



Shortest paths

Interlude

Граф — достаточно удобное место для хранения данных.

Что касается операций с вершинами: получать с них информацию ([Graph traversal, Pt. 1](#)  [Graph traversal, Pt. 2](#) ), сжимать ([конденсация](#)), находить циклы и т.д.

Но в графе также хранятся ребра, которые в прикладном/промышленном программировании несут не менее значимую информацию.

Связь между вершинами определяет отношение между вершинами, а ответственностью ребра в графе является хранение условий перехода из v в u .

Условие перехода в теории графов называется **весовой характеристикой**.

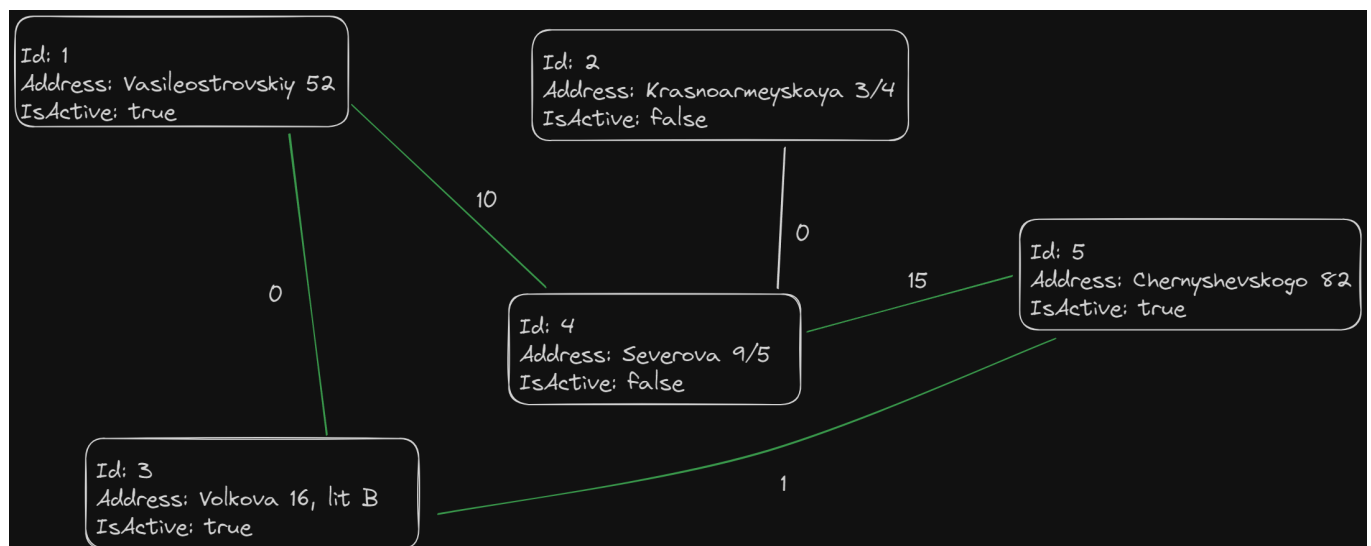
Поиск оптимального (минимального) условия W на пути (u, v) в теории графов называется кратчайшим путем. Если очень грубо говорить, то вот такая штука $\sum w_i \Rightarrow W$

How to

На обходах я затронул тему поиска кратчайшего пути с помощью BFS и DFS со своими ограничениями.

Там было видно, что асимптотика времени работы крайне не оптимальна.

