Министерство образования и науки Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт кибербезопасности и защиты информации

КУРСОВАЯ РАБОТА

Генератор лабиринтов

по дисциплине «Структуры данных»

Выполнили
студенты гр. 5151004/10001 Матылицкая А.А
Романова Е.Е.

<подпись>

Преподаватель
асс. преподавателя Панков И.Д.

<подпись>

«29» сентября 2023 г.

Санкт-Петербург
2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире городские метрополитены являются неотъемлемой частью транспортной инфраструктуры, обеспечивая эффективную и удобную транспортную связь для миллионов людей. Однако выбор оптимального маршрута в метро может быть непростой задачей, особенно в крупных городах, таких как Санкт-Петербург. Для решения этой проблемы разработка программного приложения, способного определить наилучший маршрут и время в пути от одной станции метро к другой, становится крайне важной.

Цель работы

Цель данного курсового проекта заключается в создании программы, которая позволит пользователю указать начальную и конечную станции в городе Санкт-Петербурге, а затем предоставит оптимальный маршрут с учетом времени в пути. Для реализации этой задачи будет использоваться язык программирования C и алгоритмы перебора с возвратом. Укрепить знания в использовании структур данных. Изучить работу с библиотекой GTK, а также реализовать и сравнить эти алгоритмы на языке программирования Си.

Поставленные задачи

Написать программу в ОС Windows, Linux и MacOS(кроссплатформенность), которая открывает окно с картой метро. На вход программы никаких данных не подаются. Есть возможность выбора начальной и конечной станции метро для поиска кратчайшего пути.

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1. Алгоритм перебора с возвратом

Алгоритм перебора с возвратом (backtracking) — это метод решения задачи путем систематического перебора всех возможных вариантов решения с последующим откатом (возвратом) к предыдущему шагу, если текущий вариант не приводит к желаемому результату. Этот метод особенно полезен при решении задач комбинаторной оптимизации, где требуется найти оптимальное решение из множества возможных вариантов.

Принцип работы алгоритма:

1. На каждом шаге алгоритм выбирает один из возможных вариантов продолжения решения задачи.

2. Если выбранный вариант не приводит к решению или нарушает какие-то условия, алгоритм откатывается на предыдущий шаг и выбирает другой вариант.

3. Процесс продолжается до тех пор, пока не будет найдено решение или исчерпаны все возможные варианты.

2. ПРАКТИЧЕСКАЯ ЧАСТЬ

2.1. Техническая часть

Блок кода на C содержит программу, которая работает с базой данных SQLite для отображения графа станций и связей между ними в графическом интерфейсе с использованием библиотеки GTK.

```c
int main(int argc, char **argv) {
    // Создание массива вершин
    Vertex *vertices[69]; // Указатель на массив вершин

    sqlite3 *db;
    char *err_msg = 0;
    sqlite3_stmt *res;

    int rc = sqlite3_open("db.db", &db);

    if (rc != SQLITE_OK) {…

    get_vertices_from_db(db, vertices);
    get_edges_from_db(db, vertices);
    get_transitions_from_db(db, vertices);

    sqlite3_finalize(res);
    sqlite3_close(db);
    gtk_init(&argc, &argv);

    // Создание окна и виджетов
    GtkWidget *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    GtkWidget *drawing_area = gtk_drawing_area_new();

    //gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    gtk_window_set_title(GTK_WINDOW(window), "Metro SPB");
    gtk_window_set_default_size(GTK_WINDOW(window), 800, 700);
    g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL);

    gtk_container_add(GTK_CONTAINER(window), drawing_area);
    g_signal_connect(G_OBJECT(drawing_area), "draw", G_CALLBACK(draw_callback), vertices);
    g_signal_connect(G_OBJECT(drawing_area), "button-press-event", G_CALLBACK(on_mouse_press), vertices);

    gtk_widget_set_events(drawing_area, gtk_widget_get_events(drawing_area) | GDK_BUTTON_PRESS_MASK);

    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);


    gtk_widget_show_all(window);
    gtk_main();

    return 0;
}
```

Рисунок 1 — Функция main

1. В начале программы создаются массивы вершин.

2. Затем открывается база данных SQLite "db.db" и выполняются SQL-запросы для извлечения информации о станциях, временах и переходах между станциями.

3. Полученные данные используются для создания вершин и связей в графе.

4. После этого инициализируется графическое окружение GTK, создается окно с названием "Metro SPB" и устанавливается размер.

5. На окне отображается область рисования (drawing_area), к которой привязаны функции обработки событий рисования и щелчков мыши по вершинам.

6. Наконец, программа ожидает событий пользовательского взаимодействия, таких как щелчки мыши или закрытие программы.

2.1.1. Структура Vertex

Структура Vertex отвечает за данные вершины графа. Заголовок файла имеет поля, представленные в таблице 1

Таблица 1 — Поля структуры Vertex

Поле структуры	Поле структуры
index;	Индекс вершины
name;	Название станции метро
time_on;	Время спуска на станции
time_exit;	Время подъёма на станции
color;	Цвет вершины
x, y:	Координаты вершины в графическом интерфейсе
size;	Размер вершины в графическом интерфейсе
edges;	Список станций в которые можно попасть из текущей с помощью поезда

transitions;	Список станций в которые можно попасть из текущей с помощью перехода
num_edges;	Кол-во станций в которые можно попасть из текущей с помощью поезда
num_transitions;	Кол-во станций в которые можно попасть из текущей с помощью перехода

2.1.2. Структура Edge

Структура Edge хранит данные о связях вершин.В дальнейшем он используется для установление связей между вершинами. Заголовок файла имеет поля, представленные в таблице 2.

