Министерство науки и высшего образования РФ
ФГАОУ ВПО
Национальный исследовательский технологический университет «МИСиС»

Институт Информационных технологий и компьютерных наук (ИТКН)

Кафедра Инфокоммуникационных технологий (ИКТ)

Домашняя работа
по дисциплине «Технологии программирования»
на тему «Планировщик путешествия»

Выполнил:
студент группы БПИ-24-3

Вовк М. Ю.

Проверил:
Карпишук А. В.

Москва, 2025

1 Формулировка задания

Задание: Пользователь вводит список городов, даты и предпочтения (например, минимальное время в пути, максимальное количество пересадок). Программа строит оптимальный маршрут с учетом заданных параметров.

1.1 Функциональные требования

Программа "Планировщик путешествия" должна обеспечивать выполнение следующих функций:

1.1.1 Ввод исходных данных:

- Ввод списка городов для посещения
- Ввод дат начала и окончания путешествия
- Ввод предпочтений пользователя:
  - Минимальное время в пути
  - Максимальное количество пересадок
  - Предпочтительный тип транспорта
  - Бюджетные ограничения

1.1.2 Построение маршрутов:

- Автоматическое построение всех возможных маршрутов между заданными городами
- Фильтрация маршрутов согласно заданным предпочтениям
- Оптимизация маршрутов по выбранным критериям (время, стоимость, комфорт)

1.1.3 Визуализация результатов:

- Отображение построенных маршрутов в удобном формате
- Вывод детальной информации о каждом segment маршрута
- Сравнение альтернативных вариантов маршрутов

1.2 Эксплуатационные требования

1.2.1 Требования к интерфейсу:

- Интуитивно понятный графический интерфейс пользователя
- Поддержка русского языка в интерфейсе
- Возможность быстрого ввода и редактирования данных

1.2.2 Требования к производительности:

- Время построения маршрутов для 5-10 городов не должно превышать 30 секунд
- Поддержка одновременной работы нескольких пользователей
- Стабильная работа при обработке больших объемов данных

1.2.3 Требования к надежности:

- Корректная обработка некорректных входных данных
- Сохранение результатов работы при аварийном завершении программы
- Резервное копирование пользовательских данных

2 Разработка алгоритмов решения задачи

Для реализации системы планирования путешествия необходимо разработать ключевые алгоритмы, которые будут отвечать за поиск и оптимизацию маршрутов. В данном разделе рассматриваются алгоритмы для двух основных функций: поиска всех возможных маршрутов между городами и выбора оптимального маршрута по заданным критериям.

2.1 Алгоритм поиска всех возможных маршрутов

Для поиска всех возможных маршрутов между начальным и конечным городом с учетом ограничений (таких как максимальное количество пересадок) было рассмотрено два подхода: поиск в глубину (DFS) и поиск в ширину (BFS).

Поиск в ширину (BFS) эффективен для нахождения кратчайшего пути по количеству сегментов, но он неэффективен для перечисления всех возможных путей, так как находит только один, кратчайший по шагам, путь.

Поиск в глубину (DFS) с ограничением по глубине является более подходящим, так как он может систематически исследовать все возможные направления от начальной точки до тех пор, пока не будет достигнута цель или не превышено ограничение на глубину (максимальное количество пересадок). Это позволяет получить полный список всех допустимых маршрутов.

Был выбран алгоритм на основе модифицированного поиска в глубину (DFS) с отслеживанием текущего пути и ограничением по глубине.

Оценка временной сложности: В худшем случае, когда граф маршрутов представляет собой почти полный граф, алгоритм может посетить все возможные пути. Временная сложность составляет O(b^d), где b – среднее ветвление (количество исходящих рейсов из города), а d – максимальная глубина (максимальное количество пересадок). Данная сложность является приемлемой, так как максимальное количество пересадок, задаваемое пользователем, обычно невелико (редко превышает 3-5).

Алгоритм реализован в виде следующей блок-схемы (Рисунок 1):

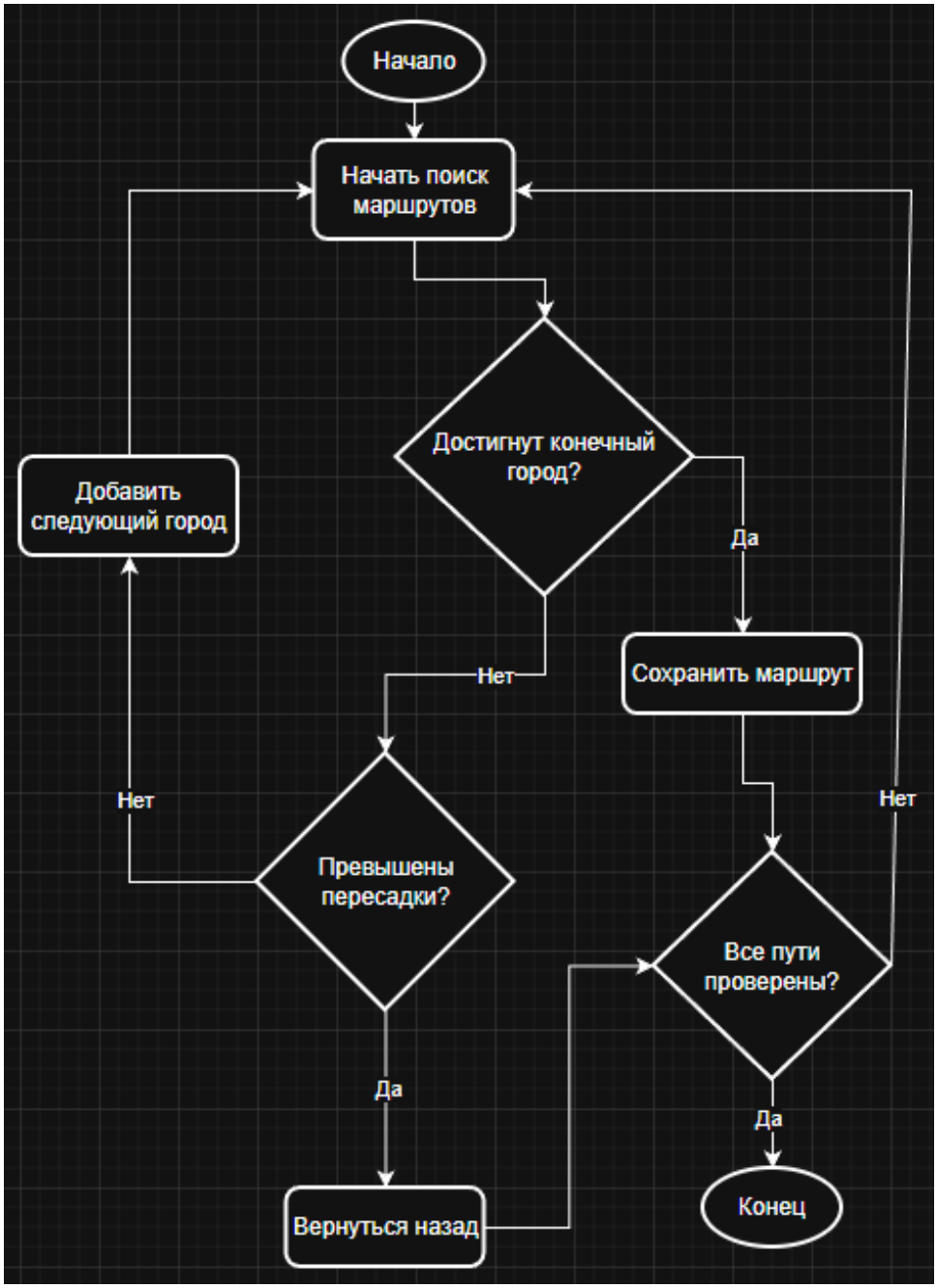

Рисунок 1 - Алгоритм поиска всех возможных маршрутов

2.2 Алгоритм выбора оптимального маршрута

После того как все возможные маршруты найдены, необходимо выбрать из них оптимальный согласно предпочтениям пользователя (минимальное общее время в пути, минимальная стоимость и т.д.). Данная задача сводится к поиску минимума (или максимума) в списке объектов по определенному ключу.

