МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«Южный федеральный университет»

Институт математики, механики
и компьютерных наук им. И. И. Воровича

Кафедра алгебры и дискретной математики

Абдухоликзода Холис

ЗАДАЧИ НАВИГАЦИИ ПРИ НАЛИЧИИ ОГРАНИЧЕНИЙ

КУРСОВАЯ РАБОТА
по направлению подготовки
01.03.02 – Прикладная математика и информатика

Научный руководитель –
проф., д. ф.-м. н. Скороходов Владимир Александрович

________________ ____________________
оценка (рейтинг) подпись руководителя

Ростов-на-Дону – 2021

Оглавление

Постановка задачи

1. Изучить методы и алгоритмы поиска кратчайших путей на ориентированных графах.
2. Разработать метод представления карты дорог в виде графа с учётом наличия ограничений на перемещение транспортных средств.
3. Разработать алгоритм построения оптимального (по времени либо расстоянию) маршрута для транспортных средств различных видов.
4. Реализовать и протестировать разработанный алгоритм.

Научный руководитель д.ф.-м.н., проф. Скороходов В.А.

Введение

Навигация – это процесс мониторинга и управления передвижением некоторого объекта (или транспортного средства) в определённом пространстве передвижения. Эта область включает в себя несколько категорий, такие как: морская навигация, автомобильная, воздушная, космическая и т. д. Определение местоположения, скорости и ориентации в пространстве, передвигающихся объектов (или транспортных средств); прокладывание оптимального маршрута передвижения – являются основными задачами навигации. В данной работе будет рассматриваться задача о нахождение оптимального пути между двумя точками в пространстве дорожной карты (сети).

Обычно, прокладывание маршрута (англ. routing) сводится к задаче о нахождении кратчайшего пути в смоделированном графе, которая соответствует дорожной сети. Правила, по которым этот граф создаётся – много. Одним из простых способов, которое можно часто встретить, является такое правило: вершины будут представлять перекрестки, рёбра (или дуги) – сегменты дорог, а весами этих самых дуг могут быть, например, длины участки дорог, время прохождения по этим дугам, расходы и т. д. Также, существуют множества различных алгоритмов для нахождения кратчайших путей, отличающиеся, например, алгоритмической сложностью или решением разных постановок этой задачи (нахождение кратчайшего пути от одной из вершин графа до всех остальных, между всеми вершинами и т. д.).

Однако, то простое правило представление дорожной сети, которое упоминалось, не предусматривает наличие других атрибутов дуг и вершин сети. Такими атрибутами могут быть, например, ограничения на поворот и движение в перекрестках, ограничения в зависимости от типа передвигаемого транспорта или веса, доступность прохождения по участке дороги только в определенные времена суток и дней, принадлежность к определенному классу дороги и т. д. Важно учитывать такие ограничения при создании графа или при работе алгоритмов нахождения кратчайшего пути, поэтому тот упомянутый простой способ малопригоден при приложении с реалистичными условиями.

С целью избавления от таких сложностей можно попробовать выбрать другие методы и правила представления дорожной сети в виде графа. Одними из таких методов могут быть следующие: вершинами графа будут сегменты дорог, а дугами между двумя вершинами – разрешенный поворот из участки дороги в другой (т. е. выворачивание первого упомянутого способа обратно, англ. edge - based graph, line graph); создание графа – развертки по определенным способам учитывая ограничения или же комбинация нескольких методов.

Знакомство и реализация некоторых таких известных методов представления графа, попытка придумывания нового метода и модификация известных алгоритмов, находящих кратчайшие пути, являются целью этой работы.

1. Основные сведения из теории графов

Сначала, приведём основные необходимые понятия и определения из теории графов, которые потребуются для дальнейшего корректного изложения. [1], [2], [6]

Определение 1.1. *Ориентированным графом (далее будем называть просто графом, если не оговорено иное) будем называть тройку $G(V, E, f)$, где $V(\neq \emptyset)$ – множество, называемое множеством вершин графа, E – множество (возможно и пустое), называемое множеством дуг, f –отображение, действующее из E в $V \times V$, называемое отображением инцидентности.*

Определение 1.2. *Вершины s и t графа G называются смежными, если найдётся такая дуга e, что либо $f(e) = (s, t)$, либо $f(e) = (t, s)$. В таком случае также говорят, что дуга e инцидентна вершинам s и t.*

Также введём в рассмотрение следующие отображения:

$$source((s, t)) := s \qquad \text{и} \qquad target((s, t)) := t$$

Для каждой вершины графа G определим множество входящих и выходящих дуг:

$$out(s) := \{e \in E : source(e) = s\}$$

$$in(s) := \{e \in E : target(e) = s\}$$

Определение 1.3. *Путь – это список дуг $\mathbf{P} = \langle e_1, e_2, \ldots, e_p \rangle$ и такое что:*

$$source(e_{i+1}) = target(e_i), \qquad \forall i = 1, \ldots, p - 1$$

Определение 1.4. *Взвешенным графом будем называть четверку $G(V, E, f, w)$, где $G(V, E, f)$ – это граф, а $w : E \to [0; +\infty]$.*

При этом w называется весовым отображением и если $e \in E$, то $w(e)$ называется весом дуги e. Длиной пути будет следующая величина:

$$w(\mathbf{P}) = \sum_{e_i \in \mathbf{P}} w(e_i), \qquad \forall i = 1, \ldots, p$$

Также, $source(\mathbf{P}) = source(e_1)$ и $target(\mathbf{P}) = target(e_p)$, называются начальной и конечной вершинами пути соответственно (в работе предполагается, что граф не содержит параллельных дуг). Путь можно обозначить, как:

$$\mathbf{P} = \langle v_1, v_2, \ldots, v_{p+1} \rangle, \qquad \forall v_i \in V$$

Определение 1.5. *Пусть $D_{(s,t)}$ будет множеством всех возможных путей из вершины s в t. Будем говорить, что путь $\mathbf{P} \in D_{(s,t)}$ является кратчайшим, тогда и только тогда, когда $d(\mathbf{P}) \leq d(\mathbf{P}')\ \forall \mathbf{P}' \in D_{(s,t)}$*

Определение 1.6. *Поворот – это путь, состоящий из двух дуг. Можно обозначить, как:*

$$T = (s, t, u), \qquad \forall s, t, u \in V$$

или

$$T = (e, u), \qquad \forall e, u \in E : source(u) = target(e)$$

В этой работе предполагается, что все рассматриваемые графы являются ориентированными, а также предполагается, что граф – статический, т. е. в зависимости от времени, отношение вершин, дуг или веса этих самых дуг не меняется. Если потребуется, то другие нужные определения будут приводится в ходе изложения.

2. Алгоритмы нахождения кратчайших путей

Задача о кратчайшем пути состоит в нахождении кратчайшего пути от заданной вершины $s \in V$ до заданной $t \in V$. Существует большое количество алгоритмов, находящих кратчайшие пути на графе. Некоторые из них могут использоваться для нахождения кратчайшего пути только между двумя вершинами (например, алг. Дейкстры), между одной и несколькими (например, алг. Беллмана-Форда), между всеми вершинами (например, алг. Флойда-Уоршелла) и т. д. Рассмотрим некоторые из этих алгоритмов. Так как следующие алгоритмы широко известны и являются классическими, то не будем подробно останавливаться на объяснение работы каждого алгоритма, а ограничимся только небольшими сведениями и псевдокодами.

2.1. Алгоритм Дейкстры

Находить кратчайший путь между двумя заданными вершинами графа (может находить и до всех остальных вершин). Этот алгоритм основан на приписывании вершинам временных пометок, каждая из которых имеет значение равной верхней границе длины пути от начальной вершины до текущей. С помощью некоторой итерационной процедуры значения этих временных пометок постепенно уменьшается и на каждой итерации алгоритма, ровна одна пометка становится постоянной, т. е. пометка уже будет иметь значение равной точной кратчайшей длины от начальной до текущей. [2], [3], [7], [6]

Algorithm 1 Алгоритм Дейкстры

```
procedure DIJKSTRA(G = (V, E, f, w), Vertex s, Vertex t)
                        ▷ Где G – граф, s – начальная вершина, t – конечная.
    PriorityQueue Q := ∅
    List d = ⟨∞, . . . , ∞⟩
    Ancestors = ⟨∅, . . . , ∅⟩
    d[s] = 0
    Q.append({d[s], s})
    while Q ≠ ∅ do
        Vertex u = Q.ExtractMin.Second
        if u = t then
            return d[u]
        end if
        for all Edge e ∈ out(u) do
            Vertex v := target(e)
            if d[v] > d[u] + w(e) then
                d[v] := d[u] + w(e)              ▷ Релаксация вдоль дуги
                Ancestors[v] := u
                Q.append({d[v], v})
            end if
        end for
    end while
    return ∞
end procedure
```

2.2. Алгоритм A*

Алгоритм A* является расширением алгоритма Дейкстры и имеет единственное отличие, такое как установка приоритета вершины $u \in V$ по следующей формуле:

$$f[u] = d[u] + h(u),$$

где $d[u]$ – длина (или стоимость) пути от начальной вершины до u, $h[u]$ – так называемая эвристическая оценка длины пути от текущей вершины u до конечной t, а f – эвристическая функция и её значение равна $f[u]$ для этой вершины. Функция $h(x)$ должна быть **допустимой эвристической оценкой**, то есть она не должна переоценивать расстояние до целевой вершины. При каждой итерации этот алгоритм выбирает из очереди вершину с наименьшим значением f и получается так, что при использовании эвристической оценки выбирается вершина, которая находится наиболее близко к конечной вершине. [5], [8]

В этой работе, в качестве функции эвристической оценки была выбрана функция гаверсинуса (англ. Haversine formula):

$$h(\varphi_1, \varphi_2, \lambda_1, \lambda_2) :=$$
$$d = 2 \cdot R \cdot \arcsin \sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)},$$

где R – это средний радиус Земли в метрах, φ_1 и φ_2 – широта двух географических точек p_1 и p_2, λ_1 и λ_2 – долгота точек p_1 и p_2, соответственно. Формула гаверсинуса определяет расстояние по дуге на сфере между двумя точками с учетом их широты и долготы. Эта эвристическая функция является допустимой, т. к. если $D_{(p_1,p_2)}$ будет множеством всех возможных путей между точками $p_1(\varphi_1, \lambda_1)$ и $p_2(\varphi_2, \lambda_2)$ на поверхности Земли, то:

$$h(\varphi_1, \varphi_2, \lambda_1, \lambda_2) \leq d[\mathbf{P}]\ \forall \mathbf{P} \in D_{(p_1,p_2)}$$