Таблица 2 — Поля структуры Edge

Поле структуры	Поле структуры
from;	Из этой станция
to;	В эту станцию
time;	Время в пути

2.1.3. Структура Transition

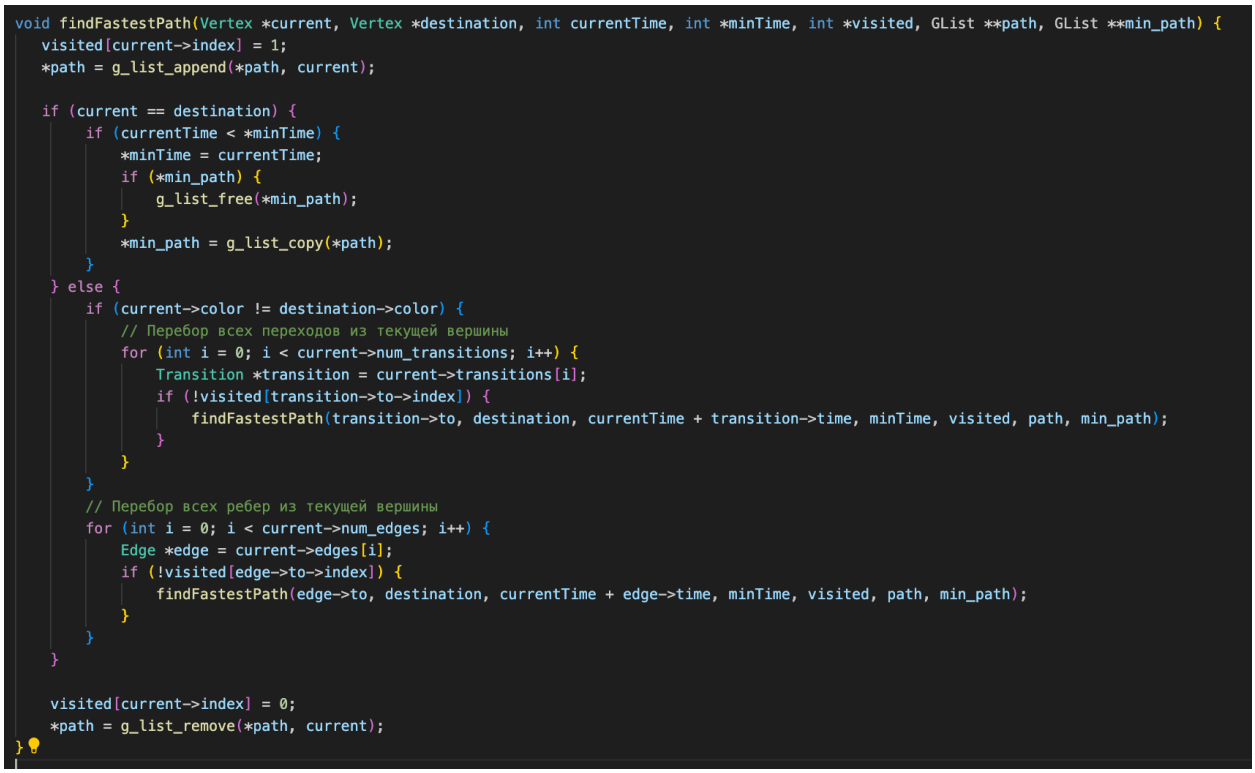
Структура Transition хранит данные о переходах на станции в дальнейшем он используется для установление переходов между вершинами. Заголовок файла имеет поля, представленные в таблице 3.

Таблица 3 — Поля структуры Transition

Поле структуры	Поле структуры
from;	Из этой станция
to;	В эту станцию

time;	Время перехода

2.1.4.Реализация алгоритма перебора с возвратом



```
void findFastestPath(Vertex *current, Vertex *destination, int currentTime, int *minTime, int *visited, GList **path, GList **min_path) {
    visited[current->index] = 1;
    *path = g_list_append(*path, current);

    if (current == destination) {
        if (currentTime < *minTime) {
            *minTime = currentTime;
            if (*min_path) {
                g_list_free(*min_path);
            }
            *min_path = g_list_copy(*path);
        }
    } else {
        if (current->color != destination->color) {
            // Перебор всех переходов из текущей вершины
            for (int i = 0; i < current->num_transitions; i++) {
                Transition *transition = current->transitions[i];
                if (!visited[transition->to->index]) {
                    findFastestPath(transition->to, destination, currentTime + transition->time, minTime, visited, path, min_path);
                }
            }
        }
        // Перебор всех ребер из текущей вершины
        for (int i = 0; i < current->num_edges; i++) {
            Edge *edge = current->edges[i];
            if (!visited[edge->to->index]) {
                findFastestPath(edge->to, destination, currentTime + edge->time, minTime, visited, path, min_path);
            }
        }
    }

    visited[current->index] = 0;
    *path = g_list_remove(*path, current);
}
```

Рисунок 2 — Реализация алгоритма перебора с возвратом

Данный алгоритм представляет собой рекурсивную функцию findFastestPath, которая использует метод перебора с возвратом для нахождения самого быстрого пути от вершины current до вершины destination в графе.

Описание работы алгоритма:

1. Функция принимает указатели на текущую вершину current, целевую вершину destination, текущее время currentTime, минимальное время minTime, массив посещенных вершин visited, указатель на список пути path и указатель на самый быстрый найденный путь min_path.

2. Помечаем текущую вершину как посещенную и добавляет ее в список пути.

3. Если текущая вершина равна целевой вершине, то проверяется, является ли текущее время быстрее, чем минимальное время найденного пути. Если да, то обновляется минимальное время и копируется текущий путь в min_path.

4. Если текущая вершина не является целевой, а ее цвет отличается от цвета целевой вершины, то происходит перебор всех переходов из текущей вершины. Для каждого перехода, который ведет к непосещенной вершине, рекурсивно вызывается функция findFastestPath для этой вершины с учетом времени прохождения перехода.

