UCSD 167113 Data Structure And Algorithms in C/C++
HW8
Cheng FEI

Source Code:

```c
//
//  main.c
//  maze-path-finder
//
//  Created by Cheng FEI on 2022/11/15.
//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "graph.h"
#include "dfs.h"

// Function Prototypes.

// Prototype of the major path finding algorithm.
int isExitReachable(Graph *pMaze, char entrance, char exit);
// Prototype of the recursive helper function.
int isExitReachableRecursive(Graph *graph, AdjList *adjList, char exit);
// Prototype of graph's match function.
int match(const void *key1, const void *key2);
// Prototype of graph's initialization functions.
int initGraph1(Graph *graph1, char chars[]);
int initGraph2(Graph *graph2, char chars[]);
int logEdgeInitStatus(int retval);

int main(int argc, const char * argv[]) {
    int result = 0, retval = 0;
    char * exist = NULL;
    char chars[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};

    // Initialize the first graph.
    printf("The first graph test...\n");
    Graph graph1;
    if ((retval = initGraph1(&graph1, chars)) == -1) {
        graph_destroy(&graph1);
        exit(EXIT_FAILURE);
    }
    // Check whether the path exists.
    result = isExitReachable(&graph1, 'A', 'G');
    if (0 != result) exist = "True";
    else exist = "False";
    printf("The path exists: %s\n", exist);
    printf("Expected: True\n");
    printf("------------------------------------------\n");
    // Destroy the graph.
    graph_destroy(&graph1);

    // Initialize the second graph.
    printf("The second graph test...\n");
    Graph graph2;
    if ((retval = initGraph2(&graph2, chars)) == -1) {
        graph_destroy(&graph2);
        exit(EXIT_FAILURE);
    }
    // Check whether the path exists.
    result = isExitReachable(&graph2, 'A', 'G');
    if (0 != result) exist = "True";
    else exist = "False";
    printf("The path exists: %s\n", exist);
    printf("Expected: False\n");
    // Destroy the graph.
    graph_destroy(&graph2);
    return EXIT_SUCCESS;
}

// Function Implementations.

// Checks whether a path exists from entrance to exit.
// @Params: pMaze -- pointer to the graph to search
// @Params: entrance -- the entrance vertex's character
// @Params: exit -- the exit vertex's character.
// @Returns: 1 indicates that a path exists.
//          0 zero return value indicates that no path exists.
int isExitReachable(Graph *pMaze, char entrance, char exit) {
    ListElmt *curListElmt = NULL;
    AdjList *curAdjList = NULL;
    DfsVertex *curVertex = NULL;
    char *curChar = NULL;

    // Initialize the color of all vertices to white.
    for (curListElmt = list_head(&pMaze->adjlists); curListElmt != NULL; curListElmt =
list_next(curListElmt)) {
        curAdjList = (AdjList *) curListElmt->data;
        curVertex = (DfsVertex *) curAdjList->vertex;
        curVertex->color = white;
    }

    // If the exit character doesn't exist, then the path doesn't exist.
    int found = 0; // 0 indicates that the vertex is not found.
    for (curListElmt = list_head(&pMaze->adjlists); curListElmt != NULL; curListElmt =
list_next(curListElmt)) {
        curAdjList = (AdjList *) curListElmt->data;
        curVertex = (DfsVertex *) curAdjList->vertex;
        if (*((char *) curVertex) == exit) {
            found = 1;
            break;
        }
    }
    if (0 == found) return 0;
    // If the entrance character doesn't exist, then the path doesn't exist.
    found = 0;
    for (curListElmt = list_head(&pMaze->adjlists); curListElmt != NULL; curListElmt =
list_next(curListElmt)) {
        curAdjList = (AdjList *) curListElmt->data;
        curVertex = (DfsVertex *) curAdjList->vertex;
        curChar = (char *) curVertex->data;
        if (*((char *) curVertex) == entrance) {
            found = 1;
            break;
        }
    }
    if (0 == found) return 0;

    // Call the recursive function:
    // Traverse adjacent vertexes using the DFS algorithm.
    found = isExitReachableRecursive(pMaze, curAdjList, exit);
    if (1 == found) return 1;

    return 0;
}

// Checks whether a path exists from the current vertex to exit.
// @Params: graph -- pointer to the graph to search
// @Params: ahjList -- pointer to the adjList with the current vertex's character.
// @Params: exit -- the exit vertex's character.
// @Returns: 1 indicates that a path exists.
//          0 indicates that no path exists.
//          -1 indicates that some vertices are out of the graph.
int isExitReachableRecursive(Graph *graph, AdjList *adjList, char exit) {
    ListElmt *curMember;
    DfsVertex *curVertex = (DfsVertex *) adjList->vertex;
    AdjList *curAdjList;
    Set curAdjacent;
```

```c
    // Base case:
    // If the character of the vertex matches that of the exit vertex, return 1.
    if (*((char *) curVertex) == exit) return 1;

    // Recursive case:
    // Else, call the same function on every adjacent vertices.
    curVertex->color = gray;
    curAdjacent = adjList->adjacent;
    for (curMember = list_head(&curAdjacent); curMember != NULL; curMember = list_next(curMember)) {
        // Retrieve the adjacent lists.
        curVertex = (DfsVertex *) list_data(curMember);
        if (curVertex->color == white) {
            if (graph_adjlist(graph, (const void *) curVertex, &curAdjList) != 0) return -1;
            if (isExitReachableRecursive(graph, curAdjList, exit) == 1) return 1;
        }
    }

    return 0;
}

// Checks whether vertex key1 and vertex key2 are the same vertex.