Таким образом, этот алгоритм находит кратчайший путь, зависящий от заданной эвристики. Приведём псевдокод:

Algorithm 2 Алгоритм A*

procedure AStar($G = (V, E, f, w)$, Vertex s, Vertex t)
  ▷ Где G – граф, s – начальная вершина, t – конечная.
  PriorityQueue $Q := \emptyset$
  List $d = \langle \infty, \ldots, \infty \rangle$
  List $f = \langle \infty, \ldots, \infty \rangle$
  Ancestors $= \langle \varnothing, \ldots, \varnothing \rangle$
  $d[s] = 0$
  $f[s] = d[s] + h(s)$
  Q.append($\{f[s], s\}$)
  while $Q \neq \emptyset$ **do**
    Vertex u =Q.ExtractMin.Second
    if $u = t$ **then**
      return $d[u]$
    end if
    for all Edge $e \in out(u)$ **do**
      Vertex $v := target(e)$
      if $d[v] > d[u] + w(e)$ **then**
        $d[v] := d[u] + w(e)$ ▷ Релаксация вдоль дуги.
        $f[v] := d[v] + h[v]$ ▷ Значение эвристической функции.
        Ancestors$[v] := u$
        Q.append($\{f[v], v\}$)
      end if
    end for
  end while
  return ∞
end procedure

2.3. Алгоритм Беллмана - Форда

Находить кратчайшие пути от одной вершины, до всех остальных и допускает дуги с отрицательным весом. Если существует цикл отрицательного веса, то кратчайшего пути до некоторых вершин может не существовать, однако, возможно изменить этот алгоритм так, чтобы он сигнализировал о наличии такого цикла. Приведём псевдокод:

Algorithm 3 Алгоритм Беллмана - Форда

```
procedure BellmanFord(G = (V, E, f, w), Vertex s)
                                    ▷ Где G – граф, s – начальная вершина.
    List d = ⟨∞, ..., ∞⟩
    Ancestors = ⟨∅, ..., ∅⟩
    d[s] = 0
    for k = 1 to |V| − 1 do
        for all Edge e ∈ E do
            Vertex v1 := source(e)
            Vertex v2 := target(e)
            if d[v1] < ∞ then
                if d[v2] > d[v1] + w(e) then
                    d[v2] := d[v1] + w(e)      ▷ Релаксация вдоль дуги.
                    Ancestors[v2] := v1
                end if
            end if
        end for
    end for
end procedure
```

Работа алгоритма состоит в следующем: сначала, находить кратчайшие пути, состоящие не более из одной дуги, потом, состоящие из двух дуг, и т. д. После i-той итерации внешнего цикла, будут вычислены кратчайшие пути, содержащие не более i дуг. Внешний цикл выполняется $|V| - 1$ раз, так как максимально возможное количество дуг в простом пути (т. е. все вершины которого попарно различны) равна $|V| - 1$.

Работу этого алгоритма можно ускорить таким образом: если после некоторой итерации внешнего цикла не произошло релаксации дуг, то работу алгоритма можно останавливать (с этой целью, внутри тела цикла можно добавить флаг, который будет об этом сообщать).[9]

Для нахождения кратчайших путей между всеми парами вершин графа можно использовать, например, алгоритм Флойда-Уоршелла. Но, со временной сложностью $O(|V|^3)$, этот алгоритм малопригоден для использования в реальной дорожной сети, где количество вершин исчисляется миллионами. Хотя, при наличии достаточной вычислительной мощности, вычисленные кратчайшие расстояния можно было бы использовать в качестве значения **точной эвристики** для других алгоритмов использующих эвристическую оценку.

Существуют более продвинутые алгоритмы, помимо перечисленных, такие как *Contraction Hierarchies, Highway Hierarchies, ALT (A* with Landmarks and Triangle inequality), Reach* и т. п., которые используются в известных картографических сервисах, однако в этой работе они не будут рассмотрены. [7]

3. Решение задачи

3.1. Типы ограничений

Как уже было упомянуто во введении, существует большое количество различных видов ограничений на перемещение транспортных средств на дорожных сетях, а также мы сами можем потребовать выполнение дополнительных условий. Рассмотрим некоторые из таких ограничений.

Запрещенные последовательности участков дорог: Пусть дан граф дорожной сети $G(V, E, f, w)$ и некоторое разбиение множества дуг: $E = E_z \cup E_n, E_z \cap E_n \neq \emptyset$. Множество E_z будем называть множеством запрещённых дуг, а E_n – множеством нейтральных дуг. [1], [4]

Зададим ограничение, согласно которому на таком графе невозможно проходить более одного раза подряд через дугу $\in E_z$.

Это ограничение можно интерпретировать по-разному, например: пусть дуги графа – это дороги дальнего расстояния, дуги $\in E_n$ – дороги, в которых присутствует автомобильная заправочная станция, а дуги $\in E_z$ – дороги в которых отсутствует. Требуется, чтобы в пути не было две последовательно идущих дуг $\in E_z$

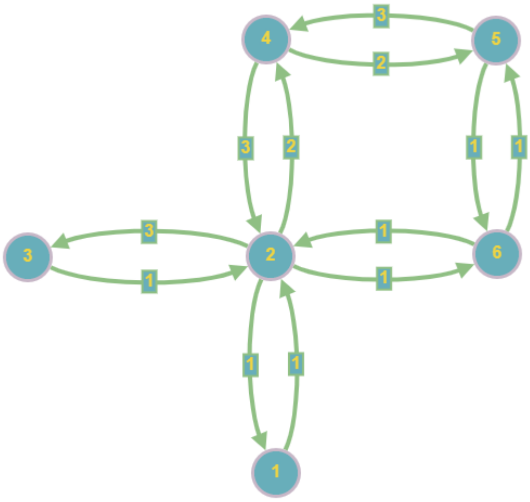


Рис. 1. Граф G без ограничений

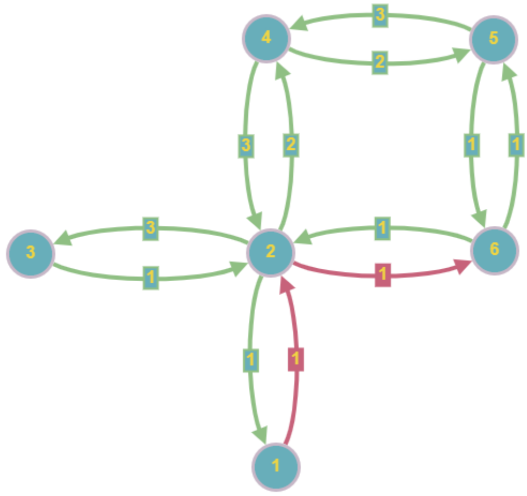


Рис. 2. Граф G с ограничениями

Здесь $E_n = \{e_2, e_3, e_4, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}\}$, а $E_z = \{e_1, e_5\}$, где $f(e_1) = (1,2), f(e_2) = (2,1), f(e_3) = (3,2), f(e_4) = (2,3), f(e_5) = (2,6), f(e_6) = (6,2), f(e_7) = (2,4), f(e_8) = (4,2), f(e_9) = (4,5), f(e_{10}) = (5,4), f(e_{11}) = (5,6), f(e_{12}) = (6,5)$.

По требованию, любой путь на этом графе не может содержать две инцидентные одной и той же вершине дуги $\in E_z$. Напрямую ранее упомянутые алгоритмы запускать не можем, так как нельзя рассматривать все пути между вершинами, а только разрешенные. Например, если на графе G - без ограничений кратчайший путь от вершины 1 до 5 это – $\mathbf{P} = \langle e_1, e_5, e_{12} \rangle$, то на графе G с ограничениями этот путь рассматривать нельзя. Чуть позднее покажем способ нахождения кратчайшего пути на графе с таким типом ограничений путём построения вспомогательного графа.

Запрещенные повороты: Другой особенностью дорожной сети является наличие запрещенных поворотов, т. е. переход между двумя дугами инцидентные одной вершине. Рассмотрим такие ограничения (рис. 3 - 6):