5. После этого происходит перебор всех ребер из текущей вершины. Для каждого ребра, которое ведет к непосещенной вершине, рекурсивно вызывается функция findFastestPath для этой вершины с учетом времени прохождения ребра.

6. После завершения всех возможных путей из текущей вершины, текущая вершина помечается как непосещенная и удаляется из списка пути.

7. Алгоритм продолжает свою работу, пытаясь найти все возможные пути от текущей вершины к целевой с учетом минимального времени прохождения.

2.2. Графическая часть

2.2.1.Работа с GTK

Для создания графической части программы на C использовалась библиотека GTK (GIMP Toolkit). GTK — это кроссплатформенная библиотека для создания графического интерфейса пользователя. Создается окно верхнего уровня с помощью функции gtk_window_new(GTK_WINDOW_TOPLEVEL) и область рисования с помощью функции gtk_drawing_area_new().

```
// Создание окна и виджетов
GtkWidget *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
GtkWidget *drawing_area = gtk_drawing_area_new();
```

Рисунок 3 — Создание окна

Далее устанавливаются различные параметры окна, такие как заголовок окна, размер по умолчанию, позиция на экране. Функция gtk_container_add() используется для добавления области рисования в окно. С помощью функции g_signal_connect() устанавливаются обработчики событий, такие как "draw" для отрисовки содержимого области рисования и "button-press-event" для обработки событий нажатия кнопки мыши.

```
gtk_window_set_title(GTK_WINDOW(window), "Metro SPB");
gtk_window_set_default_size(GTK_WINDOW(window), 800, 700);
g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL);

gtk_container_add(GTK_CONTAINER(window), drawing_area);
g_signal_connect(G_OBJECT(drawing_area), "draw", G_CALLBACK(draw_callback), vertices);
g_signal_connect(G_OBJECT(drawing_area), "button-press-event", G_CALLBACK(on_mouse_press), vertices);
```

Рисунок 4 — Параметры окна

Также устанавливаются определенные события для области рисования с помощью gtk_widget_set_events(), чтобы обеспечить правильную обработку событий нажатия кнопки мыши. Наконец, окно и все его дочерние виджеты отображаются на экране с помощью gtk_widget_show_all() и запускается основной цикл GTK с помощью gtk_main().

```
gtk_widget_set_events(drawing_area, gtk_widget_get_events(drawing_area) | GDK_BUTTON_PRESS_MASK);

gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);


gtk_widget_show_all(window);
gtk_main();
```

Рисунок 5 — Запуска окна

```
static gboolean on_mouse_press(GtkWidget *widget, GdkEventButton *event, gpointer data) {
    if (event->button == GDK_BUTTON_PRIMARY) {
        Vertex **vertices = (Vertex **)data;

        // Находим все вершины в радиусе клика
        for (int i = 0; i < 69; i++) {
            Vertex *v = vertices[i];
            double distance = sqrt(pow(v->x - event->x, 2) + pow(v->y - event->y, 2));
            if (distance < 10) {
                if (selected_vertex_start == NULL) {
                    selected_vertex_start = v;
                    selected_vertex_start->size = 15; // Увеличиваем размер выбранной вершины
                    break;
                }else if(v->size == 15) {
                    continue;
                } else {
                    if (is_transition(selected_vertex_start, v)) {
                        //selected_vertex_start->size = 10;
                        selected_vertex_start = v;
                        selected_vertex_start->size = 15; // Увеличиваем размер выбранной вершины
                    } else {
                        selected_vertex_end = v;
                        selected_vertex_end->size = 15;
                    }
                    break;
                }
            }
        }
        gtk_widget_queue_draw(widget);  // Перерисовка виджета
        if(selected_vertex_start != NULL && selected_vertex_end != NULL) {
            find_path(selected_vertex_start, selected_vertex_end);
            printf("Start: %s, End: %s\n", selected_vertex_start->name, selected_vertex_end->name);
            for (int i=0; i<selected_vertex_start->num_transitions; i++) {
                Transition *t = selected_vertex_start->transitions[i];
                t->to->size = 10;
                t->from->size = 10;
            }
            selected_vertex_start = NULL;
            selected_vertex_end = NULL;
        }
    }
    return true;
}
```

Рисунок 6 – Обработка сообщений при нажатии мышки

```c
static gboolean draw_callback(GtkWidget *widget, cairo_t *cr, gpointer data) {
    Vertex **vertices = (Vertex **)data;

    int *row = (int *)malloc(sizeof(int) * 80);

    for (int i = 0; i < 80; i++) {
        row[i] = 0;
    }

    for (int i = 0; i < 69; i++) {
        Vertex *v = vertices[i];
        int num_vertices = v->num_transitions;

        if (num_vertices > 0 && row[v->index] != 1) {
            double angle_step = 2 * G_PI / (num_vertices+1);
            double current_angle = 1.5 * G_PI;
            int n = 1;
            if (num_vertices == 2){…

            for (int j = 0; j < num_vertices; j++) {
                Transition *t = v->transitions[j];
                Vertex *v1 = t->to;
                Vertex *v2 = t->from;
                if (j > 0) {
                    draw_vertices(cr, v1, current_angle, current_angle + angle_step);
                    cairo_move_to(cr, v->x + n*15, v->y + 4 * (j+4));
                    cairo_show_text(cr, v1->name);
                    row[v1->index] = 1;
                } else {…
                current_angle += angle_step;
            }
        } else if (row[v->index] != 1) {
            draw_vertices(cr, v, 0, 2 * G_PI);
            row[v->index] = 1;
            cairo_move_to(cr, v->x + 15, v->y + 5);
            cairo_show_text(cr, v->name);
        }

        for (int j = 0; j < v->num_edges; j++) {
            Edge *e = v->edges[j];
            Vertex *adj_to = e->to;
            gdk_cairo_set_source_rgba(cr, &v->color);
            cairo_move_to(cr, v->x, v->y);
            cairo_line_to(cr, adj_to->x, adj_to->y);
            cairo_stroke(cr);
        }
    }
    return TRUE;
}
```

Рисунок 7 – Отрисовка станций метро и связий между ними

2.2.2.Работа с всплывающее окно

Всплывающее окно нужно для отображение пути от одной станции до другой.