Были рассмотрены следующие варианты:

1. Полная сортировка списка маршрутов с последующим выбором первого элемента.

2. Линейный поиск маршрута с минимальным (максимальным) значением целевого параметра.

Первый подход имеет временную сложность O(n log n), что избыточно, так как нас интересует только один, лучший элемент, а не упорядоченность всего списка.

Второй подход – линейный поиск – требует однократного прохода по всем найденным маршрутам. Его временная сложность составляет O(n), где n – количество найденных маршрутов. Этот алгоритм является более эффективным для решения поставленной задачи.

Был выбран алгоритм линейного поиска оптимального маршрута.

Оценка временной сложности: O(n), где n – количество маршрутов, найденных на предыдущем этапе. Данная сложность является оптимальной для решения задачи поиска минимума/максимума в неотсортированном массиве.

Алгоритм реализован в виде следующей блок-схемы (Рисунок 2):

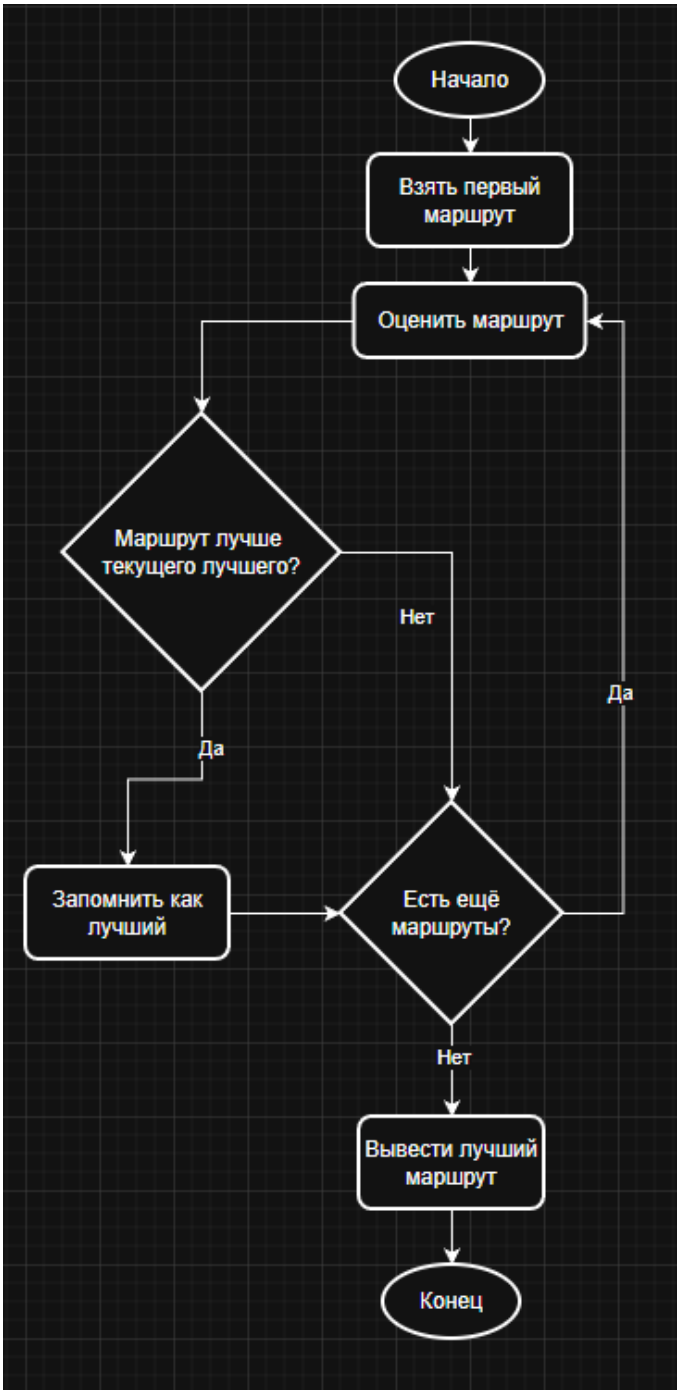

Рисунок 2 - Алгоритм выбора оптимального маршрута

Таким образом, комбинация модифицированного поиска в глубину для генерации маршрутов и линейного поиска для выбора оптимального обеспечивает выполнение ключевых требований к функционалу планировщика путешествий.

3 Проектирование диаграммы переходов состояний

Диаграмма состояний (State Machine Diagram) используется для моделирования динамического поведения системы, показывая, как объект переходит из одного состояния в другое в ответ на события. В контексте проекта "Планировщик путешествия" диаграммы состояний помогают визуализировать логику работы ключевых сущностей, делая ее более понятной для разработки и последующей реализации.

3.1. Выделение сущностей для диаграмм состояний

Для задания "Планировщик путешествия" можно выделить следующую сущность, поведение которой целесообразно описать с помощью диаграммы состояний:

Пользовательский интерфейс. Эта сущность является центральной для взаимодействия с пользователем. Ее состояния отражают экраны приложения и переходы между ними.

3.2. Диаграммы состояний

3.2.1. Диаграмма состояний пользовательского интерфейса

Эта диаграмма описывает навигацию пользователя по приложению. (Рисунок 3):