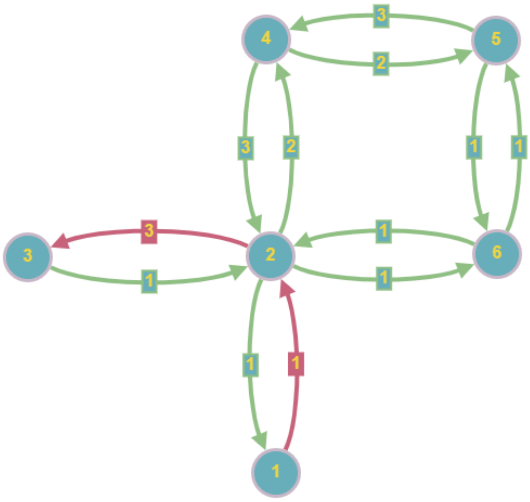


Рис. 3. Запр. поворот налево

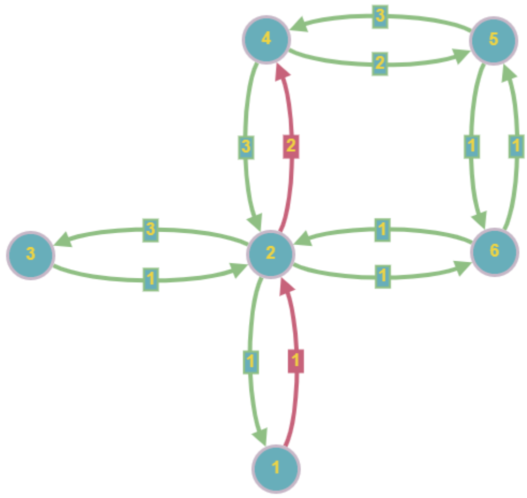


Рис. 4. Запр. поворот прямо

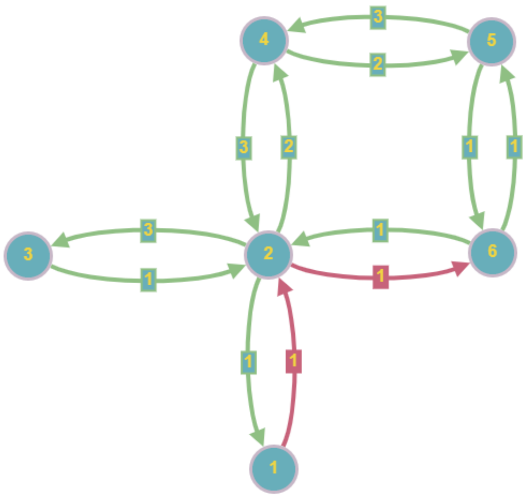


Рис. 5. Запр. поворот направо

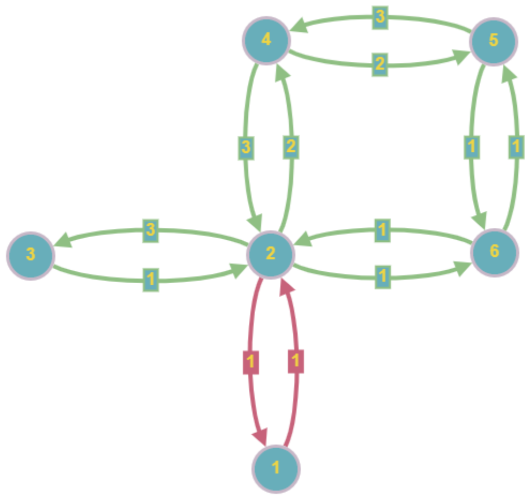


Рис. 6. Запр. поворот назад

Также, ограничения на поворот могут быть в виде комбинации нескольких таких ограничений на одном перекрестке. Уточним, что под нашим определением поворота (1.6), подходит и движение направо, и разворот. Решение представим в следующих подразделах.

Другие типы ограничений: Конечно, список всех типов ограничений на этом не заканчивается. Современные навигаторы, такие как *OpenStreetMap, Google Maps, Yandex.Navigator*, позволяют находить кратчайший (или оптимальный) путь, учитывая большое количество различных критериев (англ. multi-criteria pathfinding). В этой работе ограничимся рассмотрением только двух упомянутых ограничений.

3.2. Представление дорожного графа с учётом ограничений

Обычно, в работах по этой тематике (т. е. нахождение кратчайшего пути при наличии ограничений на графе), можно встретить три метода решения таких задач, которых предлагают авторы:

- построение вспомогательного графа;
- построение рёберного графа (line graph, edge - based graph);
- прямой метод (т. е. модифицирование алгоритмов).

Дальше, будем рассматривать каждый из этих методов.

Построение вспомогательного графа: используем для решения задачи с первым типом ограничений из 3.1. Правила его построения таковы: каждая вершина $v \in V$ исходного графа G заменяется двумя вершинами $v_0, v_1 \in V'$ на графе $G'(V', E', f', w')$ (т. е. вспомогательном), а по следующим правилам строятся дуги: [4]

1) каждой дуге $e \in (E_n \cap out(v)), f(e) = (v, t)$ ставится в соответствие две следующие дуги на графе G': (рис. 7)

$$e_1 : f'(e_1) = (v_0, t_0), w'(e_1) = w(e) \text{ и } e_2 : f'(e_2) = (v_1, t_0), w'(e_2) = w(e_2)$$

2) каждой дуге $e \in (E_z \cap out(v)), f(e) = (v, t)$ ставится в соответствие одна следующая дуга на графе G': (рис. 8)

$$e_1 : f'(e_1) = (v_0, t_1), w'(e_1) = w(e)$$

Таким образом, получаем новый граф, где: $|V'| = 2 \cdot |V|$ и $|E'| = 2 \cdot |E_n| + |E_z|$.

Теоремы о соответствие путей исходного графа на вспомогательном приведены в работах [1], [4].

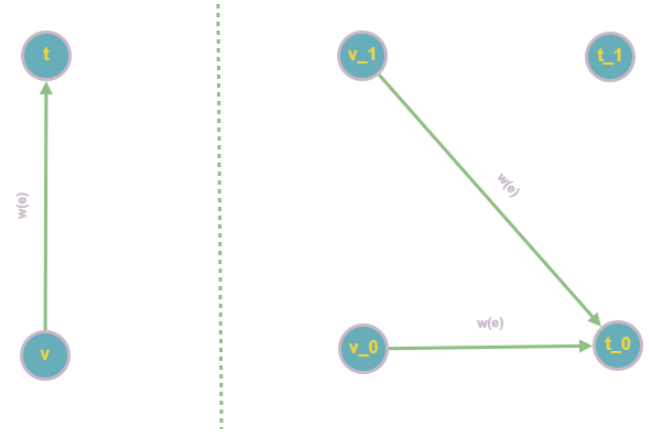


Рис. 7. Построение нейтральной дуги

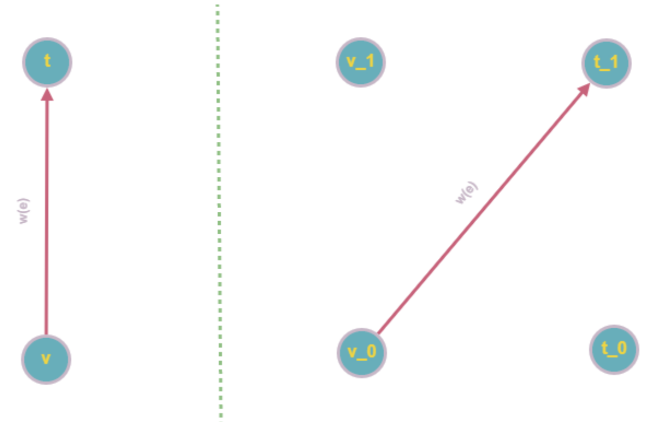


Рис. 8. Построение запрещенной дуги

Построение рёберного графа: этот метод используем для решения задачи со вторым типом ограничений из 3.1, в целях интегрирования информации о разрешённых поворотах в саму структуру графа. [6], [8]

Пусть дан граф $G(V, E, f, w)$. Тогда, назовём граф $G'(V', E', f', w')$ рёберным графом относительно графа G, где:

$$V' = E,$$
$$E' = \{((v_1, v_2), (v_3, v_4)) \in V' \times V' : v_2 = v_3\},$$
$$f'(\varepsilon) = ((v_1, v_2), (v_2, v_4)), \text{ где } \varepsilon \in E' \text{ и } (v_1, v_2), (v_2, v_4) \in V',$$
$$w'(\varepsilon) = w((v_1, v_2)), \text{ где } (v_1, v_2) \in E.$$

Опишем правила этого преобразования словами: каждая вершина нового графа G' представляет собой одну дугу графа G, а каждая дуга графа G' – пару смежных дуг графа G. Весовая функция для дуги $\in E'$ была определена выше, но можно было бы установить и другую. Если существует запрещённый поворот на исходном графе, то в графе G', новая дуга $\in E'$ представляющая пару смежных дуг (т. е. этот самый поворот) – не строится.

Кроме этого, **важно**, чтобы все вершины исходного графа имели хотя бы одну исходящую дугу, иначе при построении рёберного графа, вершина, не имеющая исходящей дуги, станет изолированной (т. е. эта вершина не будет являться конечной ни для какой дуги):

$$\forall v \in V : |out(v)| \geq 1$$

Представлены рисунки иллюстрирующие такое преобразование (см. рис. 9 - 10).

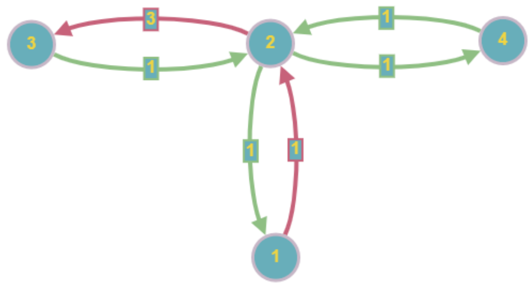


Рис. 9. Исходный граф G с запрещённым поворотом: $T = (1, 2, 3)$

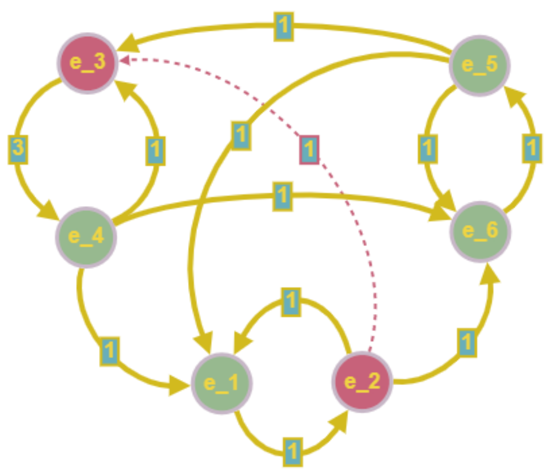


Рис. 10. Граф G', дуга представляющая запр. поворот, нарисована пунктирной линией (для наглядности), хотя к графу G' не принадлежит.

Однако, только этого преобразования не вполне достаточно, чтобы можно было запустить какой-нибудь алгоритм нахождения кратчайшего пути между парами вершин графа. Поясним предыдущее утверждение: например, на исходном графе G требовалось найти кратчайший путь между начальной вершиной s и конечной t, но после преобразования в рёберный граф, вершину s заменяют дуги $\in out(s)$, а вершину t заменяют дуги $\in in(t)$. Таким образом, задача нахождения кратчайшего пути между парами вершин превратилась в задачу нахождения между несколькими и несколькими вершинами.

В целях приведения обратно в задачу поиска кратч. пути между парами вершин, мы будем добавлять в граф G' фиктивные дуги и вершины по следующим правилам:

(a) добавим все вершины исходного графа: $V'_{new} = V' \cup V$;

(b) строим дуги между каждой $v \in V$ и $v' \in V' : source(v') = v$, с весом равным нулю, (напоминание: v' – дуга в E);

(c) строим дуги между каждой $v' \in V'$ и $v \in V : v = target(v')$, с весом равным весу дуги v';

Таким образом, получаем новый граф G'_{new}, на котором можно запустить алгоритмы поиска кратч. путей (см. рис. 11).

После всех преобразований, получаем новый граф, где:

$$|V'_{new}| = |V' \cup V| = |E| + |V|;$$

$$|E'_{new}| = |E'| + 2 \cdot |V'| = \sum_{v \in V} |in(v)| \cdot |out(v)| - |T^*| + 2 \cdot |E|,$$

где T^* – это множество всех запрещённых поворотов.