Возьмем последний пример с [Graph traversal, Pt. 2](#)



В этом графе весовой характеристикой ребра называется стоимость проезда по дороге из одного стока в другой. Задача остается той же: *найти самый дешевый путь между заданными стоками.*

План решения следующий:

- Начнем с заданной вершины
- Возьмем ближайшую к ней вершину
- Для каждой из соседних вершин:
 - Обновим расстояние до нее, если можно пройти более коротким путем и положим её в очередь
- Повторим первые 3 пункта

```
int getCheapestPath(int from, int to){
    auto ans = [].fill(-inf)
    ans[from] = 0

    var priorityQueue = []
    // положим пару (u, w), очередь
упорядочена
    // по значению w в этой паре
    priorityQueue.enqueue({0, from})

    while priorityQueue is not empty{
        (weight, vertex) =
priorityQueue.dequeue()

        // путь до вершины уже
оптимальный

        if (ans[vertex] < weight)
            continue;

        for(auto u : g[vertex]){
            auto consideredWeight =
u.weight;

            auto adjacentVertex =
u.vertex;

            // если рассматриваемый
путь короче

            // чем уже посчитанный
            if (ans[vertex] +
consideredWeight < ans[adjacentVertex]){

                // улучшим значение
                ans[adjacentVertex] =
                    ans[vertex] +
```

```

consideredWeight

                                // потом пересчитаем
расстояния

                                // для соседних вершин
priorityQueue.enqueue(
                                {

ans[adjacentVertex],

adjacentVertex

                                })
                                }
                                }
                                }

return ans[to]
}

```

Это классическая реализация алгоритма Дейкстры, который позволяет решить поставленную задачу.

Поскольку здесь используется приоритетная очередь, время получения ближайшей вершины происходит за $O(\log V)$ и так для каждой вершины, а релаксация происходит для всех ребер, то общее время работы составляет $O(V \log V + E)$, что намного лучше чем в случае BFS.

Concerns 🤔

Графы бывают разные и поиск кратчайшего пути должен зависеть от вводных данных.

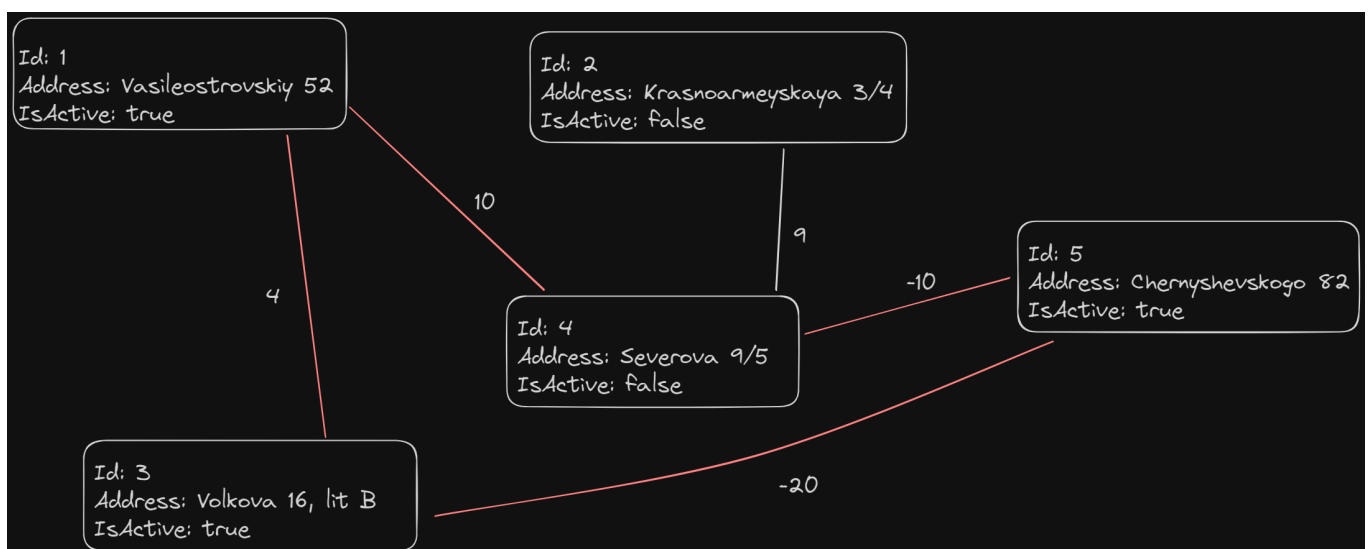
Например, для реализации представленной выше используется приоритетная очередь, но что если граф близок к полному? В таком случае асимптотика будет выглядеть примерно так: $O(V^2 * \log V)$.

