# Санкт-Петербургский государственный политехнический университет

Кафедра “Прикладная математика”

Отчет по лабораторной работе 3

“Алгоритмы и структуры данных”

Студент группы № 5030102/20002

ФИО Сатунов Буучай Юрьевич

Выполнил (дата) 13.10.2023

Оглавление

[Санкт-Петербургский государственный политехнический университет 1](#_Toc150981493)

[Постановка задачи 2](#_Toc150981494)

[Описание алгоритма 2](#_Toc150981495)

[Реализация 2](#_Toc150981496)

[Код программы 2](#_Toc150981497)

[Тестирование 4](#_Toc150981498)

[Выводы 5](#_Toc150981499)

Постановка задачи

В данной работе требуется решить задачу поиска кратчайшего пути из лабиринта, заданного матрицей, где 1 обозначает стену, а 0 обозначает проход. Задача должна быть реализована на языке программирования C++, с использованием односвязного списка для хранения данных.

Описание алгоритма

Алгоритм основан на методе поиска в ширину (BFS), который идеально подходит для поиска кратчайшего пути в невзвешенном графе, таком как наш лабиринт. Мы используем односвязный список для реализации очереди, которая является ключевым компонентом алгоритма BFS.

Реализация

В начале мы определяем структуру Node для нашего односвязного списка, который будет использоваться для реализации очереди. Затем мы реализуем функции enqueue и dequeue для работы с очередью. Функция isValid используется для проверки, является ли данная точка допустимой в лабиринте. Главная функция shortestPath реализует алгоритм BFS для поиска кратчайшего пути из лабиринта. -1 выводим в том случае, если нет пути из начальной точки до любого края лабиринта. Это может произойти, если стартовая точка окружена стенами и нет доступного пути к выходу. В этом случае, расстояние до ближайшего выхода остается равным INT\_MAX, и функция shortestPath возвращает -1, указывая на то, что путь не найден.

Код программы:

#include <iostream>

#include <climits>

// Определение структуры "Node" для односвязного списка

struct Node {

int x, y; // Координаты узла в лабиринте

Node\* next; // Указатель на следующий узел в списке

};

// Функция для создания нового узла

Node\* newNode(int x, int y) {

Node\* temp = new Node;

temp->x = x;

temp->y = y;

temp->next = NULL;

return temp;

}

// Функция для добавления элемента в конец очереди

void enqueue(Node\*\* head, Node\*\* tail, int x, int y) {

Node\* temp = newNode(x, y);

if (\*head == NULL) {

\*head = \*tail = temp;

}

else {

(\*tail)->next = temp;

\*tail = temp;

}

}

// Функция для удаления элемента из начала очереди

void dequeue(Node\*\* head, Node\*\* tail) {

if (\*head == NULL) return;

Node\* temp = \*head;

\*head = (\*head)->next;

if (\*head == NULL) \*tail = NULL;

delete temp;

}

// Функция для проверки, является ли данная точка допустимой

bool isValid(int x, int y, int m, int n) {

return (x >= 0 && x < m && y >= 0 && y < n);

}

// Функция для поиска кратчайшего пути из лабиринта

int shortestPath(int\*\* maze, int sx, int sy, int m, int n) {

// Если начальная точка является стеной, возвращаем -1

if (maze[sx][sy] == 1) {

return -1;

}

Node\* head = NULL, \* tail = NULL;

int dx[] = { -1, 0, 1, 0 }; // Возможные направления движения по x

int dy[] = { 0, 1, 0, -1 }; // Возможные направления движения по y

int\*\* dist = new int\* [m]; // Матрица расстояний

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

dist[i] = new int[n];

for (int j = 0; j < n; j++)