// @Params: key1 -- pointer to the first vertex.
// @Params: key2 -- pointer to the second vertex.
// @Returns: 1 indicates that vertexes match.
//           0 indicates that vertexes don't match.
// @Requires: both key1 and key 2 are of type char *.
int match(const void *key1, const void *key2) {
    char *char1, *char2;
    char1 = (char *) key1;
    char2 = (char *) key2;
    return *char1 == *char2;
}

// Initialize graph1 and fill the vertexes and edges.
// @Params: graph1 -- pointer to the graph to initialize.
// @Params: chars -- predefined character array for vertexes.
// @Returns: 0 indicates that everything is OK.
//          -1 indicates that the system runs out of memory
//             or the user incorrectly inserts duplicate edges.
int initGraph1(Graph *graph1, char chars[]) {
    int retval;
    // Initialize an empty graph.
    graph_init(graph1, match, NULL);
    // Initialize vertexes.
    for (int i = 0; i < 7; i++) {
        retval = graph_ins_vertex(graph1, &chars[i]);
        if (1 == retval) printf("*** Warning: '%c' already exists! ***\n", chars[i]);
        else if (-1 == retval) {
            printf("*** Error: Out of memory! ***\n");
            fprintf(stderr, "Out of memory!");
            return -1;
        }
    }
    // Initialize edges.
    retval = graph_ins_edge(graph1, (const void *) &chars[0], (const void *) &chars[2]); // A -> C
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[0], (const void *) &chars[3]); // A -> D
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[1], (const void *) &chars[3]); // B -> D
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[2], (const void *) &chars[0]); // C -> A
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[2], (const void *) &chars[5]); // C -> F
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[3], (const void *) &chars[0]); // D -> A
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[3], (const void *) &chars[1]); // D -> B
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[3], (const void *) &chars[4]); // D -> E
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[3], (const void *) &chars[6]); // D -> G
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[4], (const void *) &chars[3]); // E -> D
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[4], (const void *) &chars[6]); // E -> G
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[5], (const void *) &chars[2]); // F -> C
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[5], (const void *) &chars[6]); // F -> G
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[6], (const void *) &chars[3]); // G -> D
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[6], (const void *) &chars[4]); // G -> E
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph1, (const void *) &chars[6], (const void *) &chars[5]); // G -> F
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;

    return 0;
}

// Initialize graph2 and fill the vertexes and edges.
// @Params: graph2 -- pointer to the graph to initialize.
// @Params: chars -- predefined character array for vertexes.
// @Returns: 0 indicates that everything is OK.
//           -1 indicates that the system runs out of memory
//              or the user incorrectly inserts duplicate edges.
int initGraph2(Graph *graph2, char chars[]) {
    int retval;
    // Initialize an empty graph.
    graph_init(graph2, match, NULL);
    // Initialize vertexes.
    for (int i = 0; i < 7; i++) {
        retval = graph_ins_vertex(graph2, (const void *) &chars[i]);
        if (1 == retval) printf("*** Warning: '%c' already exists! ***\n", chars[i]);
        else if (-1 == retval) {
            printf("*** Error: Out of memory! ***\n");
            fprintf(stderr, "Out of memory!");
            return -1;
        }
    }
    // Initialize edges.
    retval = graph_ins_edge(graph2, (const void *) &chars[0], (const void *) &chars[2]); // A -> C
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph2, (const void *) &chars[0], (const void *) &chars[3]); // A -> D
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph2, (const void *) &chars[1], (const void *) &chars[3]); // B -> D
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph2, (const void *) &chars[2], (const void *) &chars[0]); // C -> A
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph2, (const void *) &chars[2], (const void *) &chars[5]); // C -> F
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph2, (const void *) &chars[3], (const void *) &chars[0]); // D -> A
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph2, (const void *) &chars[3], (const void *) &chars[1]); // D -> B
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph2, (const void *) &chars[4], (const void *) &chars[6]); // E -> G
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph2, (const void *) &chars[5], (const void *) &chars[2]); // F -> C
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;
    retval = graph_ins_edge(graph2, (const void *) &chars[6], (const void *) &chars[4]); // G -> E
    if ((retval = logEdgeInitStatus(retval)) == -1) return -1;

    return 0;
}

// Log the status of the edge initialization to the console and/or to the system.
// @Params: retval -- status return value indicating the initialization status.
// @Returns: retval
int logEdgeInitStatus(int retval) {
    if (1 == retval) printf("*** Warning: Do not allow insertion of duplicate edges! ***\n");
    else if (-1 == retval) {
        printf("*** Error: Out of memory / Vertex not found! ***\n");
        fprintf(stderr, "Out of memory / Vertex not found!");
    }
    return retval;
}
```

Console Output:

```
The first graph test...
The path exists: True
Expected: True
------------------------------------------
The second graph test...
The path exists: False
Expected: False

Program ended with exit code: 0
```