[currentStop][j];

nearestStop = j;

}

}

if(nearestStop != -1) {

bestRoute->distance += minDist;

bestRoute->path[pathIndex++] = nearestStop;

visited[nearestStop] = true;

currentStop = nearestStop;

}

}

bestRoute->distance += distanceMatrix[currentStop][startStop];

bestRoute->path[pathIndex++] = startStop;

bestRoute->numStops = pathIndex;

}

Node\* createNode(int level) {

Node\* node = (Node\*)malloc(sizeof(Node));

if(!node) {

fprintf(stderr, "Memory allocation failed\n");

exit(1);

}

node->level = level;

node->cost = 0;

node->bound = 0;

for(int i = 0; i < MAX\_STOPS; i++) {

node->visited[i] = false;

node->path[i] = -1;

}

return node;

}

void copyNode(Node \*dest, Node \*src) {

// dest->level is intentionally NOT overwritten so that caller-created level remains

dest->cost = src->cost;

dest->bound = src->bound;

for(int i = 0; i < MAX\_STOPS; i++) {

dest->visited[i] = src->visited[i];

dest->path[i] = src->path[i];

}

}

int findMinEdge(int vertex, bool \*visited) {

int minEdge = INF;

for(int i = 0; i < numStops; i++) {

if(i != vertex && distanceMatrix[vertex][i] < minEdge) {

minEdge = distanceMatrix[vertex][i];

}

}

return minEdge;

}

int firstMin(int v) {

int min = INF;

for(int i = 0; i < numStops; i++) {

if(i != v && distanceMatrix[v][i] < min) {

min = distanceMatrix[v][i];

}

}

return min;

}

int secondMin(int v) {

int first = INF, second = INF;

for(int i = 0; i < numStops; i++) {

if(i == v) continue;

int val = distanceMatrix[v][i];

if(val <= first) {

second = first;

first = val;

} else if(val < second) {

second = val;

}

}

return second;

}

int calculateInitialBound() {

int sum = 0;

for(int i = 0; i < numStops; i++) {

int f = firstMin(i);

int s = secondMin(i);

if(f == INF) f = 0;

if(s == INF) s = 0;

sum += (f + s);

}

return (sum + 1) / 2;

}

int calculateBound(Node \*node) {

int bound = 0;

int min1, min2;

if(node->level > 0) {

for (int i = 0; i < node->level; i++) {

int a = node->path[i];

int b = node->path[i + 1];

if(a >= 0 && b >= 0) {

bound += distanceMatrix[a][b];

}

}

}

for (int i = 0; i < numStops; i++) {

min1 = min2 = INT\_MAX;

for (int j = 0; j < numStops; j++) {

if (i == j) continue;

int val = distanceMatrix[i][j];

if (val < min1) {

min2 = min1;

min1 = val;

} else if (val < min2) {

min2 = val;

}

}

if (min1 == INT\_MAX) min1 = 0;

if (min2 == INT\_MAX) min2 = 0;

if (!node->visited[i] || (node->level >= 0 && i == node->path[node->level])) {

bound += (min1 + min2) / 2;

}

}

return bound;

}

void branchAndBoundTSP(Route \*bestRoute) {

Node\* queue[100000];

int queueSize = 0;

Node\* root = createNode(0);

root->path[0] = startStop;

root->visited[startStop] = true;

root->cost = 0;

root->bound = calculateBound(root);

queue[queueSize++] = root;

int minCost = INF;

bestRoute->distance = INF;

int nodesExplored = 0;

while(queueSize > 0) {

int minIdx = 0;

for(int i = 1; i < queueSize; i++) {

if(queue[i]->bound < queue[minIdx]->bound) {

minIdx = i;

}

}

Node\* current = queue[minIdx];

for(int i = minIdx; i < queueSize - 1; i++) {

queue[i] = queue[i + 1];

}

queueSize--;

nodesExplored++;

if(current->bound >= minCost) {

free(current);

continue;

}

if(current->level == numStops - 1) {

int lastVertex = current->path[current->level];

if(distanceMatrix[lastVertex][startStop] < INF) {

int totalCost = current->cost + distanceMatrix[lastVertex][startStop];

if(totalCost < minCost) {

minCost = totalCost;

bestRoute->distance = totalCost;

for(int i = 0; i <= current->level; i++) {

bestRoute->path[i] = current->path[i];