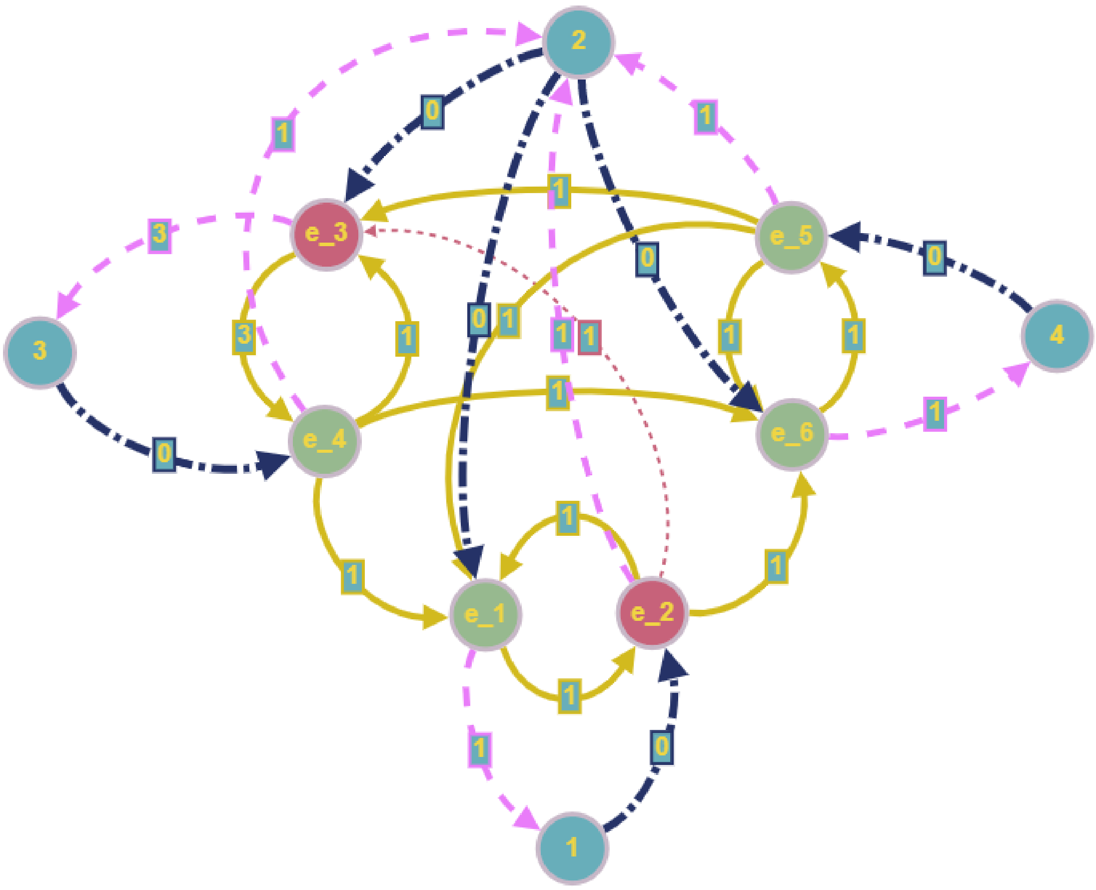


Рис. 11. Финальный граф G'_{new}, синими штрих-пунктирами нарисованы дуги (b), фиолетовыми пунктирными - дуги (c); и повторим, что дуга представляющая запр. поворот нарисована красной пунктирной линией (для наглядности), хотя к графу G'_{new} она не принадлежит.

Приведём алгоритм этого преобразования в виде псевдокода (4):

Algorithm 4 Алгоритм преобразования в рёберный граф

procedure TransformGraph($G = (V, E, f, w)$, List $BannedTurns$)
    ▷ Где G – граф, $BannedTurns$ – список запр. поворотов.
    $V' := E \cup V$
    $E' := \emptyset$
    Map $f' := \emptyset$
    Map $w' := \emptyset$
    for all $(v_1, v_2) \in E$ **do**
        for all $(v_2, v_3) \in E$ **do**
            if $(v_1, v_2, v_3) \notin BannedTurns$ **then**
                $E'.Append((v_1, v_2), (v_2, v_3))$ ▷ Добавление дуги.
                $w'[((v_1, v_2), (v_2, v_3))] := w[(v_1, v_2)]$ ▷ Установка веса дуги.
                $f'[((v_1, v_2), (v_2, v_3))] := ((v_1, v_2), (v_2, v_3))$ ▷ Установка отоб.
            end if
        end for
        for all $v \in V$ **do**
            if $v == source(v_1, v_2)$ **then**
                $E'.Append(v, (v_1, v_2))$ ▷ Добавление фикт. дуги (b).
                $w'[(v, (v_1, v_2))] := 0$ ▷ Установка веса дуги (b).
                $f'[(v, (v_1, v_2))] := (v, (v_1, v_2))$ ▷ Установка отобр. (b).
            end if
            if $v == target(v_1, v_2)$ **then**
                $E'.Append((v_1, v_2), v))$ ▷ Добавление фикт. дуги (c).
                $w'[((v_1, v_2), v)] := w[(v_1, v_2)]$ ▷ Установка веса дуги (c).
                $f'[((v_1, v_2), v)] := ((v_1, v_2), v)$ ▷ Установка отобр. (c).
            end if
        end for
    end for
    return $G'(V', E', f', w')$
end procedure

3.3. Модификация алгоритмов

В этом подразделе покажем способ решения задачи, при наличии на графе ограничений типа запрещенных поворотов путём модифицирования алгоритмов Дейкстры и A* без преобразования или создания вспомогательного графа.

На алгоритме Дейкстры (см. псевдокод 1) видим, что беря какуюто вершину из очереди, алгоритм пытается делать релаксацию вдоль всех исходящих из этой вершины дуг. Если существуют ограничения в виде запрещённых поворотов на графе, то алгоритм такие ограничения не сможет обработать. А если даже добавить дополнительные проверки, чтобы при появлении в пути тройки запрещённых вершин (иначе говоря, запр. поворота) алгоритм обходил такие повороты, то возможна ситуация, когда нужно будет снова обработать и добавить в путь уже посещённую вершину (при наличии запрещенных поворотов на графе, путь, может содержать цикл), но этого алгоритм тоже не сможет, т. к. значение $d[*]$ будет меньше в любом случае.

Сначала, приведём алгоритм функции, которая получая на входе дугу, вернёт все ей **разрешённые** смежные дуги (точнее, список $target(*)$ тех дуг с весом):

Algorithm 5 Функция определения разрешенных вершин для перемещения после прохождения по дуге e

```
procedure GetAllowedTargets(G = (V, E, f, w), Edge e, List BannedTurns)
        ▷ Где G – граф, e ∈ E, BannedTurns – список запр. поворотов.
    List Targets⟨Pair⟨Vertex, Weight⟩⟩ := ∅
    Vertex s := source(e)
    Vertex t := target(e)
    Vertex v ∈ V
    for all (t, v) ∈ E do
        if (s, t, v) ∉ BannedTurns then
            Pair temp := ⟨v, w[(t, v)]⟩
            Targets.push_back(temp)
        end if
    end for
    return Targets
end procedure
```

Далее, эту функцию интегрируем с алгоритмом Дейкстры (и с A*) таким образом: беря вершину u из очереди, каждую исходящую дугу из неё отправляем в качестве аргумента в функцию $GetAllowedTargets()$ и она вернёт список разрешенных конечных вершин дуг, которые смежны с ней. Далее, делаем релаксацию сразу вдоль каждых тех двух полученных смежных дуг, словно перескакивая через одну вершину.

Замечание: автор этой работы, в силу ограниченности времени, не смог полностью доработать этот вариант модификации. Возможно, в будущих работах будут приведены исправленные варианты с доказательством корректности. Поэтому, возможно, что в некоторых случаях этот вариант не сработает должным образом.

3.4. Программная реализация

Рассмотрим работу написанной программы, на нескольких вариантов входных данных. Входными данными в виде текстового файла служат: количество вершин графа, количество дуг, исходный граф, запрещенные повороты (если имеются).

Программа была тестирована и на небольших графах, и на графах графе больших размеров, данные которого были взяты из *9th DIMACS Implementation Challenge - Shortest Paths*. Второй упомянутый граф (реальные данные города – окрестности Нью-Йорка) содержит 264346 вершин, 733846 дуг с весами (рассстояние между вершинами в метрах), а также содержит географические координаты всех вершин (использовались для вычисления эвристической оценки). [?]

Приведём результаты некоторых тестирований: Первый пример (см. рис. 12, 13, 14):

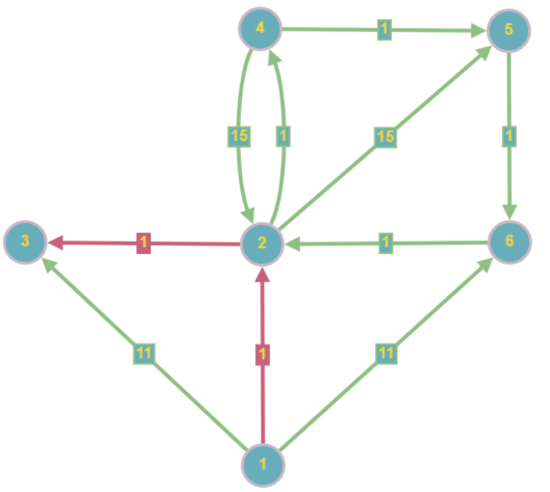
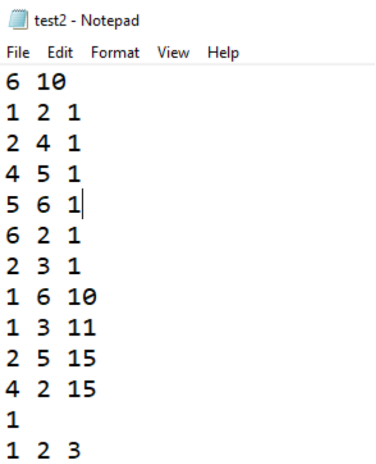


Рис. 12. Дуги красного цвета – можно интерпретировать и как запрещенный поворот, и как две запрещенные дуги.


test2 - Notepad

File Edit Format View Help

```
6 10
1 2 1
2 4 1
4 5 1
5 6 1
6 2 1
2 3 1
1 6 10
1 3 11
2 5 15
4 2 15
1
1 2 3
```

Рис. 13. Входной текст. файл.