Основное время работы алгоритма складывается из релаксации и получения ближайшей вершины.

А релаксация и получение ближайшей вершины напрямую зависят от контейнера, который используется в алгоритме.

Restrictions 🙅

Проблема Дейкстры такая же как и у BFS в подобной задаче.



В случае, когда в графе есть цикл отрицательного веса, Дейкстра всегда будет пытаться

релаксировать эти ребра.

distances

| | | | | |
|---|-----|-----|-----|-----|
| 1 | 2 | 3 | 4 | 5 |
| 0 | inf | inf | inf | inf |

id: 1 -> id: 4
inf > 0 + 10

| | | | | |
|---|-----|-----|----|-----|
| 1 | 2 | 3 | 4 | 5 |
| 0 | inf | inf | 10 | inf |

id: 4 -> id: 5
inf > 10 + (-10)

| | | | | |
|---|-----|-----|----|---|
| 1 | 2 | 3 | 4 | 5 |
| 0 | inf | inf | 10 | 0 |

id: 5 -> id: 3
inf > 0 + (-20)

| | | | | |
|---|-----|-----|----|---|
| 1 | 2 | 3 | 4 | 5 |
| 0 | inf | -20 | 10 | 0 |

id: 3 -> id: 1
inf > -20 + 4

| | | | | |
|-----|-----|-----|----|---|
| 1 | 2 | 3 | 4 | 5 |
| -16 | inf | -20 | 10 | 0 |

id: 1 -> id: 4
10 > -16 + 10

| | | | | |
|-----|-----|-----|----|---|
| 1 | 2 | 3 | 4 | 5 |
| -16 | inf | -20 | -6 | 0 |

"кратчайший" путь начал перезаписываться,
потому что имеется цикл отрицательного веса

В решение этой проблемы поможет другой [алгоритм](#)