}

bestRoute->path[current->level + 1] = startStop;

bestRoute->numStops = numStops + 1;

}

}

free(current);

continue;

}

int currentVertex = current->path[current->level];

for(int i = 0; i < numStops; i++) {

if(!current->visited[i]) {

Node\* child = createNode(current->level + 1);

copyNode(child, current);

child->path[child->level] = i;

child->visited[i] = true;

child->cost = current->cost + distanceMatrix[currentVertex][i];

child->bound = calculateBound(child);

if(child->bound < minCost) {

if(queueSize < (int)(sizeof(queue)/sizeof(queue[0]))) {

queue[queueSize++] = child;

} else {

// queue overflow fallback

free(child);

}

} else {

free(child);

}

}

}

free(current);

}

for(int i = 0; i < queueSize; i++) {

free(queue[i]);

}

if(bestRoute->distance == INF) {

printf(" No solution found or TSP unreachable.\n");

} else {

printf(" Nodes explored: %d\n", nodesExplored);

}

}

int calculateRouteDistance(int \*path, int length) {

int totalDist = 0;

for(int i = 0; i < length - 1; i++) {

totalDist += distanceMatrix[path[i]][path[i + 1]];

}

return totalDist;

}

void printRoute(Route \*route) {

printf("════════════════════════════════════════════════════\n");

printf(" OPTIMAL BUS ROUTE:\n");

for(int i = 0; i < route->numStops; i++) {

int idx = route->path[i];

if(idx < 0 || idx >= numStops) {

printf(" Stop %d: (invalid index)\n", i + 1);

} else {

printf(" Stop %d: %s\n", i + 1, stopNames[idx]);

if(i < route->numStops - 1) {

int dist = distanceMatrix[route->path[i]][route->path[i + 1]];

printf(" ↓ (Distance: %d units)\n", dist);

}

}

}

printf("\n TOTAL DISTANCE: %d units (%.1f km)\n",

route->distance, route->distance / 10.0);

printf("════════════════════════════════════════════════════\n");

}

void handleAddedStop() {

printf("\n HANDLE ADDED STOP\n");

printf("════════════════════════════════════════════════════\n");

if(numStops >= MAX\_STOPS) {

printf(" Maximum stops reached!\n");

return;

}

addNewStop();

printf("\n Recalculating optimal route with new stop...\n");

Route newRoute;

nearestNeighborHeuristic(&newRoute);

printRoute(&newRoute);

}

void compareAlgorithms() {

Route bruteForceRoute, heuristicRoute, branchBoundRoute;

double t\_brute = 0.0, t\_heuristic = 0.0, t\_branch = 0.0;

printf("\n COMPARING EXACT VS HEURISTIC SOLUTIONS\n");

printf("════════════════════════════════════════════════════\n");

if(numStops > 10) {

printf(" Warning: Brute-force may take long time for %d stops!\n", numStops);

printf(" Continue? (1=Yes, 0=No): ");

int cont;

if(getIntInput("", &cont) != 0) return;

if(!cont) {

printf("\n Running Nearest Neighbor (Heuristic)...\n");

clock\_t s2 = clock();

nearestNeighborHeuristic(&heuristicRoute);

clock\_t e2 = clock();

t\_heuristic = (double)(e2 - s2) / CLOCKS\_PER\_SEC;

printf(" Running Branch & Bound (Exact - Optimized)...\n");

clock\_t s3 = clock();

branchAndBoundTSP(&branchBoundRoute);

clock\_t e3 = clock();

t\_branch = (double)(e3 - s3) / CLOCKS\_PER\_SEC;

printf("\n COMPARISON RESULTS:\n");

printf("════════════════════════════════════════════════════\n");

printf("Branch & Bound Distance : %d units (time: %.3f s)\n", branchBoundRoute.distance, t\_branch);

printf("Heuristic Distance : %d units (time: %.3f s)\n", heuristicRoute.distance, t\_heuristic);

printf("Difference : %d units (%.1f%%)\n",

heuristicRoute.distance - branchBoundRoute.distance,

((branchBoundRoute.distance == 0) ? 0.0 : ((heuristicRoute.distance - branchBoundRoute.distance) \* 100.0) / branchBoundRoute.distance));

if(branchBoundRoute.distance == heuristicRoute.distance) {

printf(" Heuristic found OPTIMAL solution!\n");