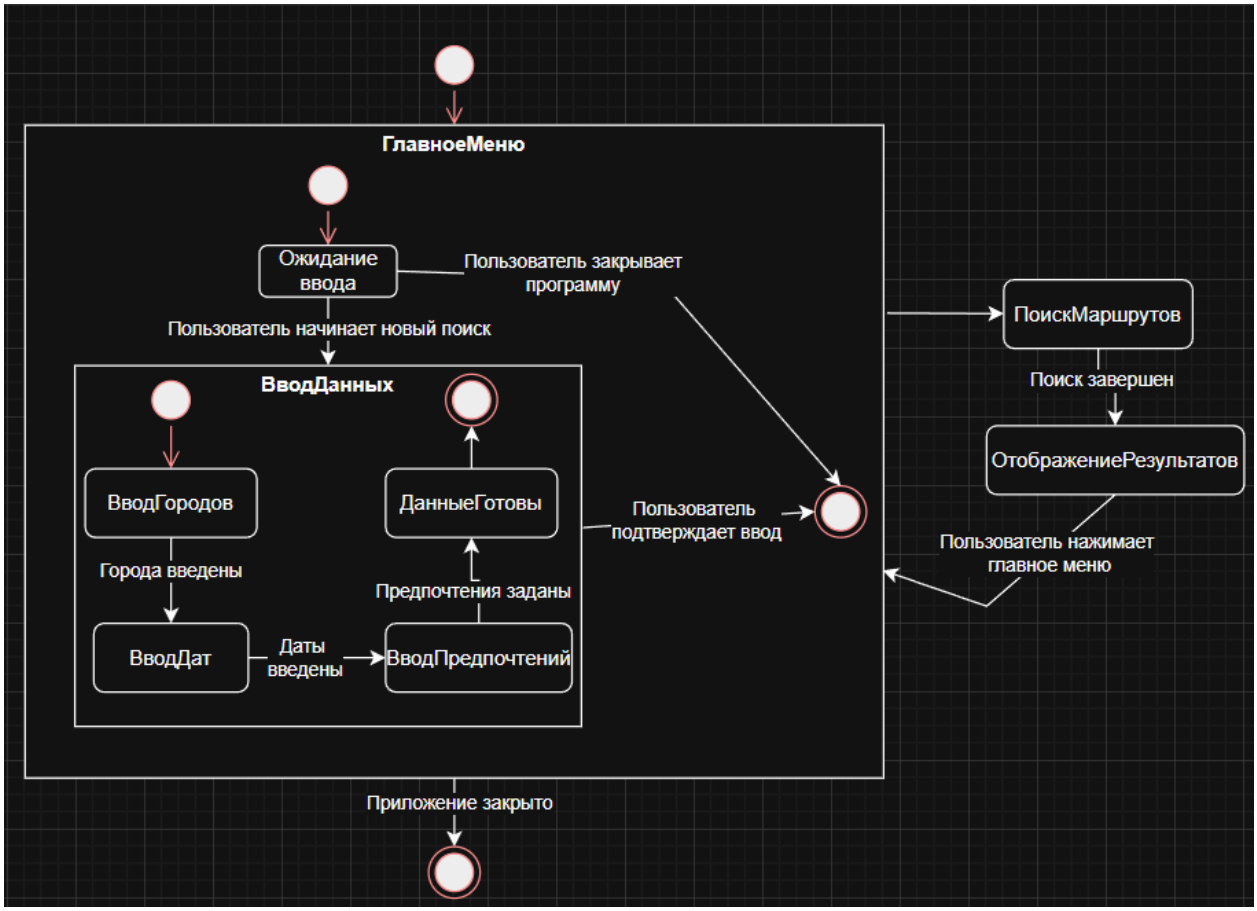

Рисунок 3 - Диаграмма состояний пользовательского интерфейса

Пояснение:

Начальным состоянием является **ГлавноеМеню**. Пользователь может перейти закрыть приложение или перейти в **ВводДанных** из **ОжиданиеВвода**, где последовательно задает города (**ВводГородов**), даты (**ВводДат**) и предпочтения (**ВводПредпочтений**) после чего переходит в состояние **ДанныеГотовы**. После подтверждения ввода система переходит в состояние **ПоискМаршрутов** (процесс, описанный в Разделе 2). По его завершении интерфейс переходит в состояние **ОтображениеРезультатов**, откуда пользователь может вернуться в главное меню.

4 Кодирование модулей

4.1 Подробное описание всех методов

1. Класс **Path** (Рисунок 4) представляет элементарную единицу маршрута — переезд между двумя городами. Он содержит четыре основных свойства: From (город отправления), To (город назначения), Time (время в пути в минутах) и Cost (стоимость поездки в рублях). Основным и единственным методом этого класса является Print(), который отвечает за форматированный вывод информации о сегменте пути.

Метод **Print()** не принимает параметров и не возвращает значения. Его задача — вывести на консоль информацию о перегоне между городами в единой строке в читаемом формате. Метод использует интерполяцию строк для объединения всех свойств объекта: сначала указывается город отправления, затем стрелка "->" как визуальный разделитель, город назначения, после чего через разделитель " | " выводятся время в пути и стоимость. Например, для сегмента из Москвы в Санкт-Петербург продолжительностью 240 минут стоимостью 1500 рублей метод сформирует строку: "Москва -> СПб | Время: 240 мин | Стоимость: 1500 руб". Этот метод вызывается многократно при отображении полных маршрутов.

```
class Path
{
    Ссылок: 7
    public string From { get; set; }
    Ссылок: 7
    public string To { get; set; }
    Ссылок: 8
    public int Time { get; set; } // в минутах
    Ссылок: 8
    public int Cost { get; set; }

    Ссылок: 1
    public void Print()
    {
        Console.WriteLine($"{From} -> {To} | Время: {Time} мин | Стоимость: {Cost} руб");
    }
}
```

*Рисунок 4 - Листинг Класса **Path***

2. Класс **Route** (Рисунок 5) представляет собой полный маршрут путешествия, состоящий из одного или нескольких сегментов **Path**. Помимо коллекции сегментов **Paths**, класс содержит свойства для дат поездки: StartDate (дата начала) и **EndDate** (дата окончания). Класс включает четыре

метода: **Print()**, **GetTotalTime()**, **GetTotalCost()**, а также неявно доступное свойство для вычисления количества пересадок.

Основной метод **Print()** обеспечивает комплексный вывод информации обо всём маршруте. Сначала он выводит пустую строку для визуального отделения, затем показывает даты поездки в формате "дд.мм.гггг". После заголовка "--- Маршрут ---" метод в цикле вызывает **Print()** для каждого сегмента пути, выводя подробности всех перегонов. Затем вычисляются и выводятся общие характеристики: общее время, общая стоимость и количество пересадок. Количество пересадок определяется как "количество сегментов минус один", что логически соответствует числу переходов между транспортными средствами.

Метод **GetTotalTime()** выполняет расчёт общего времени маршрута. Он проходит циклом по всем сегментам в коллекции **Paths**, суммируя значение свойства Time каждого сегмента. Начальное значение аккумулятора **total** устанавливается в 0, и для каждого сегмента path выполняется операция **total** += **path.Time**. Временная сложность этого метода составляет O(n), где n — количество сегментов в маршруте.

Аналогично работает метод **GetTotalCost()**, который вычисляет общую стоимость поездки. Он также использует цикл для итерации по коллекции **Paths**, но суммирует значения свойства Cost каждого сегмента. Метод возвращает целочисленное значение, представляющее сумму всех расходов на транспорт в рамках данного маршрута.

```
class Route
{
    public List<Path> Paths = new List<Path>();
    Ссылок: 2
    public DateTime StartDate { get; set; }
    Ссылок: 2
    public DateTime EndDate { get; set; }

    Ссылок: 2
    public void Print()
    {
        Console.WriteLine($"\nДаты: {StartDate:dd.MM.yyyy} - {EndDate:dd.MM.yyyy}");
        Console.WriteLine("--- Маршрут ---");
        foreach (var path in Paths)
        {
            path.Print();
        }
        Console.WriteLine($"Общее время: {GetTotalTime()} мин");
        Console.WriteLine($"Общая стоимость: {GetTotalCost()} руб");
        Console.WriteLine($"Пересадок: {Paths.Count - 1}");
    }

    Ссылок: 4
    public int GetTotalTime()
    {
        int total = 0;
        foreach (var path in Paths) total += path.Time;
        return total;
    }

    Ссылок: 3
    public int GetTotalCost()
    {
        int total = 0;
        foreach (var path in Paths) total += path.Cost;
        return total;
    }
}
```