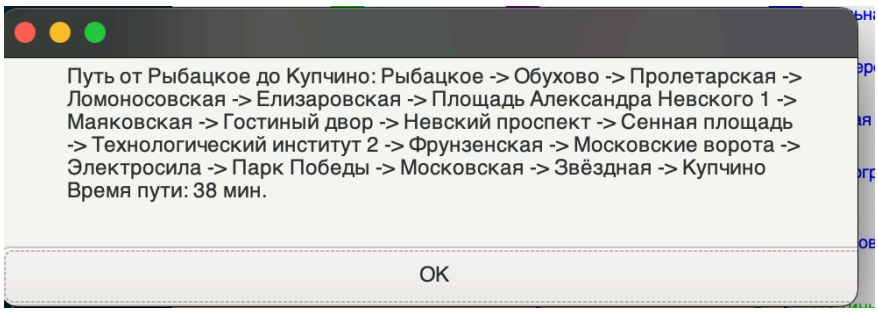


Рисунок 8 – Пример всплывающее окна

Окно появляется после того как пользователь выбрал начальную и конечную станцию. Вызывается функция find_path в которой ищется быстрый путь от одной станции к другой с помощью алгоритма перебора с возвратом, после работы алгоритмы создаётся всплывающие окно с отображением пути и времени.

```c
void find_path(Vertex *start, Vertex *end) {
    GList *path = NULL, *min_path = NULL;
    int min_time = MAX_INT;
    int *visited = (int *)calloc(69, sizeof(int));

    findFastestPath(start, end, 0, &min_time, visited, &path, &min_path);
    free(visited);

    // Вывод результата в GUI
    if (min_time != MAX_INT) {
        GString *path_str = g_string_new("");
        for (GList *l = min_path; l != NULL; l = l->next) {
            Vertex *v = l->data;
            if (l->next != NULL)
                g_string_append_printf(path_str, "%s -> ", v->name);
            else
                g_string_append_printf(path_str, "%s", v->name);
        }

        GtkWidget *dialog = gtk_message_dialog_new(NULL, GTK_DIALOG_DESTROY_WITH_PARENT,
                                                   GTK_MESSAGE_INFO, GTK_BUTTONS_OK,
                                                   "Путь от %s до %s: %s \nВремя пути: %d мин.",
                                                   start->name, end->name, path_str->str, min_time);
        gtk_dialog_run(GTK_DIALOG(dialog));
        gtk_widget_destroy(dialog);
    } else {
        GtkWidget *dialog = gtk_message_dialog_new(NULL, GTK_DIALOG_DESTROY_WITH_PARENT,
                                                   GTK_MESSAGE_INFO, GTK_BUTTONS_OK,
                                                   "Путь от %s до %s не найден.",
                                                   start->name, end->name);
        gtk_dialog_run(GTK_DIALOG(dialog));
        gtk_widget_destroy(dialog);
    }
    start->size = 10;
    end->size = 10;
    g_list_free(min_path);
    g_list_free(path);
}
```

Рисунок 9— Создание всплывающего окна

3. ТЕСТИРОВАНИЕ

3.1. Тест-кейсы

Таблица 4 — Тест-кейсы

№	Название этапа	Входные данные	Ожидаемый результат
1	Выбор станции	Площадь Восстание	В интерфейсе станция «Площадь восстание» выбирается
2	Выбор другой станции	Маяковская	В интерфейсе станция «Маяковская» выбирается
3	Отмена выбора	Клик по уже выбранной станции	В интерфейсе уже выбранная станция не выделятся
4	Выбор конечной станции	Сенная площадь	Появление всплывающего окна с информацией о пути