Консоль отладки Microsoft Visual Studio

```
Runtime(Reading and initializing graph): 0s


*** Graph without restrictions ***
Runtime(Dijkstra's algorithm): 0s
Distance to target_vertex: 2
Path from 1 to 3:
1-2-3

*** Graph without restrictions ***
Runtime(Bellman-Ford algorithm): 0s
Distance to target_vertex: 2
Path from 1 to 3:
1-2-3

*** Graph with turn restrictions ***
Runtime(Modified Dijkstra's algorithm): 0s
Distance to target_vertex: 6
Path from 1 to 3:
1-2-4-5-6-2-3

Runtime(Splitting graph): 0s

*** Graph with restrictions (forbidden subsequences) ***
Runtime(Dijkstra's algorithm for graph with forbidden sequences - splitted): 0s
Distance to target_vertex: 6
Path from 1 to 3:
1-2-4-5-6-2-3

*** Graph with restrictions (forbidden subsequences) ***
Runtime(Bellman-Ford algorithm for graph with forbidden sequences - splitted): 0s
Distance to target_vertex: 6
Path from 1 to 3:
1-2-4-5-6-2-3

Press any key to continue . . .
```

Рис. 14. Результат работы программы

Так как эти данные были без географических координат, поэтому алгоритм A* запустить не можем.

Второй пример (см. рис. 15, 16, 17):

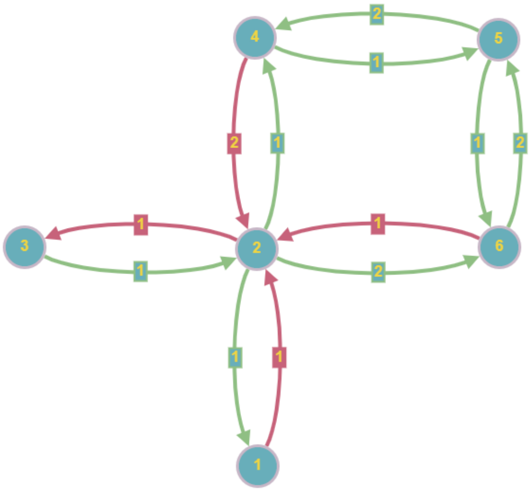


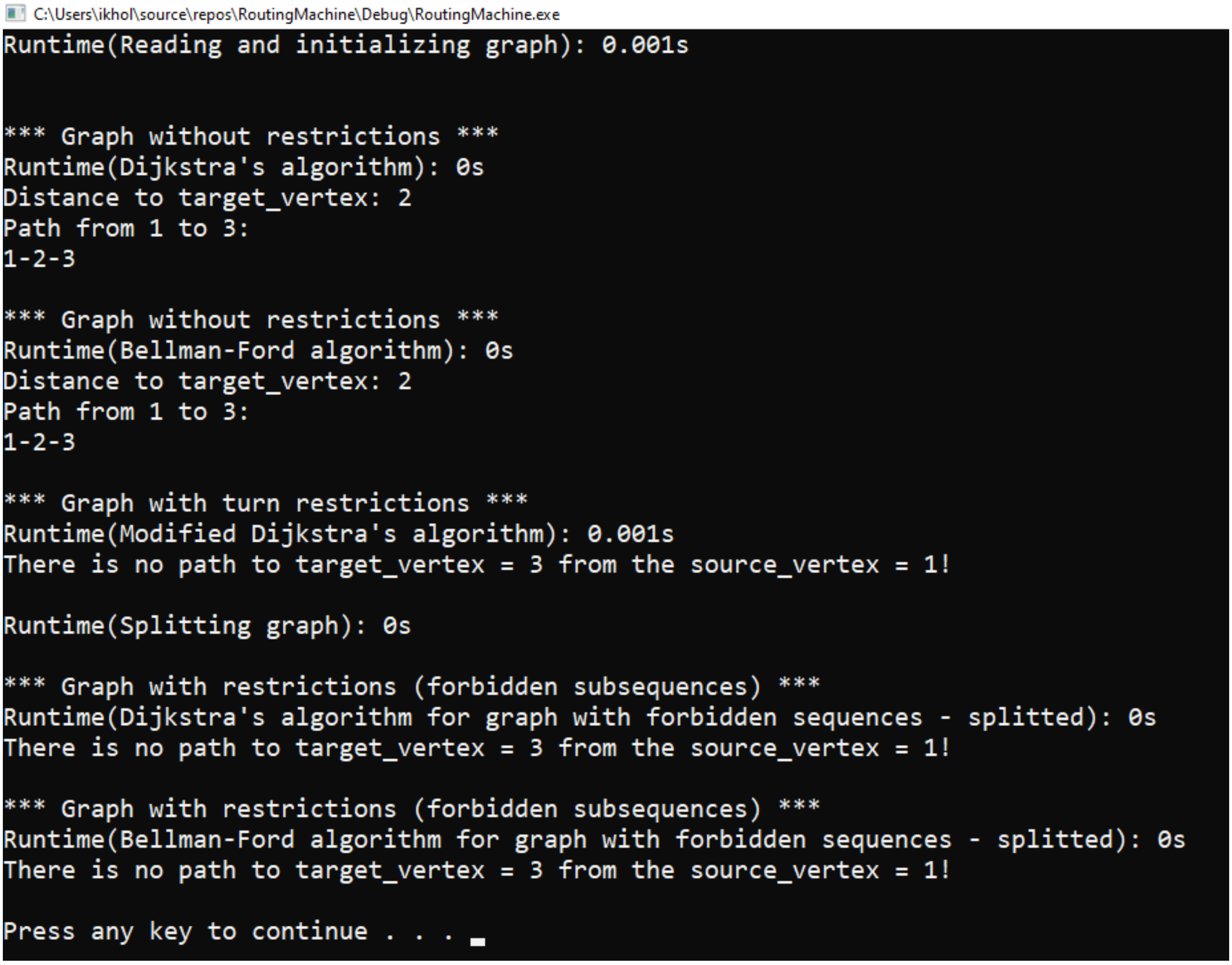
Рис. 15. Запрещенные повороты $T = (1,2,3), T = (4,2,3), T = (6,2,3)$

data_test - Notepad

File Edit Format View Help

```
6 12
1 2 1
2 1 1
2 3 1
3 2 1
2 6 2
6 2 1
2 4 1
4 2 2
6 5 2
5 6 1
4 5 1
5 4 2
3
1 2 3
4 2 3
6 2 3
```

Рис. 16. Входной текст. файл.



```
C:\Users\ikhol\source\repos\RoutingMachine\Debug\RoutingMachine.exe
Runtime(Reading and initializing graph): 0.001s


*** Graph without restrictions ***
Runtime(Dijkstra's algorithm): 0s
Distance to target_vertex: 2
Path from 1 to 3:
1-2-3

*** Graph without restrictions ***
Runtime(Bellman-Ford algorithm): 0s
Distance to target_vertex: 2
Path from 1 to 3:
1-2-3

*** Graph with turn restrictions ***
Runtime(Modified Dijkstra's algorithm): 0.001s
There is no path to target_vertex = 3 from the source_vertex = 1!

Runtime(Splitting graph): 0s

*** Graph with restrictions (forbidden subsequences) ***
Runtime(Dijkstra's algorithm for graph with forbidden sequences - splitted): 0s
There is no path to target_vertex = 3 from the source_vertex = 1!

*** Graph with restrictions (forbidden subsequences) ***
Runtime(Bellman-Ford algorithm for graph with forbidden sequences - splitted): 0s
There is no path to target_vertex = 3 from the source_vertex = 1!

Press any key to continue . . .
```

Рис. 17. Результат работы программы

И запустим программу с графом большого размера. Так как в исходном графе нет данных о запрещенных поворотах и запрещенных дуг, то была написана функция, генерирующая их. Результат работы программы, однако, проверить не получится, разве что можно со значением эвристической оценки сравнить. Сначала, без ограничений (см. рис. 18 и 19):

C:\Users\ikhol\source\repos\RoutingMachine\Debug\RoutingMachine.exe

```
Runtime(Reading and initializing graph): 12.179s

Heuristic between source and target: 4195

*** Graph without restrictions ***
Runtime(Dijkstra's algorithm): 0.083s
Distance to target_vertex: 5113
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-
3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-
3211-3261-3262-3923-3924-3864-3926-4

*** Graph without restrictions ***
Runtime(A* algorithm): 0.004s
Distance to target_vertex: 5113
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-
3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-
3211-3261-3262-3923-3924-3864-3926-4

*** Graph without restrictions ***
Runtime(Bellman-Ford algorithm): 51.726s
Distance to target_vertex: 5113
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-
3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-
3211-3261-3262-3923-3924-3864-3926-4

*** Graph with turn restrictions ***
Runtime(Modified Dijkstra's algorithm): 0.074s
Distance to target_vertex: 5113
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-
3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-
3211-3261-3262-3923-3924-3864-3926-4

*** Graph with turn restrictions ***
Runtime(Modified A* algorithm): 0.015s
Distance to target_vertex: 5113
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-
3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-
3211-3261-3262-3923-3924-3864-3926-4
```

Рис. 18. Первая часть результата работы программы

```
C:\Users\ikhol\source\repos\RoutingMachine\Debug\RoutingMachine.exe

Runtime(Splitting graph): 0.553s

*** Graph with restrictions (forbidden subsequences) ***
Runtime(Dijkstra's algorithm for graph with forbidden sequences - splitted): 0.012s
Distance to target_vertex: 5113
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-
3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-
3211-3261-3262-3923-3924-3864-3926-4

*** Graph with restrictions (forbidden subsequences) ***
Runtime(Bellman-Ford algorithm for graph with forbidden sequences - splitted): 49.978s
Distance to target_vertex: 5113
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-
3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-
3211-3261-3262-3923-3924-3864-3926-4

*** Graph with restrictions (forbidden subsequences) ***
Runtime(A* algorithm for graph with forbidden sequences - splitted): 0.004s
Distance to target_vertex: 5113
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-
3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-
3211-3261-3262-3923-3924-3864-3926-4

Press any key to continue . . .
```