} else {

printf(" Heuristic is sub-optimal but faster.\n");

}

printf("════════════════════════════════════════════════════\n");

return;

}

}

printf("\n Running Brute-Force (Exact)...\n");

clock\_t s1 = clock();

bruteForceTSP(&bruteForceRoute);

clock\_t e1 = clock();

t\_brute = (double)(e1 - s1) / CLOCKS\_PER\_SEC;

printf(" Running Nearest Neighbor (Heuristic)...\n");

clock\_t s2 = clock();

nearestNeighborHeuristic(&heuristicRoute);

clock\_t e2 = clock();

t\_heuristic = (double)(e2 - s2) / CLOCKS\_PER\_SEC;

printf(" Running Branch & Bound (Exact - Optimized)...\n");

clock\_t s3 = clock();

branchAndBoundTSP(&branchBoundRoute);

clock\_t e3 = clock();

t\_branch = (double)(e3 - s3) / CLOCKS\_PER\_SEC;

printf("\n COMPARISON RESULTS:\n");

printf("════════════════════════════════════════════════════\n");

printf("Brute-Force Distance : %d units (time: %.3f s)\n", bruteForceRoute.distance, t\_brute);

printf("Branch & Bound Distance : %d units (time: %.3f s)\n", branchBoundRoute.distance, t\_branch);

printf("Heuristic Distance : %d units (time: %.3f s)\n", heuristicRoute.distance, t\_heuristic);

if (bruteForceRoute.distance != 0) {

printf("\nBrute-Force vs Heuristic: %d units (%.1f%% diff)\n",

heuristicRoute.distance - bruteForceRoute.distance,

((heuristicRoute.distance - bruteForceRoute.distance) \* 100.0) / bruteForceRoute.distance);

} else {

printf("\nBrute-Force vs Heuristic: (division by zero avoided)\n");

}

if (branchBoundRoute.distance != 0) {

printf("Branch & Bound vs Heuristic: %d units (%.1f%% diff)\n",

heuristicRoute.distance - branchBoundRoute.distance,

((heuristicRoute.distance - branchBoundRoute.distance) \* 100.0) / branchBoundRoute.distance);

} else {

printf("Branch & Bound vs Heuristic: (division by zero avoided)\n");

}

if(bruteForceRoute.distance == heuristicRoute.distance) {

printf(" Heuristic found OPTIMAL solution!\n");

} else {

printf(" Heuristic is sub-optimal but faster.\n");

}

if(bruteForceRoute.distance == branchBoundRoute.distance) {

printf(" Branch & Bound matches Brute-Force (both optimal)!\n");

} else {

printf(" Verification needed - algorithms differ.\n");

}

printf("════════════════════════════════════════════════════\n");

}

void addNewStop() {

if(numStops >= MAX\_STOPS) {

printf(" Cannot add more stops! Maximum limit reached.\n");

return;

}

printf("\nEnter new stop name: ");

getchar(); // Clear buffer leftover newline

fgets(stopNames[numStops], 50, stdin);

stopNames[numStops][strcspn(stopNames[numStops], "\n")] = 0;

printf("\nEnter distances from new stop to existing stops:\n");

for(int i = 0; i < numStops; i++) {

char prompt[200];

snprintf(prompt, sizeof(prompt), " Distance to %s: ", stopNames[i]);

if(getIntInput(prompt, &distanceMatrix[numStops][i]) != 0) {

printf("Invalid input, setting distance to INF.\n");

distanceMatrix[numStops][i] = INF;

}

distanceMatrix[i][numStops] = distanceMatrix[numStops][i]; // Symmetric

}

distanceMatrix[numStops][numStops] = 0;

numStops++;

printf("\n Stop added successfully! Total stops: %d\n", numStops);