3.2. Примеры тест-кейсов

Рисунок 10— Тест №1

Рисунок 11— Тест №2

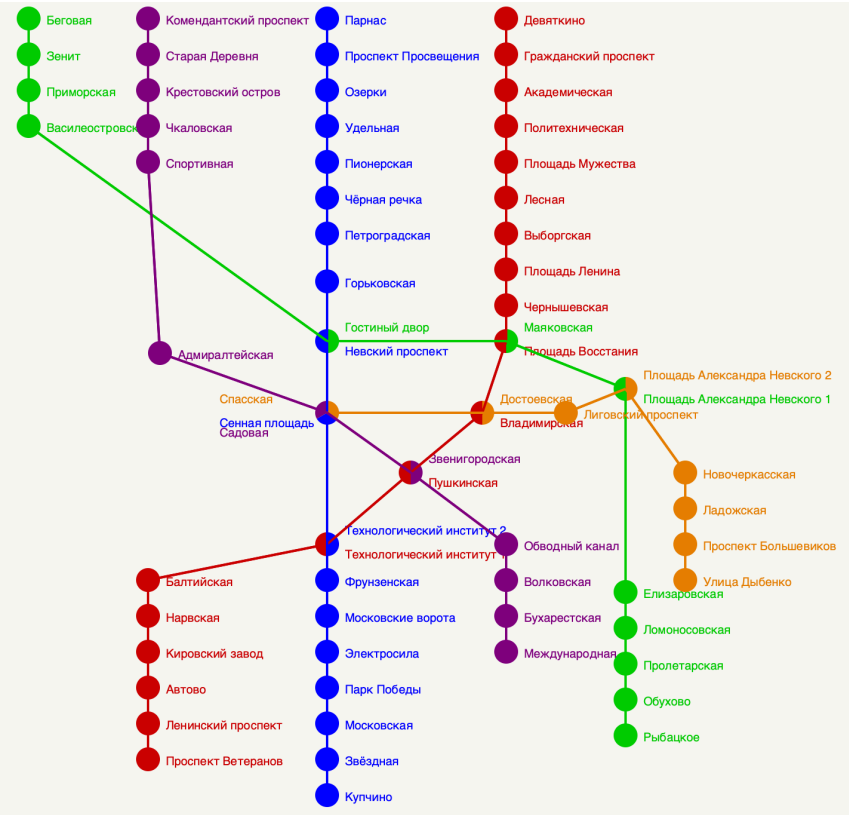


Рисунок 12— Тест №3

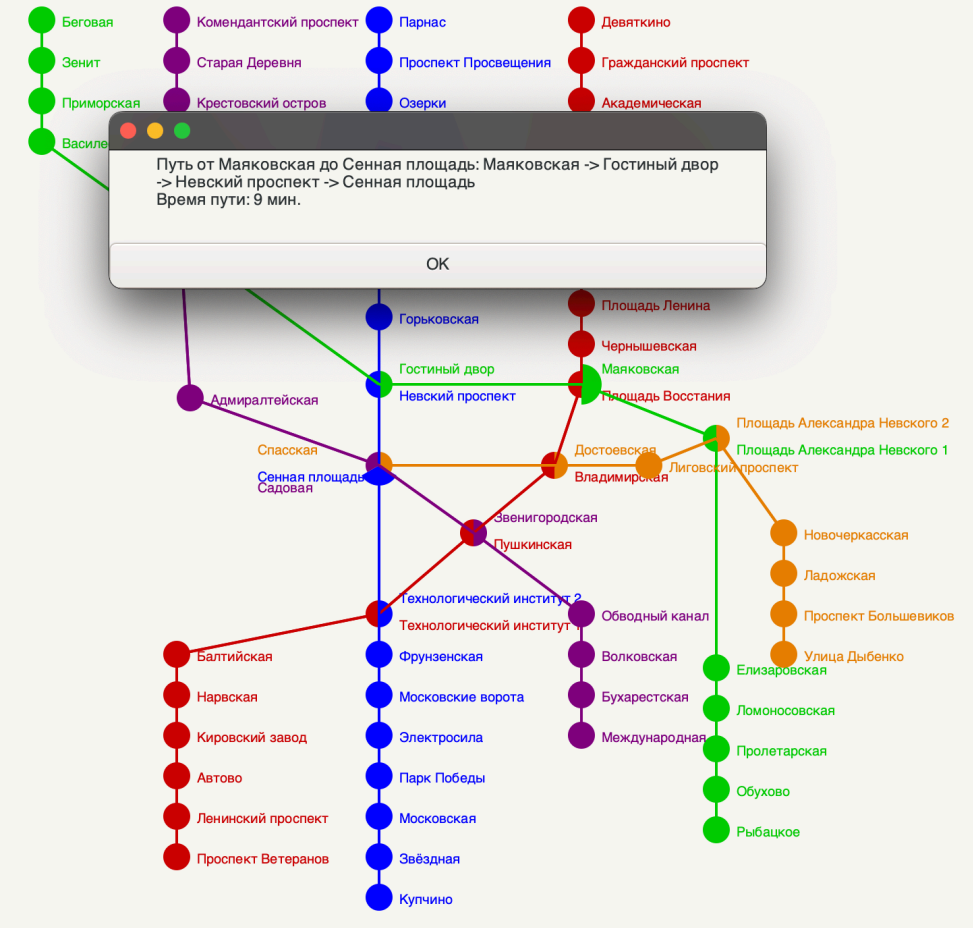


Рисунок 13— Тест №4

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проекта была разработана программа, предназначенная для определения оптимального маршрута между станциями метрополитена Санкт-Петербурга. Программа реализована на языке программирования C с использованием библиотеки GTK, что обеспечило её кроссплатформенность и возможность работы в операционных системах Windows, Linux и MacOS.

Основной задачей проекта было создание удобного и функционального приложения, которое позволяет пользователю выбирать начальную и конечную станции и находить кратчайший путь с учетом времени в пути. Для решения этой задачи были использованы алгоритмы перебора с возвратом, что позволило эффективно обрабатывать данные о маршрутах метро и предоставлять пользователю точную информацию.

Результаты тестирования программы показали, что она корректно работает на различных платформах и способна обеспечить точный расчет маршрутов в условиях реального времени. Интерфейс программы оказался интуитивно понятным и удобным для пользователя, что делает приложение привлекательным для широкой аудитории.

В процессе работы над проектом были углублены знания в области структур данных и алгоритмов на языке C, а также получен значительный опыт в разработке кроссплатформенных приложений с использованием GTK.