Рис. 19. Вторая часть результата работы программы

Теперь, со сгенерированными ограничениями (см. рис. 20 и 21):

```
C:\Users\ikhol\source\repos\RoutingMachine\Debug\RoutingMachine.exe
Runtime(Reading and initializing graph): 12.322s

Heuristic between source and target: 4195

*** Graph without restrictions ***
Runtime(Dijkstra's algorithm): 0.087s
Distance to target_vertex: 5113
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-
3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-
3211-3261-3262-3923-3924-3864-3926-4

*** Graph without restrictions ***
Runtime(A* algorithm): 0.004s
Distance to target_vertex: 5113
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-
3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-
3211-3261-3262-3923-3924-3864-3926-4

*** Graph without restrictions ***
Runtime(Bellman-Ford algorithm): 51.467s
Distance to target_vertex: 5113
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-
3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-
3211-3261-3262-3923-3924-3864-3926-4

Runtime(Generating turn restriction): 15.664s
Count of generated turns: 10483

*** Graph with turn restrictions ***
Runtime(Modified Dijkstra's algorithm): 32.612s
Distance to target_vertex: 5255
Path from 1 to 4:
1-1363-1364-1366-1367-1368-1388-1391-1393-1395-1398-1386-1400-1451-1449-
1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-3142-
3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-3211-
3261-3262-3923-3924-3864-3926-4

*** Graph with turn restrictions ***
Runtime(Modified A* algorithm): 6.683s
Distance to target_vertex: 5255
Path from 1 to 4:
1-1363-1364-1366-1367-1368-1388-1391-1393-1395-1398-1386-1400-1451-1449-
1442-1444-1445-1447-1448-1460-1461-1462-1463-1474-3137-3138-3139-3142-
3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-3206-3207-3211-
3261-3262-3923-3924-3864-3926-4

Runtime(Splitting graph): 52.642s
```

Рис. 20. Первая часть результата работы программы

```
C:\Users\ikhol\source\repos\RoutingMachine\Debug\RoutingMachine.exe

Runtime(Splitting graph): 52.642s

*** Graph with restrictions (forbidden subsequences) ***
Runtime(Dijkstra's algorithm for graph with forbidden sequences - splitted): 0.163s
Distance to target_vertex: 5275
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1472-1462-1463-1474-3137-
3138-3139-3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-
3206-3207-3211-3261-3262-3923-3924-3864-3926-4

*** Graph with restrictions (forbidden subsequences) ***
Runtime(Bellman-Ford algorithm for graph with forbidden sequences - splitted): 96.351s
Distance to target_vertex: 5275
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1472-1462-1463-1474-3137-
3138-3139-3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-
3206-3207-3211-3261-3262-3923-3924-3864-3926-4

*** Graph with restrictions (forbidden subsequences) ***
Runtime(A* algorithm for graph with forbidden sequences - splitted): 0.007s
Distance to target_vertex: 5362
Path from 1 to 4:
1-1363-1358-1357-1359-1280-1287-1371-1373-1374-1382-1383-1381-1385-1387-
1443-1442-1444-1445-1447-1448-1460-1461-1462-1472-1462-1463-1474-3137-
3138-3139-3142-3141-3145-3158-3160-3157-3164-3165-3175-3176-3178-3180-
3206-3207-3212-3213-3225-3262-3923-3924-3864-3926-4

Press any key to continue . . .
```

Рис. 21. Вторая часть результата работы программы

Заключение

При написании данной курсовой работы автор познакомился со множеством научных работ по этой тематике, изучил методы и алгоритмы нахождения кратчайших путей на графах, модифицировал некоторые из них и написал соответствующую программу, которая решает задачу несколькими алгоритмами и методами представления графа.

Исходный код и все входные данные доступны по этому адресу [10].