}

void modifyDistances() {

int from, to, newDist;

displayStops();

printf("\nEnter stop indices to modify distance:\n");

if(getIntInput("From stop: ", &from) != 0) return;

if(getIntInput("To stop: ", &to) != 0) return;

if(from < 0 || from >= numStops || to < 0 || to >= numStops) {

printf(" Invalid stop indices!\n");

return;

}

printf("Current distance: %d units\n", distanceMatrix[from][to]);

if(getIntInput("Enter new distance: ", &newDist) != 0) return;

distanceMatrix[from][to] = newDist;

distanceMatrix[to][from] = newDist; // Symmetric

printf(" Distance updated successfully!\n");

}

void saveConfiguration() {

FILE \*file = fopen(CONFIG\_FILE, "w");

if(file == NULL) {

printf(" Error: Unable to create configuration file!\n");

return;

}

fprintf(file, "%d\n", numStops);

for(int i = 0; i < numStops; i++) {

fprintf(file, "%s\n", stopNames[i]);

}

for(int i = 0; i < numStops; i++) {

for(int j = 0; j < numStops; j++) {

fprintf(file, "%d ", distanceMatrix[i][j]);

}

fprintf(file, "\n");

}

fprintf(file, "%d\n", startStop);

fclose(file);

printf(" Configuration saved successfully to '%s'!\n", CONFIG\_FILE);

}

void loadConfiguration() {

FILE \*file = fopen(CONFIG\_FILE, "r");

if(file == NULL) {

printf(" Configuration file not found.\n");

return;

}

if(fscanf(file, "%d\n", &numStops) != 1) {

printf(" Error reading configuration file.\n");

fclose(file);

numStops = 0;

return;

}

for(int i = 0; i < numStops; i++) {

if(fgets(stopNames[i], 50, file) == NULL) {

stopNames[i][0] = '\0';

} else {

stopNames[i][strcspn(stopNames[i], "\n")] = 0;

}

}

for(int i = 0; i < numStops; i++) {

for(int j = 0; j < numStops; j++) {

if(fscanf(file, "%d", &distanceMatrix[i][j]) != 1) {

distanceMatrix[i][j] = INF;

}

}

}

if(fscanf(file, "%d", &startStop) != 1) {

startStop = 0;

}

fclose(file);

printf(" Configuration loaded successfully from '%s'!\n", CONFIG\_FILE);

printf(" Total stops: %d\n", numStops);

}

void fileIOMenu() {

int choice;

printf("\n FILE I/O MENU\n");

printf("════════════════════════════════════════\n");

printf(" 1. Save Current Configuration\n");

printf(" 2. Load Configuration\n");

printf(" 3. View Configuration File Info\n");

printf(" 4. Back to Main Menu\n");

printf("════════════════════════════════════════\n");

if(getIntInput("Enter choice: ", &choice) != 0) {

printf(" Invalid input!\n");

return;

}

switch(choice) {

case 1:

saveConfiguration();

break;

case 2:

loadConfiguration();

break;

case 3:

printf("\n Configuration File: %s\n", CONFIG\_FILE);

printf("Current stops: %d\n", numStops);

if(startStop >= 0 && startStop < numStops)

printf("Start stop: %s\n", stopNames[startStop]);

else

printf("Start stop: (invalid index)\n");

break;

case 4:

return;

default:

printf(" Invalid choice!\n");

}

}

int getIntInput(const char \*prompt, int \*out) {

char buffer[256];

char \*endptr;

long val;

if(prompt && prompt[0] != '\0') {

printf("%s", prompt);

}

if(!fgets(buffer, sizeof(buffer), stdin)) {

return -1; // input failure

}

buffer[strcspn(buffer, "\n")] = 0;

if(buffer[0] == '\0') {

return -1;

}

val = strtol(buffer, &endptr, 10);

if(endptr == buffer || \*endptr != '\0') {

printf(" Invalid number entered. Please enter numeric digits only.\n");

return -1;

}

if(val > INT\_MAX || val < INT\_MIN) return -1;

\*out = (int)val;

return 0;

}

**Output Screenshots:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A white background with black text

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A white rectangular sign with black text

AI-generated content may be incorrect.**

**A screenshot of a computer menu

AI-generated content may be incorrect.**

**GitHub Repository Link of Project:**