Таким образом, разработанное программное обеспечение успешно решает поставленные задачи, предоставляя пользователю быстрый и точный способ определения оптимального маршрута в метрополитене Санкт-Петербурга. Это делает его ценным инструментом для ежедневного использования миллионами жителей и гостей города.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Лабиринты: классификация, генерирование, поиск решений [Электронный ресурс] – Режим доступа: https://habr.com/ru/articles/445378/ (дата обращения 10.08.2023)
2. Классические алгоритмы генерации лабиринтов. Часть 1: вступление [Электронный ресурс] – Режим доступа: https://habr.com/ru/articles/320140/ (дата обращения 10.08.2023)
3. Maze Generation: Eller’s Algorithm [Электронный ресурс] – Режим доступа: https://weblog.jamisbuck.org/2010/12/29/maze-generation-eller-s-algorithm (дата обращения 14.08.2023)
4. Просто алгоритм генерации лабиринта на C/C++ [Электронный ресурс] – Режим доступа: https://ru.stackoverflow.com/questions/482663/Простой-алгоритм-генерации-лабиринта-на-c- (дата обращения 30.05.2023)
5. Генерация лабиринтов [Электронный ресурс] – Режим доступа: https://algolist.manual.ru/games/maze.php (дата обращения 6.06.2023)
6. Генерация и решение лабиринта с помощью метода поиска в глубину по графу [Электронный ресурс] – Режим доступа: https://habr.com/ru/articles/262345/ (дата обращения 4.08.2023)
7. Неприлично простая реализация неприлично простого алгоритма генерации лабиринта [Электронный ресурс] – Режим доступа: https://habr.com/ru/articles/319532/ (дата обращения 9.05.2023)
8. Maze Generator in C [Электронный ресурс] – Режим доступа: https://codereview.stackexchange.com/questions/277817/ (дата обращения 19.07.2023)
9. OpenGL Particle System Maze Generation and Gradient Coloring with Sidewinder Algorithm Visual [Электронный ресурс] – Режим доступа: https://www.youtube.com/watch?v=x17KtEEYqk8 (дата обращения 3.09.2023)
10. Генерация лабиринтов: Алгоритм Эллера [Электронный ресурс] – Режим доступа: https://habr.com/ru/articles/667576/ (дата обращения 23.08.2023)

ПРИЛОЖЕНИЕ А

Листинг программы «main.c»

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>
#include <math.h>
#include <limits.h>
#include <stdbool.h>
#include <cairo.h>
#include <gtk/gtk.h>
#include "header.h"
#include <glib.h>

#define MAX_INT 2147483647

typedef struct _Edge Edge;
typedef struct _Transition Transition;
typedef struct _Vertex Vertex;

Vertex *selected_vertex_start = NULL;
Vertex *selected_vertex_end = NULL;

void findFastestPath(Vertex *current, Vertex *destination, int
currentTime, int *minTime, int *visited, GList **path, GList
**min_path) {
   visited[current->index] = 1;
   *path = g_list_append(*path, current);

   if (current == destination) {
        if (currentTime < *minTime) {
            *minTime = currentTime;
            if (*min_path) {
                g_list_free(*min_path);
            }
```

```
            *min_path = g_list_copy(*path);
        }
    } else {
        // Перебор всех переходов из текущей вершины
        for (int i = 0; i < current->num_transitions; i++) {
            Transition *transition = current->transitions[i];
            if (!visited[transition->to->index]) {
                    findFastestPath(transition->to, destination,
currentTime + transition->time, minTime, visited, path,
min_path);
            }
        }
        // Перебор всех ребер из текущей вершины
        for (int i = 0; i < current->num_edges; i++) {
            Edge *edge = current->edges[i];
            if (!visited[edge->to->index]) {
                        findFastestPath(edge->to, destination,
currentTime + edge->time, minTime, visited, path, min_path);
            }
        }
    }

    visited[current->index] = 0;
    *path = g_list_remove(*path, current);
}

Vertex *create_vertex(int index, char *name, GdkRGBA color, int
x, int y, int time_on, int time_exit) {
    Vertex *v = malloc(sizeof(Vertex));
    v->index = index;
    v->name = name;
    v->time_on = time_on;
    v->time_exit = time_exit;
    v->color = color;
    v->x = x;
```

```
    v->y = y;
    v->size = 10;
    v->edges = NULL;
    v->transitions = NULL;
    v->num_edges = 0;
    v->num_transitions = 0;
    return v;
}

void add_edge(Vertex *v1, Vertex *v2, int time) {
    Edge *edge = malloc(sizeof(Edge));
    edge->from = v1;
    edge->to = v2;
    edge->time = time;

        v1->edges = realloc(v1->edges, (v1->num_edges + 1) *
sizeof(Edge *));
    v1->edges[v1->num_edges++] = edge;

    Edge *reverse_edge = malloc(sizeof(Edge));
    reverse_edge->from = v2;
    reverse_edge->to = v1;
    reverse_edge->time = time;

        v2->edges = realloc(v2->edges, (v2->num_edges + 1) *
sizeof(Edge *));
    v2->edges[v2->num_edges++] = reverse_edge;
}

void add_transition(Vertex *v1, Vertex *v2, int time) {
    Transition *transition = malloc(sizeof(Transition));
    transition->from = v1;
    transition->to = v2;
    transition->time = time;
```

```
        v1->transitions = realloc(v1->transitions, (v1->num_transitions + 1) * sizeof(Transition *));
    v1->transitions[v1->num_transitions++] = transition;

    Transition *reverse_transition = malloc(sizeof(Transition));
    reverse_transition->from = v2;
    reverse_transition->to = v1;
    reverse_transition->time = time;

        v2->transitions = realloc(v2->transitions, (v2->num_transitions + 1) * sizeof(Transition *));
    v2->transitions[v2->num_transitions++] = reverse_transition;
}

void draw_vertices(cairo_t *cr, Vertex *v, double current_angle, double angle_step) {
    gdk_cairo_set_source_rgba(cr, &v->color);
    cairo_move_to(cr, v->x, v->y);
        cairo_arc(cr, v->x, v->y, v->size, current_angle, angle_step);
    cairo_line_to(cr, v->x, v->y);
    cairo_fill(cr);
}

static gboolean draw_callback(GtkWidget *widget, cairo_t *cr, gpointer data) {
    Vertex **vertices = (Vertex **)data;

    int *row = (int *)malloc(sizeof(int) * 80);

    for (int i = 0; i < 80; i++) {
        row[i] = 0;
    }

    for (int i = 0; i < 69; i++) {
```

```
        Vertex *v = vertices[i];
        int num_vertices = v->num_transitions;