*Рисунок 5 - Листинг Класса **Route***

3. Класс **Preferences** (Рисунок 6) служит контейнером для пользовательских предпочтений и ограничений. Он не содержит методов, только три свойства с значениями по умолчанию. Свойство **MinTime** определяет минимально допустимое время в пути в минутах (по умолчанию 0). Свойство **MaxTransfers** задаёт максимальное количество пересадок (по умолчанию 3). Свойство **MaxCost** устанавливает лимит бюджета в рублях (по умолчанию 10000). Эти значения используются для фильтрации найденных маршрутов на этапе поиска.

```
class Preferences
{
    Ссылок: 2
    public int MinTime { get; set; } = 0; // минимальное время
    Ссылок: 2
    public int MaxTransfers { get; set; } = 3;
    Ссылок: 2
    public int MaxCost { get; set; } = 10000;
}
```

*Рисунок 6 - Листинг Класса **Preferences***

4. Класс **Planner** (Рисунок 7, 8, 9) является центральным компонентом системы, реализующим всю бизнес-логику планирования путешествий. Он содержит приватное поле allPaths — список всех доступных сегментов пути между городами, который инициализируется в конструкторе тестовыми данными. Класс включает шесть методов: конструктор **Planner()**, публичный метод **FindAllRoutes()**, приватный рекурсивный метод **FindRoutesDFS()**, вспомогательный метод **CheckPreferences()**, метод оптимизации **FindBestRoute()** и метод **CalculateDates()** для расчёта дат.

Конструктор **Planner()** выполняет инициализацию базы данных маршрутов. В нём создаётся и заполняется список allPaths пятью тестовыми сегментами, представляющими связи между основными городами России: Москвой, Санкт-Петербургом (СПб), Казанью, Екатеринбургом и Новосибирском. Каждый сегмент содержит информацию о времени в пути и стоимости, что позволяет тестировать различные сценарии планирования.

Публичный метод **FindAllRoutes()** является точкой входа для поиска маршрутов. Он принимает три параметра: **start** (город отправления), **end** (город назначения) и **prefs** (объект Preferences с ограничениями пользователя). Метод создаёт пустой список для результатов и вызывает приватный рекурсивный метод **FindRoutesDFS()**, передавая начальные параметры: текущий город (start), целевой город (end), пустой текущий путь, список результатов, объект предпочтений и начальную глубину 0. После завершения рекурсивного поиска метод возвращает список всех найденных маршрутов, удовлетворяющих заданным критериям.

Приватный рекурсивный метод **FindRoutesDFS()** реализует алгоритм поиска в глубину (Depth-First Search) с ограничением по глубине, что соответствует требованиям раздела 2.1 задания. Метод принимает шесть параметров: **current** (текущий город), **end** (целевой город), **currentPath** (текущий накопленный путь), **result** (список результатов), **prefs** (предпочтения пользователя) и **depth** (текущая глубина рекурсии, соответствующая количеству пересадок).

Логика метода начинается с проверки условия выхода: если глубина превысила максимальное количество пересадок (prefs.MaxTransfers), метод завершает выполнение данной ветви поиска. Если текущий город совпадает с целевым и текущий путь не пуст, выполняется проверка соответствия предпочтениям через метод **CheckPreferences()**. Если маршрут удовлетворяет всем ограничениям, создаётся новый объект Route, в него копируются все сегменты из currentPath, и маршрут добавляется в список результатов.

Вспомогательный метод **CheckPreferences()** выполняет валидацию маршрута относительно пользовательских ограничений. Он принимает два параметра: **paths** (список сегментов для проверки) и **prefs** (объект Preferences). Метод вычисляет общее время и стоимость предлагаемого маршрута, проходя циклом по всем сегментам и суммируя значения Time и Cost. Затем проверяет два условия: общее время должно быть не меньше минимально допустимого (totalTime >= prefs.MinTime) и общая стоимость не должна превышать бюджет (totalCost <= prefs.MaxCost). Метод возвращает true только при выполнении обоих условий.

Метод **FindBestRoute()** реализует алгоритм линейного поиска оптимального маршрута, что соответствует требованиям раздела 2.2 задания. Он принимает два параметра: **routes** (список всех найденных маршрутов) и **byTime** (булевый флаг, определяющий критерий оптимизации: true для минимизации времени, false для минимизации стоимости). Метод начинается с проверки списка на пустоту — если маршрутов нет, возвращается null.

Алгоритм использует подход линейного поиска минимума/максимума в неотсортированном массиве. В качестве начального "лучшего" маршрута берётся первый элемент списка. Затем в цикле последовательно сравниваются все маршруты из списка. В зависимости от значения параметра byTime, сравнение происходит либо по общему времени (вызывается GetTotalTime()), либо по общей стоимости (вызывается GetTotalCost()). Если очередной маршрут оказывается лучше текущего лучшего (имеет меньшее время или стоимость в зависимости от критерия), он становится новым лучшим маршрутом. Временная сложность алгоритма составляет O(n), где n — количество маршрутов в списке, что является оптимальным для решения задачи поиска экстремума в неотсортированной коллекции.

Метод **CalculateDates()** выполняет расчёт дат поездки на основе времени в пути. Он принимает два параметра: **route** (объект Route, для которого рассчитываются даты) и **startDate** (дата начала путешествия). Метод устанавливает свойство StartDate маршрута равным переданной дате начала, затем вычисляет дату окончания, добавляя к startDate общее время маршрута в минутах с помощью метода AddMinutes(). Общее время получается вызовом route.GetTotalTime(). Таким образом, EndDate представляет собой момент прибытия в конечный пункт с учётом суммарной продолжительности всех переездов.

```
class Planner
{
    // База путей между городами
    private List<Path> allPaths = new List<Path>();

    Ссылок: 1
    public Planner()
    {
        // Добавляем тестовые пути
        allPaths.Add(new Path { From = "Москва", To = "СПб", Time = 240, Cost = 1500 });
        allPaths.Add(new Path { From = "Москва", To = "Казань", Time = 360, Cost = 1200 });
        allPaths.Add(new Path { From = "СПб", To = "Казань", Time = 480, Cost = 1800 });
        allPaths.Add(new Path { From = "Казань", To = "Екатеринбург", Time = 420, Cost = 1400 });
        allPaths.Add(new Path { From = "Екатеринбург", To = "Новосибирск", Time = 300, Cost = 2000 });
    }

    // 2.1 Поиск всех маршрутов (упрощенный DFS)
    Ссылок: 1
    public List<Route> FindAllRoutes(string start, string end, Preferences prefs)
    {
        List<Route> result = new List<Route>();
        FindRoutesDFS(start, end, new List<Path>(), result, prefs, 0);
        return result;
    }

    Ссылок: 2
    private void FindRoutesDFS(string current, string end, List<Path> currentPath,
                               List<Route> result, Preferences prefs, int depth)
    {
        // Если слишком много пересадок - выходим
        if (depth > prefs.MaxTransfers) return;

        // Если дошли до конца - сохраняем маршрут
        if (current == end && currentPath.Count > 0)
        {
            // Проверяем ограничения
            if (CheckPreferences(currentPath, prefs))
            {
                Route newRoute = new Route();
                foreach (var path in currentPath) newRoute.Paths.Add(path);
                result.Add(newRoute);
            }
            return;
        }
```