Литература

1. Ерусалимский Я. М., Скороходов В. А., Кузьминова М. В., Петросян А. Г. Графы с нестандартной достижимостью: задачи, приложения. – Ростов н/Д.: Южный федеральный университет, 2009. – 195с.: ил.
2. Н.Кристофидес Теория графов. Алгоритмический подход. М.: Мир, 1978, 432 стр.
3. Кормен, Томас Х. и др. А45 Алгоритмы: построение и анализ, 3-е изд. : Пер. с англ. — М. : ООО "И. Д. Вильямс", 2013. — 1328 с. : ил.
4. Письменский М. А. Алгоритм Дейкстры на графах с нестандартной достижимостью. https://hub.lib.sfedu.ru/storage/1/1290079/76fa08e9-b063-4dd2-96af-027316551a9a/ [дата обр.: 07.10.2021]
5. D. M. Laparra. Pathfinding algorithms in graphs and applications. – 2019. http://diposit.ub.edu/dspace/bitstream/2445/140466/1/memoria.pdf [дата обр.: 06.10.2021]
6. L. Volker. Route Planning in Road Networks with Turn Costs. – 2008. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.331.8085&rep=rep1&type=pdf [дата обр.: 06.10.2021]
7. D. Delling, P. Sanders, D. Shultes and D, Wagner. Engineering Route Planning Algorithms. – Karlsruhe, Germany, 2009. https://i11www.iti.kit.edu/extra/publications/dssw-erpa-09.pdf [дата обр.: 06.10.2021]
8. S. Winter, A. Gruhbacker. Modeling Costs of Turns in Route Planning. – Vienna, Austria, 2009. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.24.8035&rep=rep1&type=pdf [дата обр.: 06.10.2021]
9. https://e-maxx.ru/algo/ford_bellman [дата обр.: 07.10.2021]
10. https://github.com/gitkholis/FinalPathfinder [дата обр.: 07.10.2021]

```
#define _CRT_SECURE_NO_WARNINGS
#define _USE_MATH_DEFINES
#include <bits/stdc++.h>
#include <random>
using namespace std;

const int INF = 999999999;

double readInputData(vector<vector<pair<int, int>>>& graph,
                     vector<tuple<int, int, int>>& banned_turns,
                     int& vertex_count, int& arc_count,
                     vector<pair<long double, long double>>& vertex_coordinates,
                     bool data_with_coordinates) {
    clock_t start_time = clock();
    int banned_turns_count;
    char source_file[] = "nyc_data_meters.txt";
    ifstream fin;
    fin.open(source_file);
    fin >> vertex_count >> arc_count;
    graph.resize(vertex_count + 1);
    for (int i = 1; i <= arc_count; ++i) {
        int from_vertex, to_vertex, arc_weight;
        fin >> from_vertex >> to_vertex >> arc_weight;
        graph[from_vertex].push_back({ to_vertex, arc_weight });
    }
    fin >> banned_turns_count;
    for (int i = 0; i < banned_turns_count; ++i) {
        int from_vertex, by_vertex, to_vertex;
        fin >> from_vertex >>by_vertex >> to_vertex;
        banned_turns.push_back({ from_vertex, by_vertex , to_vertex});
    }
    fin.close();
    if (data_with_coordinates) {
        char source_file_coord[] = "nyc_coordinates.txt";
        fin.open(source_file_coord);
        fin >> vertex_count;
        vertex_coordinates.push_back({ 0 , 0 });
        for (int i = 1; i <= vertex_count; ++i) {
            int vertex_id, latitude, longitude;
            fin >> vertex_id >> latitude >> longitude;
            vertex_coordinates.push_back({ (long double)latitude/1000000,
              (long double)longitude/1000000 });
        }
        fin.close();
    }
    clock_t end_time = clock();
    return (end_time - start_time);
}

pair<int, int> findArcsWeight(vector<vector<pair<int, int>>>& source_graph,
                              int& from_vertex, int& to_vertex,
                              vector<tuple<int, int, int>>&
                    temporary_arc_copies) {
    for (tuple<int, int, int> arc : temporary_arc_copies) {
        if (from_vertex == get<0>(arc) and to_vertex == get<1>(arc))
            return { get<2>(arc), 1 };
```

```cpp
    }
    for (int i = 0; i < source_graph[from_vertex].size(); ++i) {
        if (to_vertex == source_graph[from_vertex][i].first) {
            int cost = source_graph[from_vertex][i].second;
            temporary_arc_copies.push_back({ from_vertex, to_vertex, cost });
            source_graph[from_vertex].erase(source_graph[from_vertex].begin() + i);
            return { cost, 0 };
        }
    }
}

double splitGraph(vector<vector<pair<int, int>>>& source_graph,
                  vector<vector<pair<int, int>>>& splitted_graph,
                  vector<tuple<int, int, int>>& banned_turns) {
    clock_t start_time = clock();
    if (banned_turns.size() != 0) {
        vector<tuple<int, int, int>> temporary_arc_copies;
        int source_graph_size = source_graph.size();
        splitted_graph.resize(2 * source_graph_size - 1);
        for (int i = 0; i < banned_turns.size(); ++i) {
            int from_vertex = get<0>(banned_turns[i]);
            int by_vertex = get<1>(banned_turns[i]);
            int to_vertex = get<2>(banned_turns[i]);
            pair<int, int> cost_and_bool;
            cost_and_bool = findArcsWeight(source_graph, from_vertex, by_vertex, temporary_arc_copies);
            if (!cost_and_bool.second)
                splitted_graph[from_vertex].push_back({ by_vertex + source_graph_size - 1, cost_and_bool.first });
            cost_and_bool = findArcsWeight(source_graph, by_vertex, to_vertex, temporary_arc_copies);
            if (!cost_and_bool.second)
                splitted_graph[by_vertex].push_back({ to_vertex + source_graph_size - 1, cost_and_bool.first });
        }
        for (int i = 1; i < source_graph.size(); ++i) {
            for (int j = 0; j < source_graph[i].size(); ++j) {
                pair<int, int> arc = source_graph[i][j];
                splitted_graph[i].push_back({ arc.first, arc.second });
                splitted_graph[i + source_graph_size - 1].push_back({ arc.first, arc.second });
            }
        }
    }
    else {
        splitted_graph = source_graph;
    }
    clock_t end_time = clock();
    return (end_time - start_time);
}

int generateRandomNumber(int start_range, int end_range) {
    random_device                    rand_dev;
    mt19937                          generator(rand_dev());
    uniform_int_distribution<int>    distr(start_range, end_range);
```

```
    return distr(generator);
}

bool findInBannedTurnsVector(vector<tuple<int, int, int>>& banned_turns,
                             tuple<int, int, int> banned_turn) {
    for (tuple<int, int, int> temp : banned_turns) {
        if ((get<0>(temp) == get<0>(banned_turn))
            and (get<1>(temp) == get<1>(banned_turn))
            and (get<2>(temp) == get<2>(banned_turn))) {
                return true;
        }
    }
    return false;
}

double generateTurnRestrictions(vector<vector<pair<int, int>>>& graph,
                              vector<tuple<int, int, int>>& banned_turns,
                              int& vertex_count, int& arc_count) {
    clock_t start_time = clock();
    char source_file[] = "random_turn_r.txt";
    ofstream fin;
    fin.open(source_file, ios_base::app);
    int turn_restrictions_count = arc_count / 50;
    for (int i = 1; i <= turn_restrictions_count; ++i) {
        int from_vertex, by_vertex, to_vertex;
        from_vertex = generateRandomNumber(1, vertex_count);
        if (graph[from_vertex].size()) {
            int by_vertex_index = generateRandomNumber(0, graph[from_vertex].size()-1);
            by_vertex = graph[from_vertex][by_vertex_index].first;
            int to_vertex_index = generateRandomNumber(0, graph[by_vertex].size()-1);
            to_vertex = graph[by_vertex][to_vertex_index].first;
            tuple<int, int, int> banned_turn = make_tuple(from_vertex, by_vertex, to_vertex);
            if (!findInBannedTurnsVector(banned_turns, banned_turn)) {
                banned_turns.push_back(banned_turn);
            }
        }
    }
    fin << banned_turns.size() << endl;
    for (tuple<int, int, int> banned_turn : banned_turns) {
        fin << get<0>(banned_turn) << " " << get<1>(banned_turn) << " " << get<2>(banned_turn) << endl;
    }
    fin.close();
    clock_t end_time = clock();
    return (end_time - start_time);
}

void printPath(vector<int>& ancestors,
               int& source_vertex,
               int& target_vertex,
               vector<int>& distances,
               int& vertex_count,
               bool& graph_is_splitted,
```

```
                 vector<pair<long double, long double>>& vertex_coordinates) {
    vector<int> path;
    int v = target_vertex;
    if (ancestors[v] == -1) {
        if (ancestors[v + vertex_count] == -1) {
            printf("There is no path to target_vertex = %d from the source_vertex = %d!\n", target_vertex, source_vertex);
            return;
        }
    }
    if (distances.size() > vertex_count + 1) {
        if (graph_is_splitted and (distances[v] >= distances[v + vertex_count])) {
            v = v + vertex_count;
        }
    }

    cout << "Distance to target_vertex: " << distances[v] << endl;
    for (; v != source_vertex; v = ancestors[v]) {
        if (v > vertex_count) {
            path.push_back(v - vertex_count);
        }
        else {
            path.push_back(v);
        }
    }
    path.push_back(source_vertex);
    reverse(path.begin(), path.end());
    cout << "Path from " << source_vertex << " to " << target_vertex << ":\n";
    //ofstream fout;
    //char pathh[] = "path2.csv";
    //fout.open(pathh);
    //fout << setprecision(9) << endl;
    for (size_t i = 0; i < path.size() - 1; ++i) {
        cout << path[i] << '-';
        //fout << vertex_coordinates[i + 1].first << "," << vertex_coordinates[i + 1].second << endl;
        if (i != 0 and i % 14 == 0)
            cout << endl;
    }
    cout << path[path.size() - 1] << endl;
    //fout << vertex_coordinates[path.size()].first << "," << vertex_coordinates[path.size() - 1].second << endl;
    //fout.close();
}

void printPath2(vector<pair<int, int>>& ancestors,
    int& source_vertex,
    int& target_vertex,
    vector<int>& distances,
    int& vertex_count,
    bool& graph_is_splitted,
    vector<pair<long double, long double>>& vertex_coordinates) {
    vector<int> path;
    int v = target_vertex;
    if (distances[v] == INF) {
```

```
        printf("There is no path to target_vertex = %d from the source_vertex = %d!\n", target_vertex, source_vertex);
        return;
    }
    cout << "Distance to target_vertex: " << distances[v] << endl;
    path.push_back(v);
    for (; v != source_vertex;) {
        pair<int, int> temp = ancestors[v];
        path.push_back(temp.second);
        if (temp.first != -5) {
            path.push_back(temp.first);
            v = ancestors[v].first;
        }
        else {
            v = ancestors[v].second;
        }

    }
    //path.push_back(source_vertex);
    reverse(path.begin(), path.end());
    cout << "Path from " << source_vertex << " to " << target_vertex << ":\n";
    //ofstream fout;
    //char pathh[] = "path2.csv";
    //fout.open(pathh);
    //fout << setprecision(9) << endl;
    for (size_t i = 0; i < path.size() - 1; ++i) {
        cout << path[i] << '-';
        //fout << vertex_coordinates[i + 1].first << "," << vertex_coordinates[i + 1].second << endl;
        if (i != 0 and i % 14 == 0)
            cout << endl;
    }
    cout << path[path.size() - 1] << endl;
    //fout << vertex_coordinates[path.size()].first << "," << vertex_coordinates[path.size() - 1].second << endl;
    //fout.close();
}

double Dijkstra(vector<vector<pair<int, int>>>& adjacency_list,
                vector<int>& distances,
                vector<int>& ancestors,
                int& source_vertex,
                int& target_vertex,
                int& vertex_count) {
    clock_t start_time = clock();
    clock_t end_time;
    priority_queue<pair<int, int>> Queue;
    distances.assign(adjacency_list.size(), INF);
    ancestors.assign(adjacency_list.size(), -1);
    distances[source_vertex] = 0;
    Queue.push({ distances[source_vertex], source_vertex });
    while (!Queue.empty()) {
        pair<int, int> u = Queue.top(); Queue.pop();
        if ((u.second == target_vertex) or (u.second == target_vertex + vertex_count)) {
            end_time = clock();
```

```
            return (end_time - start_time);
        }
        for (pair<int, int> arc : adjacency_list[u.second]) {
            int v = arc.first;
            int alt = (distances[u.second] + arc.second);
            if (distances[v] > alt) {
                distances[v] = alt;
                Queue.push({ -alt, v });
                ancestors[v] = u.second;
            }
        }
    }
    end_time = clock();
    return (end_time - start_time);
}

void getAllowedTargets(vector<vector<pair<int, int>>>& adjacency_list,
                vector<tuple<int, int, int>>& banned_turns,
                int& s, int& t,
                vector<pair<int, int>>& targets) {
    for (int i = 0; i < adjacency_list[t].size(); ++i) {
        tuple<int, int, int> turn = make_tuple(s, t, adjacency_list[t][i].first);
        if (!findInBannedTurnsVector(banned_turns, turn)) {
            targets.push_back({adjacency_list[t][i].first, adjacency_list[t][i].second });
        }
    }
}

double DijkstraModified(vector<vector<pair<int, int>>>& adjacency_list,
                vector<int>& distances,
                vector<pair<int, int>>& ancestors,
                int& source_vertex,
                int& target_vertex,
                int& vertex_count,
                vector<tuple<int, int, int>>& banned_turns) {
    clock_t start_time = clock();
    clock_t end_time;
    priority_queue<pair<int, int>> Queue;
    vector<pair<int, int>> targets;
    distances.assign(adjacency_list.size(), INF);
    ancestors.assign(adjacency_list.size(), {});
    distances[source_vertex] = 0;
    Queue.push({ distances[source_vertex], source_vertex });
    while (!Queue.empty()) {
        pair<int, int> u = Queue.top(); Queue.pop();
        if ((u.second == target_vertex)) {
            end_time = clock();
            return (end_time - start_time);
        }

        for (pair<int, int> arc : adjacency_list[u.second]) {
            int v = arc.first;
            targets.clear();
            //distances[v] = distances[u.second] + arc.second;
```

```
            //ancestors[v] = { ancestors[u.second].second, u.second };
            /*if (v == target_vertex) {
                distances[v] = distances[u.second] + arc.second;
                ancestors[v] = { -5, u.second };
                end_time = clock();
                return (end_time - start_time);
            }*/
            //Queue.push({ -(distances[u.second] + arc.second), v });
            if (v == target_vertex) {
                if (ancestors[u.second].second != 0) {
                    if (!findInBannedTurnsVector(banned_turns, { ancestors[u.second].second, u.second, v })) {
                        if (distances[v] > distances[u.second] + arc.second) {
                            distances[v] = distances[u.second] + arc.second;
                            ancestors[v] = { -5, u.second };
                            end_time = clock();
                            return (end_time - start_time);
                        }

                    }
                    else {
                        continue;
                    }
                }
                /*else {
                    if (distances[v] > distances[u.second] + arc.second) {
                        distances[v] = distances[u.second] + arc.second;
                        ancestors[v] = { -5, u.second };
                        end_time = clock();
                        return (end_time - start_time);
                    }
                }*/
            }
            getAllowedTargets(adjacency_list, banned_turns, u.second, v, targets);
            for (int i = 0; i < targets.size(); ++i) {
                int alt = distances[u.second] + arc.second + targets[i].second;
                if (distances[targets[i].first] > alt ) {
                    distances[targets[i].first] = alt;
                    Queue.push({ -alt, targets[i].first });
                    ancestors[targets[i].first] = make_pair(u.second, v);
                    //ancestors[targets[i].first] = v;
                }
                /*if ((targets[i].first == target_vertex) or (targets[i].first == target_vertex + vertex_count)) {
                    end_time = clock();
                    return (end_time - start_time);
                }*/
            }
        }
    }
    end_time = clock();
    return (end_time - start_time);
}