        if (num_vertices > 0 && row[v->index] != 1) {
            double angle_step = 2 * G_PI / (num_vertices+1);
            double current_angle = 1.5 * G_PI;
            int n = 1;
            if (num_vertices == 2){
                n = -6;
            }

            for (int j = 0; j < num_vertices; j++) {
                Transition *t = v->transitions[j];
                Vertex *v1 = t->to;
                Vertex *v2 = t->from;
                if (j > 0) {
                            draw_vertices(cr, v1, current_angle,
current_angle + angle_step);
                        cairo_move_to(cr, v->x + n*15, v->y + 4 *
(j+4));
                    cairo_show_text(cr, v1->name);
                    row[v1->index] = 1;
                } else {
                            draw_vertices(cr, v1, current_angle,
current_angle + angle_step);
                        cairo_move_to(cr, v->x + n*15, v->y + 4 *
(j-2));
                    cairo_show_text(cr, v1->name);
                    row[v1->index] = 1;
                    current_angle += angle_step;
                            draw_vertices(cr, v2, current_angle,
current_angle + angle_step);
                        cairo_move_to(cr, v->x + n*15, v->y + 4 *
(j+3));
                    cairo_show_text(cr, v2->name);
```

```
                    row[v2->index] = 1;
                }
                current_angle += angle_step;
            }
        } else if (row[v->index] != 1) {
            draw_vertices(cr, v, 0, 2 * G_PI);
            row[v->index] = 1;
            cairo_move_to(cr, v->x + 15, v->y + 5);
            cairo_show_text(cr, v->name);
        }

        for (int j = 0; j < v->num_edges; j++) {
            Edge *e = v->edges[j];
            Vertex *adj_to = e->to;
            gdk_cairo_set_source_rgba(cr, &v->color);
            cairo_move_to(cr, v->x, v->y);
            cairo_line_to(cr, adj_to->x, adj_to->y);
            cairo_stroke(cr);
        }
    }
    return TRUE;
}

static int is_transition(Vertex *v1, Vertex *v2) {
    for (int i = 0; i < v1->num_transitions; i++) {
        Transition *t = v1->transitions[i];
        if (t->to == v2) {
            return true;
        }
    }
    return false;
}

static gboolean on_mouse_press(GtkWidget *widget, GdkEventButton
*event, gpointer data) {
```

```
    if (event->button == GDK_BUTTON_PRIMARY) {
        Vertex **vertices = (Vertex **)data;

        // Находим все вершины в радиусе клика
        for (int i = 0; i < 69; i++) {
            Vertex *v = vertices[i];
                double distance = sqrt(pow(v->x - event->x, 2) +
pow(v->y - event->y, 2));
            if (distance < 10) {
                if (selected_vertex_start == NULL) {
                    selected_vertex_start = v;
                            selected_vertex_start->size = 15; //
Увеличиваем размер выбранной вершины
                    break;
                }else if(v->size == 15) {
                    continue;
                } else {
                     if (is_transition(selected_vertex_start, v))
{
                        //selected_vertex_start->size = 10;
                        selected_vertex_start = v;
                             selected_vertex_start->size = 15; //
Увеличиваем размер выбранной вершины
                    } else {
                        selected_vertex_end = v;
                        selected_vertex_end->size = 15;
                    }
                    break;
                }
            }
        }
        gtk_widget_queue_draw(widget);  // Перерисовка виджета
                     if(selected_vertex_start != NULL &&
selected_vertex_end != NULL) {
```

```
                                find_path(selected_vertex_start,
selected_vertex_end);
                                printf("Start:  %s,  End:  %s\n",
selected_vertex_start->name, selected_vertex_end->name);
                            for  (int  i=0;  i<selected_vertex_start-
>num_transitions; i++) {
                             Transition  *t  =  selected_vertex_start-
>transitions[i];
                    t->to->size = 10;
                    t->from->size = 10;
                }
                selected_vertex_start = NULL;
                selected_vertex_end = NULL;
            }
        }
        return true;
}

void find_path(Vertex *start, Vertex *end) {
    GList *path = NULL, *min_path = NULL;
    int min_time = MAX_INT;
    int *visited = (int *)calloc(69, sizeof(int));

       gint64  start_time  =  g_get_monotonic_time();  //  Засекаем
время перед запуском алгоритма

      findFastestPath(start,  end,  0,  &min_time,  visited,  &path,
&min_path);

     gint64 end_time = g_get_monotonic_time(); // Засекаем время
после выполнения алгоритма
    gint64 elapsed_time = end_time - start_time;

        g_print("Время  работы  алгоритма:  %lld  микросекунд\n",
elapsed_time);
```

```
    free(visited);

    // Вывод результата в GUI
    if (min_time != MAX_INT) {
        GString *path_str = g_string_new("");
        for (GList *l = min_path; l != NULL; l = l->next) {
            Vertex *v = l->data;
            if (l->next != NULL)
                    g_string_append_printf(path_str, "%s -> ", v-
>name);
            else
                g_string_append_printf(path_str, "%s", v->name);
        }

                GtkWidget *dialog = gtk_message_dialog_new(NULL,
GTK_DIALOG_DESTROY_WITH_PARENT,

GTK_MESSAGE_INFO, GTK_BUTTONS_OK,
                                                    "Путь от %s
до %s: %s \nВремя пути: %d мин.",
                                                   start->name,
end->name, path_str->str, min_time);
        gtk_dialog_run(GTK_DIALOG(dialog));
        gtk_widget_destroy(dialog);
    } else {
                GtkWidget *dialog = gtk_message_dialog_new(NULL,
GTK_DIALOG_DESTROY_WITH_PARENT,

GTK_MESSAGE_INFO, GTK_BUTTONS_OK,
                                                    "Путь от %s
до %s не найден.",
                                                   start->name,
end->name);
        gtk_dialog_run(GTK_DIALOG(dialog));
```