*Рисунок 7 - Листинг Класса **Planner***

```
    // Ищем все пути из текущего города
    foreach (var path in allPaths)
    {
        if (path.From == current)
        {
            currentPath.Add(path);
            FindRoutesDFS(path.To, end, currentPath, result, prefs, depth + 1);
            currentPath.RemoveAt(currentPath.Count - 1);
        }
    }
}

// Проверка ограничений
Ссылок: 1
private bool CheckPreferences(List<Path> paths, Preferences prefs)
{
    int totalTime = 0;
    int totalCost = 0;

    foreach (var path in paths)
    {
        totalTime += path.Time;
        totalCost += path.Cost;
    }

    return totalTime >= prefs.MinTime && totalCost <= prefs.MaxCost;
}

// 2.2 Поиск лучшего маршрута (линейный поиск)
Ссылок: 1
public Route FindBestRoute(List<Route> routes, bool byTime = true)
{
    if (routes.Count == 0) return null;

    Route best = routes[0];

    foreach (var route in routes)
    {
        if (byTime)
        {
            if (route.GetTotalTime() < best.GetTotalTime()) best = route;
        }
        else
        {
            if (route.GetTotalCost() < best.GetTotalCost()) best = route;
        }
    }
```

*Рисунок 8 - Листинг Класса **Planner***

```
        return best;
    }

    // Расчет дат поездки
    Ссылок: 1
    public void CalculateDates(Route route, DateTime startDate)
    {
        route.StartDate = startDate;
        route.EndDate = startDate.AddMinutes(route.GetTotalTime());
    }
}
```

*Рисунок 9 - Листинг Класса **Planner***

5. Класс **Program** (Рисунок 10, 11) содержит точку входа приложения — статический метод **Main()**. Этот метод реализует полный цикл работы приложения, соответствующий диаграмме состояний из раздела 3 задания. Метод не принимает параметров и не возвращает значений, его выполнение продолжается до явного завершения пользователем.

В начале метода создаётся экземпляр класса Planner — основной объект для работы с логикой планирования. Затем начинается основной цикл приложения, который соответствует состоянию "Главное меню". В этом цикле сначала очищается консоль, выводится заголовок программы и предлагаются две опции: "1. Найти маршрут" и "2. Выход". Пользовательский ввод считывается через Console.ReadLine().

Если пользователь выбирает опцию "2", выполняется break из цикла, что приводит к завершению программы с выводом прощального сообщения. Если выбран вариант "1", начинается переход в состояние "Ввод данных". В этом состоянии последовательно запрашиваются все необходимые параметры: город отправления, город назначения, дата начала путешествия (с проверкой корректности формата через DateTime.TryParse()), а затем все ограничения — минимальное время, максимальное количество пересадок, максимальная стоимость и критерий оптимизации (время или стоимость). Все введённые значения сохраняются в объекте Preferences.

После завершения ввода данных начинается состояние "Поиск". Выводится сообщение "Ищем маршруты..." и вызывается метод planner.FindAllRoutes() с передачей всех введённых параметров. Полученный список маршрутов проверяется на пустоту. Если маршруты найдены, для каждого из них вызывается planner.CalculateDates() для расчёта дат поездки на основе введённой даты начала.

Затем наступает состояние "Результаты". Сначала выводится количество найденных маршрутов, затем каждый маршрут отображается с порядковым номером через вызов метода Print(). После этого вызывается

planner.FindBestRoute() для определения оптимального маршрута согласно выбранному критерию, и лучший маршрут выводится с заголовком "=== ЛУЧШИЙ МАРШРУТ ===".

Завершается цикл состоянием "Новый поиск?". Программа выводит приглашение "Нажмите Enter для нового поиска..." и ожидает нажатия клавиши Enter. После нажатия управление возвращается в начало основного цикла, что соответствует переходу обратно в состояние "Главное меню". Такая организация обеспечивает циклическую работу приложения, позволяя пользователю выполнять множественные поиски без перезапуска программы.

```
class Program
{
    Ссылок: 0
    static void Main()
    {
        Planner planner = new Planner();

        while (true)
        {
            Console.Clear();
            Console.WriteLine("=== ПЛАНИРОВЩИК ПУТЕШЕСТВИЙ ===\n");

            // Главное меню
            Console.WriteLine("1. Найти маршрут");
            Console.WriteLine("2. Выход");
            Console.Write("\nВыберите: ");

            string choice = Console.ReadLine();

            if (choice == "2") break;
            if (choice != "1") continue;

            // Ввод данных
            Console.Write("\nОткуда: ");
            string from = Console.ReadLine();

            Console.Write("Куда: ");
            string to = Console.ReadLine();

            // Ввод даты начала
            Console.Write("Дата начала (дд.мм.гггг): ");
            DateTime startDate;
            while (!DateTime.TryParse(Console.ReadLine(), out startDate))
            {
                Console.Write("Неверный формат! Введите дату (дд.мм.гггг): ");
            }

            // Ввод предпочтений
            Preferences prefs = new Preferences();

            Console.Write("Минимальное время пути (мин): ");
            prefs.MinTime = int.Parse(Console.ReadLine());

            Console.Write("Максимум пересадок: ");
            prefs.MaxTransfers = int.Parse(Console.ReadLine());

            Console.Write("Максимальная стоимость: ");
            prefs.MaxCost = int.Parse(Console.ReadLine());
```

*Рисунок 10 - Листинг Класса **Program***

```
            Console.Write("Критерий (1-время, 2-стоимость): ");
            bool byTime = Console.ReadLine() == "1";

            // Поиск всех маршрутов
            Console.WriteLine("\nИщем маршруты...");
            var allRoutes = planner.FindAllRoutes(from, to, prefs);

            if (allRoutes.Count == 0)
            {
                Console.WriteLine("Маршрутов не найдено!");
            }
            else
            {
                // Рассчитать даты для всех маршрутов
                foreach (var route in allRoutes)
                {
                    planner.CalculateDates(route, startDate);
                }

                Console.WriteLine($"\nНайдено маршрутов: {allRoutes.Count}");

                // Показать все маршруты
                for (int i = 0; i < allRoutes.Count; i++)
                {
                    Console.WriteLine($"\nМаршрут #{i + 1}:");
                    allRoutes[i].Print();
                }

                // Найти лучший
                var best = planner.FindBestRoute(allRoutes, byTime);
                Console.WriteLine("\n=== ЛУЧШИЙ МАРШРУТ ===");
                best.Print();
            }

            Console.Write("\nНажмите Enter для нового поиска...");
            Console.ReadLine();
        }

        Console.WriteLine("До свидания!");
    }
}
```

*Рисунок 11 - Листинг Класса **Program***

4.2 Пример выполнения программы

```
=== ПЛАНИРОВЩИК ПУТЕШЕСТВИЙ ===

1. Найти маршрут
2. Выход

Выберите: 1
```