dist[i][j] = INT\_MAX; // Инициализация матрицы расстояний

}

dist[sx][sy] = 0; // Расстояние от начальной точки до себя равно 0

enqueue(&head, &tail, sx, sy); // Добавление начальной точки в очередь

while (head != NULL) { // Пока очередь не пуста

int x = head->x, y = head->y; // Извлечение текущей точки

dequeue(&head, &tail); // Удаление текущей точки из очереди

for (int i = 0; i < 4; i++) { // Проверка всех возможных направлений

int newX = x + dx[i], newY = y + dy[i]; // Новая точка

// Если новая точка допустима и ближе, чем текущее расстояние

if (isValid(newX, newY, m, n) && maze[newX][newY] == 0 && dist[newX][newY] > dist[x][y] + 1) {

dist[newX][newY] = dist[x][y] + 1; // Обновление расстояния

enqueue(&head, &tail, newX, newY); // Добавление новой точки в очередь

}

}

}

// Поиск минимального расстояния до края лабиринта

int minDist = INT\_MAX;

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

minDist = std::min(minDist, dist[i][0]);

minDist = std::min(minDist, dist[i][n - 1]);

}

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

minDist = std::min(minDist, dist[0][i]);

minDist = std::min(minDist, dist[m - 1][i]);

}

return (minDist == INT\_MAX) ? -1 : minDist; // Возвращение минимального расстояния

}

int main() {

// Определение лабиринта

int m = 7, n = 4;

int\*\* maze = new int\* [m];

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

maze[i] = new int[n];

}

maze[0][0] = 1; maze[0][1] = 1; maze[0][2] = 1; maze[0][3] = 1;

maze[1][0] = 1; maze[1][1] = 1; maze[1][2] = 1; maze[1][3] = 0;

maze[2][0] = 1; maze[2][1] = 0; maze[2][2] = 0; maze[2][3] = 1;

maze[3][0] = 1; maze[3][1] = 0; maze[3][2] = 1; maze[3][3] = 1;

maze[4][0] = 1; maze[4][1] = 0; maze[4][2] = 1; maze[4][3] = 1;

maze[5][0] = 0; maze[5][1] = 0; maze[5][2] = 1; maze[5][3] = 1;

maze[6][0] = 0; maze[6][1] = 0; maze[6][2] = 1; maze[6][3] = 1;

// Вызов функции для поиска кратчайшего пути

int shortest\_distance = shortestPath(maze, 1, 0, m, n);

// Вывод кратчайшего пути

std::cout << "The shortest path is " << shortest\_distance << std::endl;

return 0;

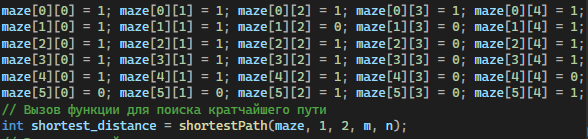
}

Тестирование

Для тестирования данной программы производился ее неоднократный запуск с вводом конкретных данных, при котором проверялось:

1. Стабильность работы программы при одинаковых входных данных;
2. Отсутствие “падений” и “зависаний”;
3. Корректное выполнение всех заявленных процедур.

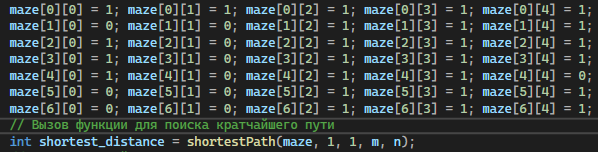
Приведем пример тестирования на скриншотах, данных ниже



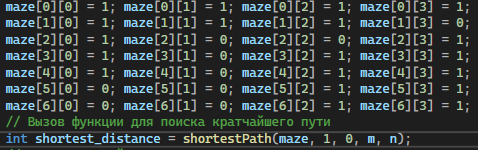




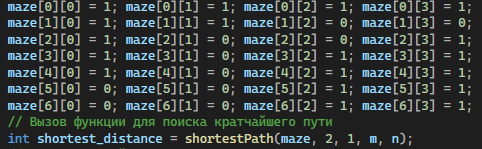














Выводы

В ходе выполнения этого задания были разработаны и протестированы функции для работы с односвязным списком, а также реализован алгоритм поиска в ширину для решения задачи поиска кратчайшего пути из лабиринта.