```
        gtk_widget_destroy(dialog);
    }
    start->size = 10;
    end->size = 10;
    g_list_free(min_path);
    g_list_free(path);
}

static void get_vertices_from_db(sqlite3 *db, Vertex **vertices)
{
    // Определение цветов
    GdkRGBA colors[] = {
        {0.8, 0.0, 0.0, 1.0},  // Красный
        {0.0, 0.0, 1.0, 1.0},  // Синий
        {0.0, 0.8, 0.0, 1.0},  // Зелёный
        {0.9, 0.5, 0.0, 1.0},  // Оранжевый
        {0.5, 0.0, 0.5, 1.0}   // Фиолетовый
    };
    sqlite3_stmt *res;
    char *sql = "SELECT * FROM Station;";

    int rc = sqlite3_prepare_v2(db, sql, -1, &res, 0);

    if (rc != SQLITE_OK) {

                fprintf(stderr, "Failed to fetch data: %s\n",
sqlite3_errmsg(db));
        sqlite3_close(db);

    }

    while ((rc = sqlite3_step(res)) == SQLITE_ROW) {
        int id = sqlite3_column_int(res, 0);
           const unsigned  char *name = sqlite3_column_text(res,
1);
```

```
        int time_on = sqlite3_column_int(res, 2);
        int time_exit = sqlite3_column_int(res, 3);
        int x = sqlite3_column_int(res, 5);
        int y = sqlite3_column_int(res, 6);
        int color_link = sqlite3_column_int(res, 4);
        if (color_link >= 1 && color_link <= 5) {
                            vertices[id-1] = create_vertex(id,
g_strdup_printf("%s", name), colors[color_link - 1], x, y,
time_on, time_exit);
        }

        //printf("ID: %d, Name: %s\n", id, name);
    }

    if (rc != SQLITE_DONE) {
                fprintf(stderr, "Error in SQL execution: %s\n",
sqlite3_errmsg(db));
    }
    sqlite3_finalize(res);
}

static void get_edges_from_db(sqlite3 *db, Vertex **vertices) {
    sqlite3_stmt *res;
    char *sql = "SELECT * FROM Times;";

    int rc = sqlite3_prepare_v2(db, sql, -1, &res, 0);

    if (rc != SQLITE_OK) {

                fprintf(stderr, "Failed to fetch data: %s\n",
sqlite3_errmsg(db));
        sqlite3_close(db);

    }
```

```
    while ((rc = sqlite3_step(res)) == SQLITE_ROW) {
        int from_station_id = sqlite3_column_int(res, 0);
        int to_station_id = sqlite3_column_int(res, 1);
        int time = sqlite3_column_int(res, 2);
        int peah_time = sqlite3_column_int(res, 3);
                    add_edge(vertices[from_station_id-1],
vertices[to_station_id-1], time);
    }

    if (rc != SQLITE_DONE) {
                fprintf(stderr, "Error in SQL execution: %s\n",
sqlite3_errmsg(db));
    }
    sqlite3_finalize(res);
}

static void get_transitions_from_db(sqlite3 *db, Vertex
**vertices) {
    sqlite3_stmt *res;
    char *sql = "SELECT * FROM Transfers;";

    int rc = sqlite3_prepare_v2(db, sql, -1, &res, 0);

    if (rc != SQLITE_OK) {

                fprintf(stderr, "Failed to fetch data: %s\n",
sqlite3_errmsg(db));
        sqlite3_close(db);

    }

    while ((rc = sqlite3_step(res)) == SQLITE_ROW) {
        int from_station_id = sqlite3_column_int(res, 0);
        int to_station_id = sqlite3_column_int(res, 1);
        int time = sqlite3_column_int(res, 2);
```

```
                    add_transition(vertices[from_station_id-1],
vertices[to_station_id-1], time);
    }

    if (rc != SQLITE_DONE) {
                fprintf(stderr, "Error in SQL execution: %s\n",
sqlite3_errmsg(db));
    }
    sqlite3_finalize(res);
}

int main(int argc, char **argv) {
    // Создание массива вершин
    Vertex *vertices[69]; // Указатель на массив вершин

    sqlite3 *db;
    char *err_msg = 0;

    int rc = sqlite3_open("db.db", &db);

    if (rc != SQLITE_OK) {

                fprintf(stderr, "Cannot open database: %s\n",
sqlite3_errmsg(db));
        sqlite3_close(db);

        return 1;
    }

    get_vertices_from_db(db, vertices);
    get_edges_from_db(db, vertices);
    get_transitions_from_db(db, vertices);

    sqlite3_close(db);
    gtk_init(&argc, &argv);
```

```
    // Создание окна и виджетов
    GtkWidget *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    GtkWidget *drawing_area = gtk_drawing_area_new();

          //gtk_window_set_position(GTK_WINDOW(window),
GTK_WIN_POS_CENTER);
    gtk_window_set_title(GTK_WINDOW(window), "Metro SPB");
    gtk_window_set_default_size(GTK_WINDOW(window), 800, 700);
             g_signal_connect(window, "destroy",
G_CALLBACK(gtk_main_quit), NULL);

    gtk_container_add(GTK_CONTAINER(window), drawing_area);
          g_signal_connect(G_OBJECT(drawing_area), "draw",
G_CALLBACK(draw_callback), vertices);
         g_signal_connect(G_OBJECT(drawing_area), "button-press-
event", G_CALLBACK(on_mouse_press), vertices);

             gtk_widget_set_events(drawing_area,
gtk_widget_get_events(drawing_area) | GDK_BUTTON_PRESS_MASK);

          gtk_window_set_position(GTK_WINDOW(window),
GTK_WIN_POS_CENTER);


    gtk_widget_show_all(window);
    gtk_main();

    return 0;
}
```