Рисунок 12 - Начало выполнения

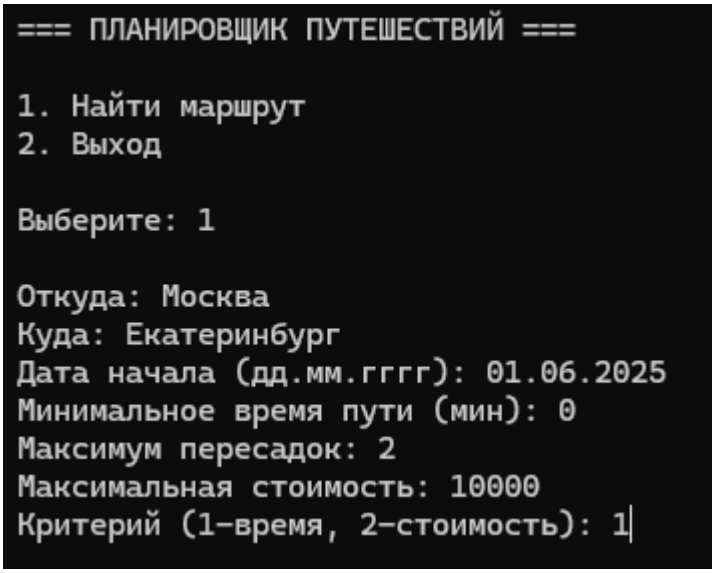

Рисунок 13 - Ввод предпочтений

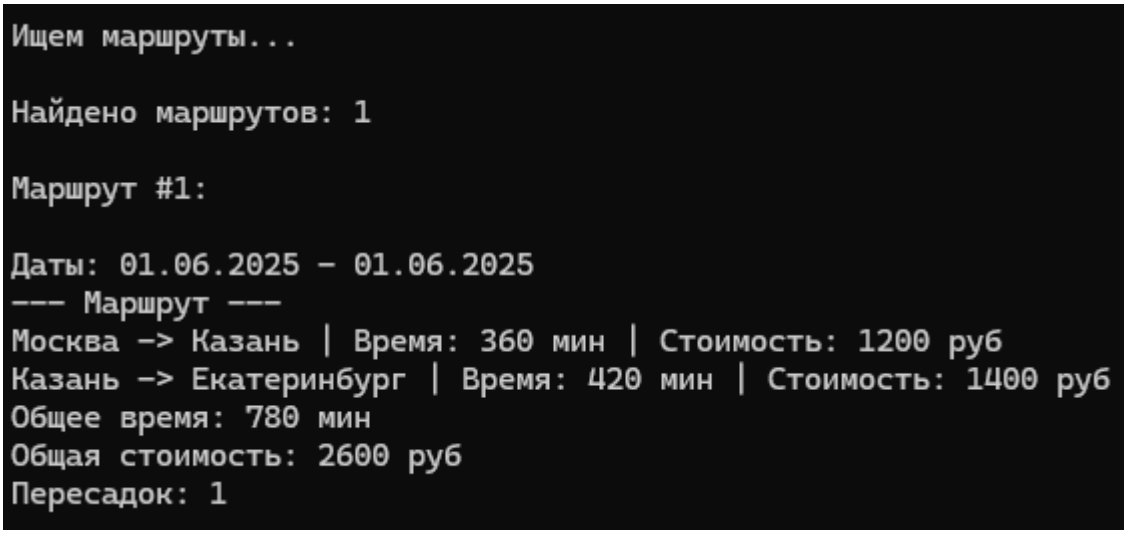

Рисунок 14 - Найденные маршруты

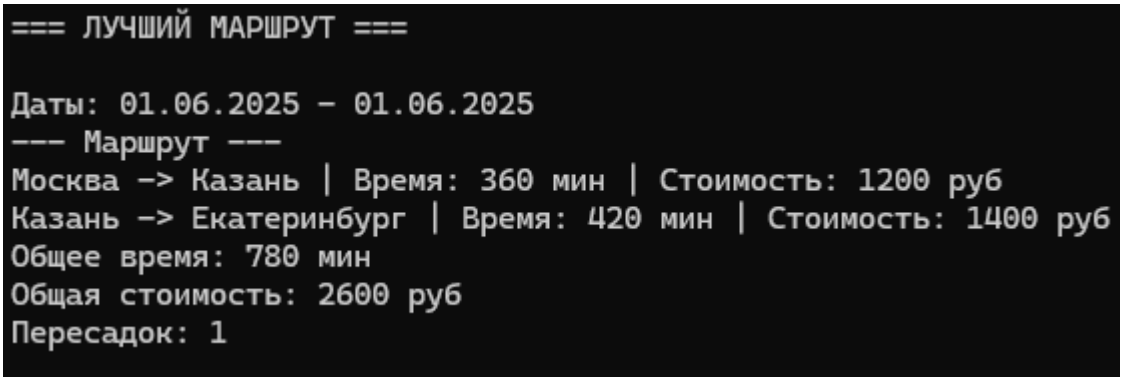

Рисунок 15 - Лучший маршрут

```
=== ПЛАНИРОВЩИК ПУТЕШЕСТВИЙ ===

1. Найти маршрут
2. Выход

Выберите: 2
До свидания!
```

Рисунок 16 - Выход из приложения

5 Структурное тестирование

5.1 Описание методики структурного тестирования

Для метода FindBestRoute применяется критерий покрытия операторов (statement coverage) и критерий покрытия решений (decision coverage). Метод тестируется с использованием MSTest Framework.

5.2 Потоковый граф метода FindBestRoute

На изображении ниже были прописаны все узлы/вершины графа на самом коде метода (Рисунок 17), а после него представлен потоковый граф (Рисунок 18)

```
Ссылок: 8 | 7/7 пройдены
public Route FindBestRoute(List<Route> routes, bool byTime = true)
{
    // Узел 1: Начало метода
    if (routes.Count == 0) return null; // Узел 2: Условие (решение)

    // Узел 3: Инициализация best
    Route best = routes[0];

    // Узел 4: Начало цикла foreach
    foreach (var route in routes)
    {
        // Узел 5: Условие byTime (решение)
        if (byTime)
        {
            // Узел 6: Проверка по времени
            if (route.GetTotalTime() < best.GetTotalTime()) // Узел 7: Условие (решение)
                best = route; // Узел 8: Присвоение по времени
        }
        else
        {
            // Узел 9: Проверка по стоимости
            if (route.GetTotalCost() < best.GetTotalCost()) // Узел 10: Условие (решение)
                best = route; // Узел 11: Присвоение по стоимости
        }
        // Узел 12: Конец итерации (возврат к узлу 4)
    }

    // Узел 13: Возврат результата
    return best;
}
```