```

```
double BellmanFord(vector<vector<pair<int, int>>>& adjacency_list,
                   vector<int>& distances,
                   vector<int>& ancestors,
                   int& source_vertex,
                   int& target_vertex) {
    clock_t start_time = clock();
    distances.assign(adjacency_list.size(), INF);
    ancestors.assign(adjacency_list.size(), -1);
    distances[source_vertex] = 0;
    for (;;) {
        bool flag = false;
        for (int i = 1; i < adjacency_list.size(); ++i) {
            for (pair<int, int> arc : adjacency_list[i]) {
                int from = i;
                int to = arc.first;
                int cost = arc.second;
                if (distances[to] > distances[from] + cost) {
                    distances[to] = distances[from] + cost;
                    ancestors[to] = from;
                    flag = true;
                }
            }
        }
        if (not flag)
            break;
    }
    clock_t end_time = clock();
    return (double)(end_time - start_time);
}

long double toRadians(const long double& degree) {
    return ((long double)(M_PI/180) * degree);
}

int heuristic(vector<pair<long double, long double>>& vertex_coordinates, int vertex_1, int vertex_2) {
    if (vertex_1 >= vertex_coordinates.size()) {
        vertex_1 -= vertex_coordinates.size();
    }
    if (vertex_2 >= vertex_coordinates.size()) {
        vertex_2 -= vertex_coordinates.size();
    }
    long double latitude_1;
    long double longitude_1;
    long double latitude_2;
    long double longitude_2;

    latitude_1 = toRadians(vertex_coordinates[vertex_1].first);
    longitude_1 = toRadians(vertex_coordinates[vertex_1].second);
    latitude_2 = toRadians(vertex_coordinates[vertex_2].first);
    longitude_2 = toRadians(vertex_coordinates[vertex_2].second);

    long double d_longitude = longitude_2 - longitude_1;
    long double d_latitude = latitude_2 - latitude_1;

    long double ans = pow(sin(d_latitude / 2), 2) +
```

```
        cos(latitude_1) * cos(latitude_2) * pow(sin(d_longitude / 2), 2);

    ans = 2 * asin(sqrt(ans));
    long double R = 6371;
    ans = 1000 * ans * R;
    return (int)ans;
}

double Astar(vector<vector<pair<int, int>>>& adjacency_list,
             vector<int>& distances,
             vector<int>& ancestors,
             int& source_vertex,
             int& target_vertex,
             int& vertex_count,
             vector<pair<long double, long double>>& vertex_coordinates) {
    clock_t start_time = clock();
    clock_t end_time;
    priority_queue<pair<int, int>> Queue;
    distances.assign(adjacency_list.size(), INF);
    ancestors.assign(adjacency_list.size(), -1);
    distances[source_vertex] = 0;
    Queue.push({ distances[source_vertex], source_vertex });
    while (!Queue.empty()) {
        pair<int, int> u = Queue.top(); Queue.pop();
        if ((u.second == target_vertex) or (u.second == target_vertex + vertex_count)) {
            end_time = clock();
            return (end_time - start_time);
        }
        for (pair<int, int> arc : adjacency_list[u.second]) {
            int v = arc.first;
            int alt = (distances[u.second] + arc.second);
            if (distances[v] > alt) {
                distances[v] = alt;
                Queue.push({ -(alt + heuristic(vertex_coordinates, v, target_vertex)), v });
                ancestors[v] = u.second;
            }
        }
    }
    end_time = clock();
    return (end_time - start_time);
}

double AstarModified(vector<vector<pair<int, int>>>& adjacency_list,
                     vector<int>& distances,
                     vector<pair<int, int>>& ancestors,
                     int& source_vertex,
                     int& target_vertex,
                     int& vertex_count,
                     vector<pair<long double, long double>>& vertex_coordinates,
                     vector<tuple<int, int, int>>& banned_turns) {
    clock_t start_time = clock();
    clock_t end_time;
    priority_queue<pair<int, int>> Queue;
```

```
    vector<pair<int, int>> targets;
    distances.assign(adjacency_list.size(), INF);
    ancestors.assign(adjacency_list.size(), {});
    distances[source_vertex] = 0;
    Queue.push({ distances[source_vertex], source_vertex });
    while (!Queue.empty()) {
        pair<int, int> u = Queue.top(); Queue.pop();
        if ((u.second == target_vertex) or (u.second == target_vertex + vertex_count)) {
            end_time = clock();
            return (end_time - start_time);
        }
        for (pair<int, int> arc : adjacency_list[u.second]) {
            int v = arc.first;
            targets.clear();
            /*if (v == target_vertex) {
                distances[v] = distances[u.second] + arc.second;
                ancestors[v] = { -5, u.second };
                end_time = clock();
                return (end_time - start_time);
            }*/
            if (v == target_vertex) {
                if (ancestors[u.second].second != 0) {
                    if (!findInBannedTurnsVector(banned_turns, { ancestors[u.second].second, u.second, v })) {
                        if (distances[v] > distances[u.second] + arc.second) {
                            distances[v] = distances[u.second] + arc.second;
                            ancestors[v] = { -5, u.second };
                            end_time = clock();
                            return (end_time - start_time);
                        }

                    }
                    else {
                        continue;
                    }
                }
                /*else {
                    if (distances[v] > distances[u.second] + arc.second) {
                        distances[v] = distances[u.second] + arc.second;
                        ancestors[v] = { -5, u.second };
                        end_time = clock();
                        return (end_time - start_time);
                    }
                }*/
            }
            getAllowedTargets(adjacency_list, banned_turns, u.second, v, targets);
            for (int i = 0; i < targets.size(); ++i) {
                int alt = distances[u.second] + arc.second + targets[i].second;
                if (distances[targets[i].first] > alt) {
                    distances[targets[i].first] = alt;
                    Queue.push({ -(alt + heuristic(vertex_coordinates, targets[i].first, target_vertex)), targets[i].first });
                    ancestors[targets[i].first] = make_pair(u.second, v);
```

```cpp
                }
                /*if ((targets[i].first == target_vertex) or (targets[i].first == target_vertex + vertex_count)) {
                    end_time = clock();
                    return (end_time - start_time);
                }*/
            }
        }
    }
    end_time = clock();
    return (end_time - start_time);
}

int main() {
    vector<vector<pair<int, int>>> graph;
    vector<tuple<int, int, int>> banned_turns;
    vector<pair<long double, long double>> vertex_coordinates;
    bool data_with_coordinates = true;  // <-
    bool data_without_coordinates = false;  // <-
    bool graph_is_splitted = true;      // <-
    bool graph_is_not_splitted = false;
    int vertex_count, arc_count;
    vector<int> distances;
    vector<int> ancestors1;
    vector<pair<int, int>> ancestors2;

    int source_vertex = 1; int target_vertex = 4;
    double runtime = readInputData(graph, banned_turns, vertex_count, arc_count, vertex_coordinates, data_with_coordinates); // <-
    cout << "Runtime(Reading and initializing graph): " << runtime / CLOCKS_PER_SEC << "s\n\n";

    /***************** Without turn restrictions *****************/
    cout << "Heuristic between source and target: " << heuristic(vertex_coordinates, source_vertex, target_vertex) << endl; // <-
    runtime = Dijkstra(graph, distances, ancestors1, source_vertex, target_vertex, vertex_count);
    cout << "\n*** Graph without restrictions ***" << endl;
    cout << "Runtime(Dijkstra's algorithm): " << runtime / CLOCKS_PER_SEC << "s \n";
    printPath(ancestors1, source_vertex, target_vertex, distances, vertex_count, graph_is_not_splitted, vertex_coordinates);

    runtime = Astar(graph, distances, ancestors1, source_vertex, target_vertex, vertex_count, vertex_coordinates);
    cout << "\n*** Graph without restrictions ***" << endl;
    cout << "Runtime(A* algorithm): " << runtime / CLOCKS_PER_SEC << "s \n";
    printPath(ancestors1, source_vertex, target_vertex, distances, vertex_count, graph_is_not_splitted, vertex_coordinates);

    runtime = BellmanFord(graph, distances, ancestors1, source_vertex, target_vertex);
    cout << "\n*** Graph without restrictions ***" << endl;
    cout << "Runtime(Bellman-Ford algorithm): " << runtime / CLOCKS_PER_SEC << "s \n";
    printPath(ancestors1, source_vertex, target_vertex, distances,
```

```
        vertex_count, graph_is_not_splitted, vertex_coordinates);

    /******************** With turn restrictions *****************/
    vector<vector<pair<int, int>>> splitted_graph, copy_of_graph;
    copy_of_graph = graph;
    runtime = generateTurnRestrictions(graph, banned_turns, vertex_count, arc_count);
    cout << "\nRuntime(Generating turn restriction): " << runtime / CLOCKS_PER_SEC << "s \n";
    cout << "Count of generated turns: " << arc_count / 70 << endl;
    runtime = DijkstraModified(graph, distances, ancestors2, source_vertex, target_vertex, vertex_count, banned_turns);
    cout << "\n*** Graph with turn restrictions ***" << endl;
    cout << "Runtime(Modified Dijkstra's algorithm): " << runtime / CLOCKS_PER_SEC << "s \n";
    printPath2(ancestors2, source_vertex, target_vertex, distances, vertex_count, graph_is_not_splitted, vertex_coordinates);

    runtime = AstarModified(graph, distances, ancestors2, source_vertex, target_vertex, vertex_count, vertex_coordinates, banned_turns);
    cout << "\n*** Graph with turn restrictions ***" << endl;
    cout << "Runtime(Modified A* algorithm): " << runtime / CLOCKS_PER_SEC << "s \n";
    printPath2(ancestors2, source_vertex, target_vertex, distances, vertex_count, graph_is_not_splitted, vertex_coordinates);

    /****************** With forbidden sequences *****************/
    runtime = splitGraph(copy_of_graph, splitted_graph, banned_turns);
    cout << "\nRuntime(Splitting graph): " << runtime / CLOCKS_PER_SEC << "s \n";

    runtime = Dijkstra(splitted_graph, distances, ancestors1, source_vertex, target_vertex, vertex_count);
    cout << "\n*** Graph with restrictions (forbidden subsequences) ***" << endl;
    cout << "Runtime(Dijkstra's algorithm for graph with forbidden sequences - splitted): " << runtime / CLOCKS_PER_SEC << "s \n";
    printPath(ancestors1, source_vertex, target_vertex, distances, vertex_count, graph_is_splitted, vertex_coordinates);

    runtime = BellmanFord(splitted_graph, distances, ancestors1, source_vertex, target_vertex);
    cout << "\n*** Graph with restrictions (forbidden subsequences) ***" << endl;
    cout << "Runtime(Bellman-Ford algorithm for graph with forbidden sequences - splitted): " << runtime / CLOCKS_PER_SEC << "s \n";
    printPath(ancestors1, source_vertex, target_vertex, distances, vertex_count, graph_is_splitted, vertex_coordinates);

    runtime = Astar(splitted_graph, distances, ancestors1, source_vertex, target_vertex, vertex_count, vertex_coordinates);
    cout << "\n*** Graph with restrictions (forbidden subsequences) ***" << endl;
    cout << "Runtime(A* algorithm for graph with forbidden sequences - splitted): " << runtime / CLOCKS_PER_SEC << "s \n";
    printPath(ancestors1, source_vertex, target_vertex, distances,
```

```
        vertex_count, graph_is_splitted, vertex_coordinates);

    cout << endl;
    system("pause");
    return 0;
}
```