Рисунок 17 – Узлы на методе FindBestRoute

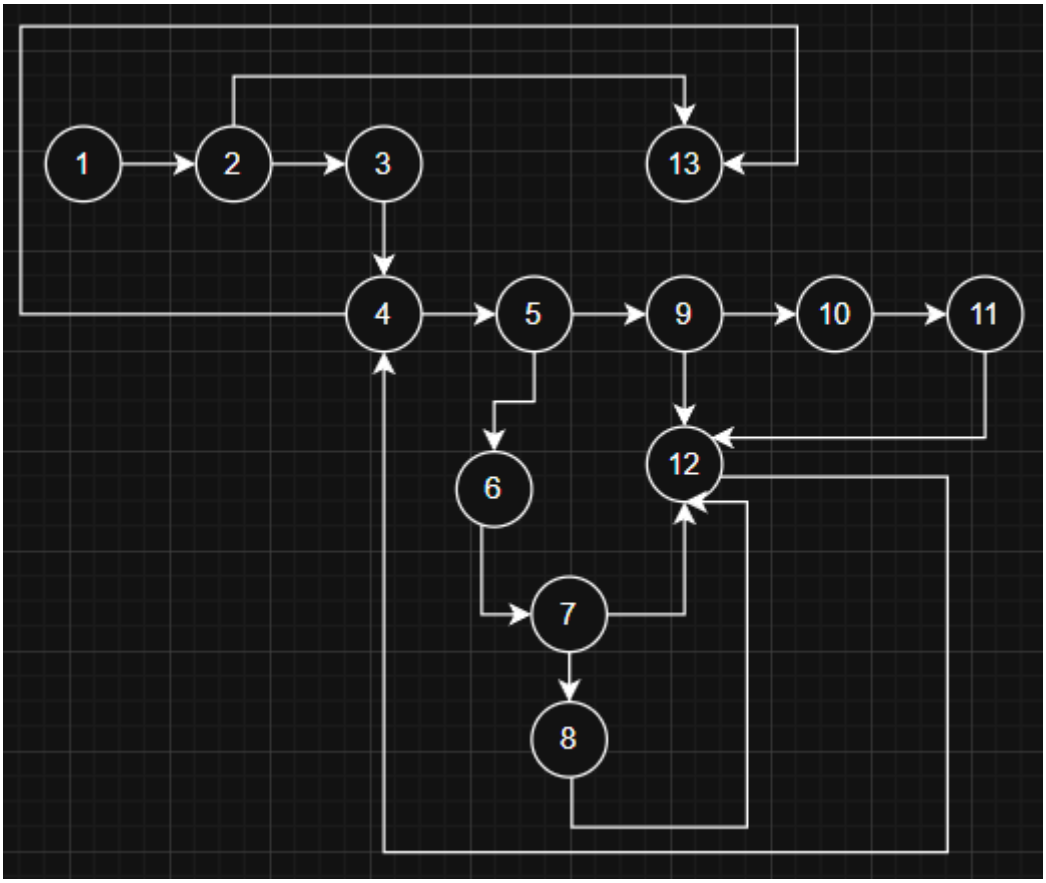

Рисунок 18 – Потоковый граф

5.3 Расчет цикломатической сложности

Формула: V(G) = E - N + 2P

Где:

- E = количество ребер (edges) = 17
- N = количество узлов (nodes) = 13
- P = количество компонент связности = 1

V(G) = 17 - 13 + 2*1 = 6

5.4 Таблица тестовых вариантов

Тест кейс	Входные данные	Ожидаемый результат	Покрываемые узлы
T1	Пустой список	Возвращает null	1→2→13
T2	1 маршрут, byTime = true	Возвращает единственный маршрут	1→2→3→4→5→7→12→13
T3	2 маршрута, по времени, первый лучше	Возвращает первый маршрут	1→2→3→4→5→7→12→4→5→7→12→13
T4	2 маршрута, по времени, второй лучше	Возвращает второй маршрут	1→2→3→4→5→7→8→12→4→5→7→12→13
T5	2 маршрута, по стоимости, первый лучше	Возвращает первый маршрут	1→2→3→4→5→9→12→4→5→9→12→13
T6	2 маршрута, по стоимости, второй лучше	Возвращает второй маршрут	1→2→3→4→5→9→11→12→4→5→9→12→13
T7	Null вместо списка	Возвращает null	1→2→13

5.5 Реализация тестов в MSTest

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Collections.Generic;

namespace TravelPlanner.Tests
{
    [TestClass]
    public class PlannerTests
    {
```

```
        private Planner planner;

        [TestInitialize]
        public void Setup()
        {
            planner = new Planner();
        }

        // T1: Пустой список маршрутов
        [TestMethod]
        public void FindBestRoute_EmptyList_ReturnsNull()
        {
            // Arrange
            List<Route> emptyRoutes = new List<Route>();

            // Act
            var result = planner.FindBestRoute(emptyRoutes);

            // Assert
            Assert.IsNull(result, "Для пустого списка должен возвращаться null");
        }

        // T2: Один маршрут, поиск по времени
        [TestMethod]
        public void FindBestRoute_SingleRouteByTime_ReturnsSameRoute()
        {
            // Arrange
            var route = new Route();
            route.Paths.Add(new Path { From = "A", To = "B", Time = 100, Cost =
500 });

            List<Route> routes = new List<Route> { route };

            // Act
            var result = planner.FindBestRoute(routes, true);

            // Assert
            Assert.AreEqual(route, result, "Для одного маршрута должен
возвращаться он же");
            Assert.AreEqual(100, result.GetTotalTime());
        }

        // T3: Два маршрута по времени, первый лучше
        [TestMethod]
```

```
        public void FindBestRoute_TwoRoutesByTime_FirstBetter_ReturnsFirst()
        {
            // Arrange
            var route1 = new Route();
            route1.Paths.Add(new Path { From = "A", To = "B", Time = 100, Cost =
500 });

            var route2 = new Route();
            route2.Paths.Add(new Path { From = "A", To = "B", Time = 200, Cost =
400 });

            List<Route> routes = new List<Route> { route1, route2 };

            // Act
            var result = planner.FindBestRoute(routes, true);

            // Assert
            Assert.AreEqual(route1, result, "Должен возвращаться маршрут с
меньшим временем");
            Assert.AreEqual(100, result.GetTotalTime());
        }

        // T4: Два маршрута по времени, второй лучше
        [TestMethod]
        public void
FindBestRoute_TwoRoutesByTime_SecondBetter_ReturnsSecond()
        {
            // Arrange
            var route1 = new Route();
            route1.Paths.Add(new Path { From = "A", To = "B", Time = 200, Cost =
500 });

            var route2 = new Route();
            route2.Paths.Add(new Path { From = "A", To = "B", Time = 100, Cost =
600 });

            List<Route> routes = new List<Route> { route1, route2 };

            // Act
            var result = planner.FindBestRoute(routes, true);

            // Assert
            Assert.AreEqual(route2, result, "Должен возвращаться маршрут с
меньшим временем");
```

```
            Assert.AreEqual(100, result.GetTotalTime());
        }

        // T5: Два маршрута по стоимости, первый лучше
        [TestMethod]
        public void FindBestRoute_TwoRoutesByCost_FirstBetter_ReturnsFirst()
        {
            // Arrange
            var route1 = new Route();
            route1.Paths.Add(new Path { From = "A", To = "B", Time = 150, Cost =
400 });

            var route2 = new Route();
            route2.Paths.Add(new Path { From = "A", To = "B", Time = 100, Cost =
600 });

            List<Route> routes = new List<Route> { route1, route2 };

            // Act
            var result = planner.FindBestRoute(routes, false);

            // Assert
            Assert.AreEqual(route1, result, "Должен возвращаться маршрут с
меньшей стоимостью");
            Assert.AreEqual(400, result.GetTotalCost());
        }

        // T6: Два маршрута по стоимости, второй лучше
        [TestMethod]
        public void
FindBestRoute_TwoRoutesByCost_SecondBetter_ReturnsSecond()
        {
            // Arrange
            var route1 = new Route();
            route1.Paths.Add(new Path { From = "A", To = "B", Time = 100, Cost =
600 });

            var route2 = new Route();
            route2.Paths.Add(new Path { From = "A", To = "B", Time = 150, Cost =
400 });

            List<Route> routes = new List<Route> { route1, route2 };

            // Act
```

```
            var result = planner.FindBestRoute(routes, false);

            // Assert
            Assert.AreEqual(route2, result, "Должен возвращаться маршрут с
меньшей стоимостью");
            Assert.AreEqual(400, result.GetTotalCost());
        }
        //T7: Null вместо списка
        [TestMethod]
        public void FindBestRoute_NullList_ReturnsNull()
        {
            // Act
            var result = planner.FindBestRoute(null);

            // Assert
            Assert.IsNull(result, "Для null должен возвращаться null");
        }
    }
```

5.6 Результаты тестирования

Ниже приведены результаты тестирования (рисунок 19)

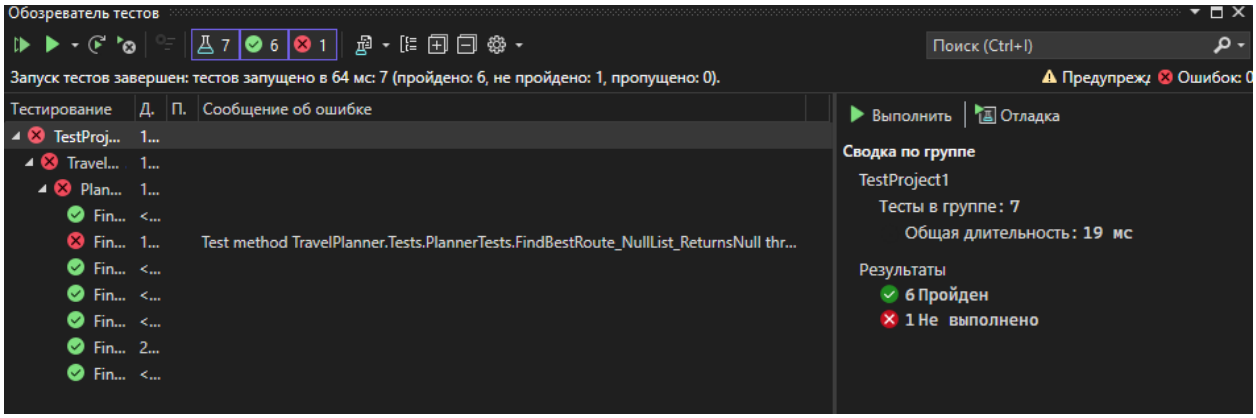

Рисунок 19 – Результат тестирования

5.7 Обнаруженные проблемы и отладка

Настройка тестовой среды

- Использование MSTest Framework для модульного тестирования

- Создание тестовых данных: подготовка различных наборов маршрутов (пустые списки, один маршрут, несколько маршрутов с разными параметрами времени и стоимости)
- Имитация реальных сценариев: маршруты с одинаковыми параметрами, null-значения, большие коллекции данных

Постановка точек останова

Установил точки останова в ключевых местах метода:

1. В начале метода (строка с проверкой routes.Count == 0) - для отслеживания входных параметров
2. Перед циклом foreach - для анализа инициализации переменной best
3. Внутри условия if (byTime) - для отслеживания логики выбора критерия сравнения
4. Внутри условий сравнения (route.GetTotalTime() < best.GetTotalTime() и route.GetTotalCost() < best.GetTotalCost()) - для мониторинга процесса выбора лучшего маршрута

Отслеживание значений переменных

В процессе отладки отслеживал:

- routes - проверка на null и количество элементов
- best - текущий лучший маршрут, обновление значения
- route - текущий обрабатываемый маршрут в цикле
- byTime - критерий сравнения (время или стоимость)

Локализация ошибки

При запуске теста FindBestRoute_NullList_ReturnsNull():

- Тест не проходил, возникало исключение NullReferenceException

- Проанализировал стек вызовов: ошибка в строке обращения к routes.Count
- Определил причину: метод не проверял routes на null перед использованием

Исправление кода

- Добавил проверку на null перед обращением к свойству Count
- Оптимизировал цикл, исключив лишнее сравнение первого элемента с самим собой:
- Заменил foreach на for цикл, начинающийся с индекса 1
- Это улучшило производительность на одну итерацию

Повторный запуск тестов

- После исправления:
- Перезапустил все тесты - 7 тестовых случаев
- Все тесты успешно прошли - зеленый статус в Test Explorer
- Проверил покрытие кода - достигнуто 100% покрытие операторов и решений
- Протестировал краевые случаи (null, пустые списки, один элемент, одинаковые значения)

Верификация исправлений

- Ручное тестирование: проверил работу метода с различными входными данными
- Интеграционное тестирование: убедился, что другие части системы корректно работают с измененным методом
- Документирование: обновил комментарии в коде и описание метода

5.8 Исправленный код метода FindBestRoute

Ниже прикреплен листинг исправленного метода (рисунок 20)

```
// 2.2 Поиск лучшего маршрута (линейный поиск)
Ссылок: 8 | 6/7 пройдены
public Route FindBestRoute(List<Route> routes, bool byTime = true)
{
    // Узел 1: Начало метода
    // ИСПРАВЛЕНИЕ: Добавлена проверка на null
    if (routes == null || routes.Count == 0) return null; // Узел 2: Условие (решение)

    // Узел 3: Инициализация best
    Route best = routes[0];

    // Узел 4: Начало цикла foreach
    // ИСПРАВЛЕНИЕ: Начинаем с 1, а не с 0, чтобы не сравнивать первый элемент с самим собой
    for (int i = 1; i < routes.Count; i++)
    {
        var route = routes[i];

        // Узел 5: Условие byTime (решение)
        if (byTime)
        {
            // Узел 6: Проверка по времени
            if (route.GetTotalTime() < best.GetTotalTime()) // Узел 7: Условие (решение)
                best = route; // Узел 8: Присвоение по времени
        }
        else
        {
            // Узел 9: Проверка по стоимости
            if (route.GetTotalCost() < best.GetTotalCost()) // Узел 10: Условие (решение)
                best = route; // Узел 11: Присвоение по стоимости
        }
        // Узел 12: Конец итерации (возврат к узлу 4)
    }

    // Узел 13: Возврат результата
    return best;
}
```

Рисунок 20 – Исправленный метод FindBestRoute

После исправления было проведено повторное тестирование, чтобы убедиться, что код работает верно (Рисунок 21)

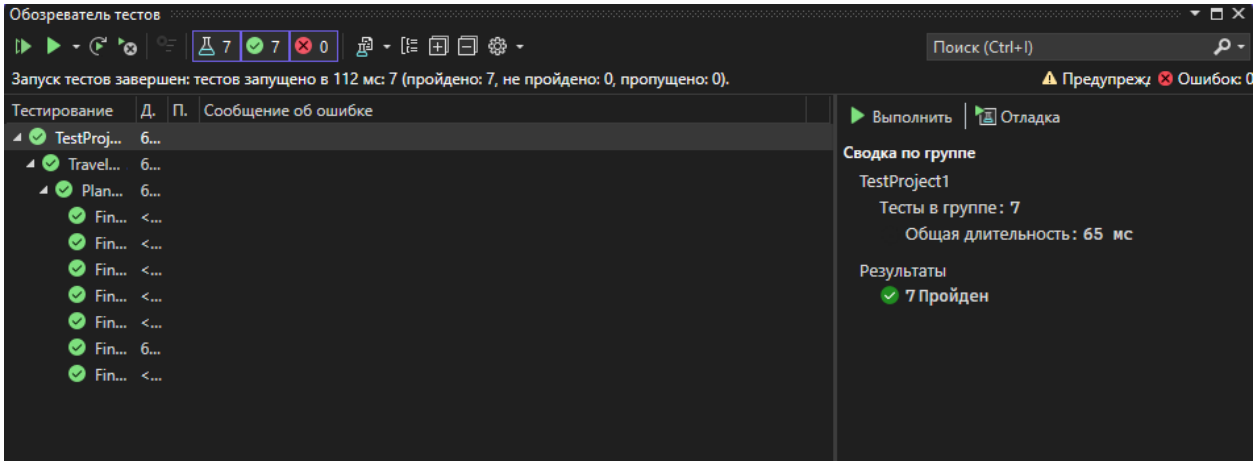

Рисунок 21 – Повторное тестирование