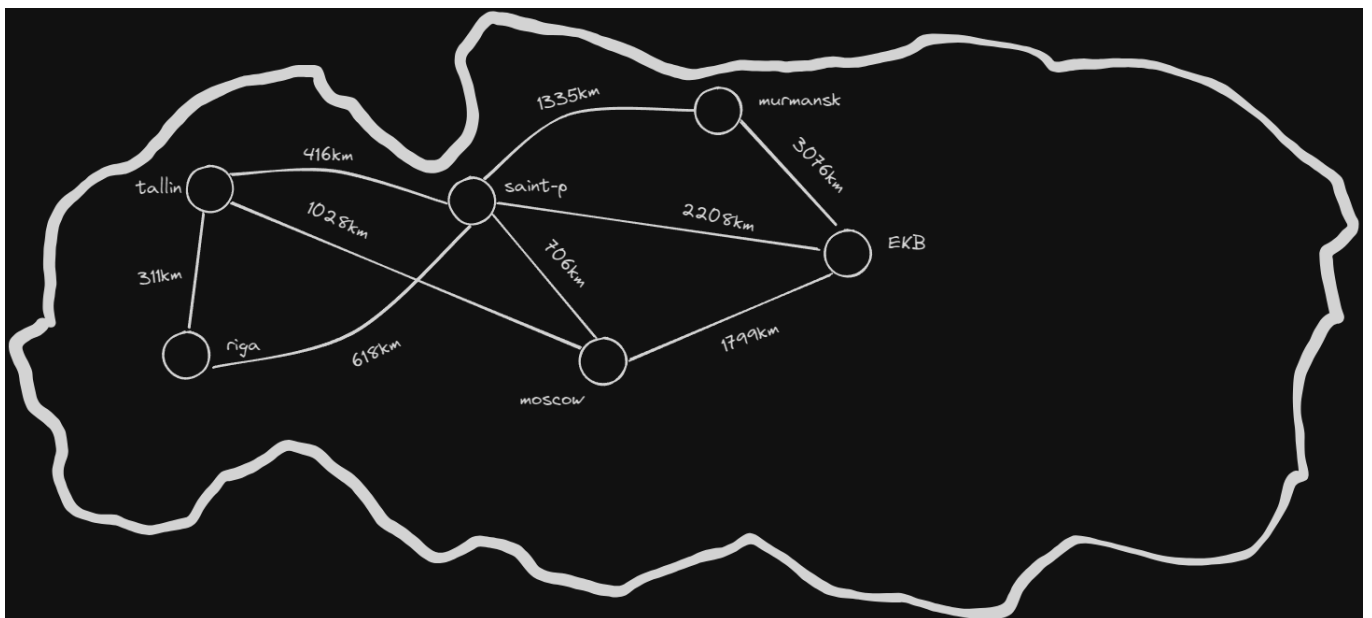
Heuristics

Одной из эвристик алгоритма Дейкстры является алгоритм A* (A-Star). Поскольку является этой эвристика, то она перенимает все ограничения.

Идеей алгоритма введение дополнительной функции, которая будет позволять Дейкстре понимать, хороший сейчас рассматривается путь или нет.

Рассмотрим пример, который позволит понять, как изменится отношение к графу, при добавлении этой функции.

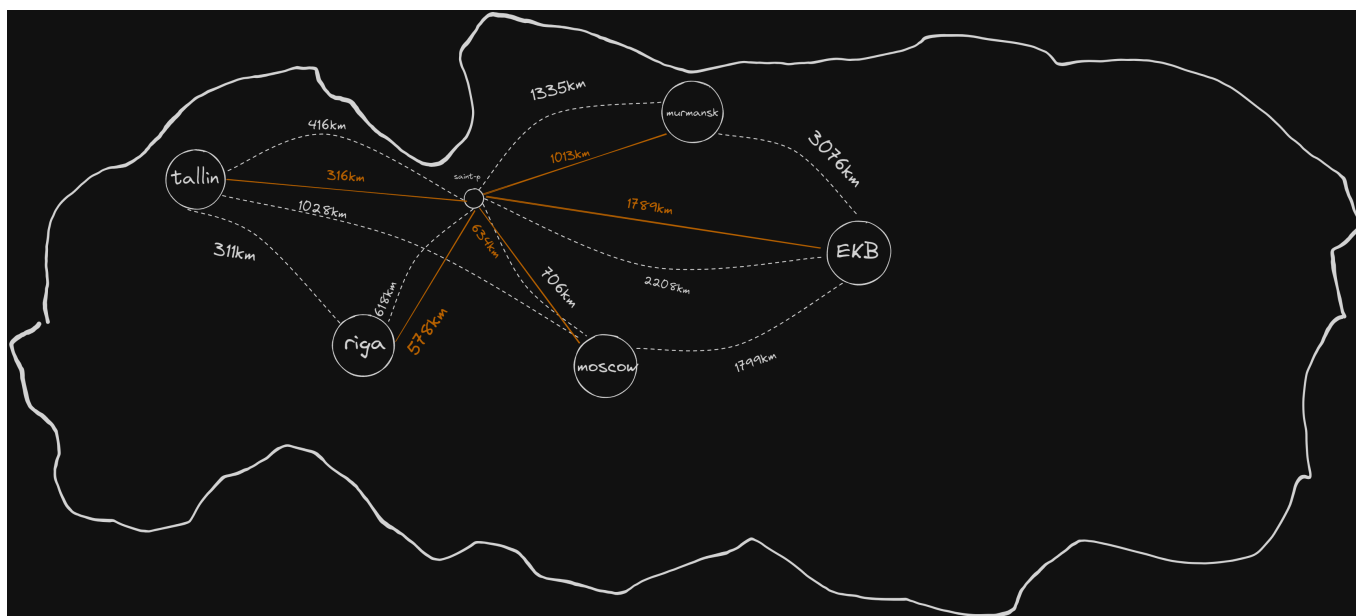
Пусть исходный граф отображает некоторую окрестность Санкт-Петербурга:



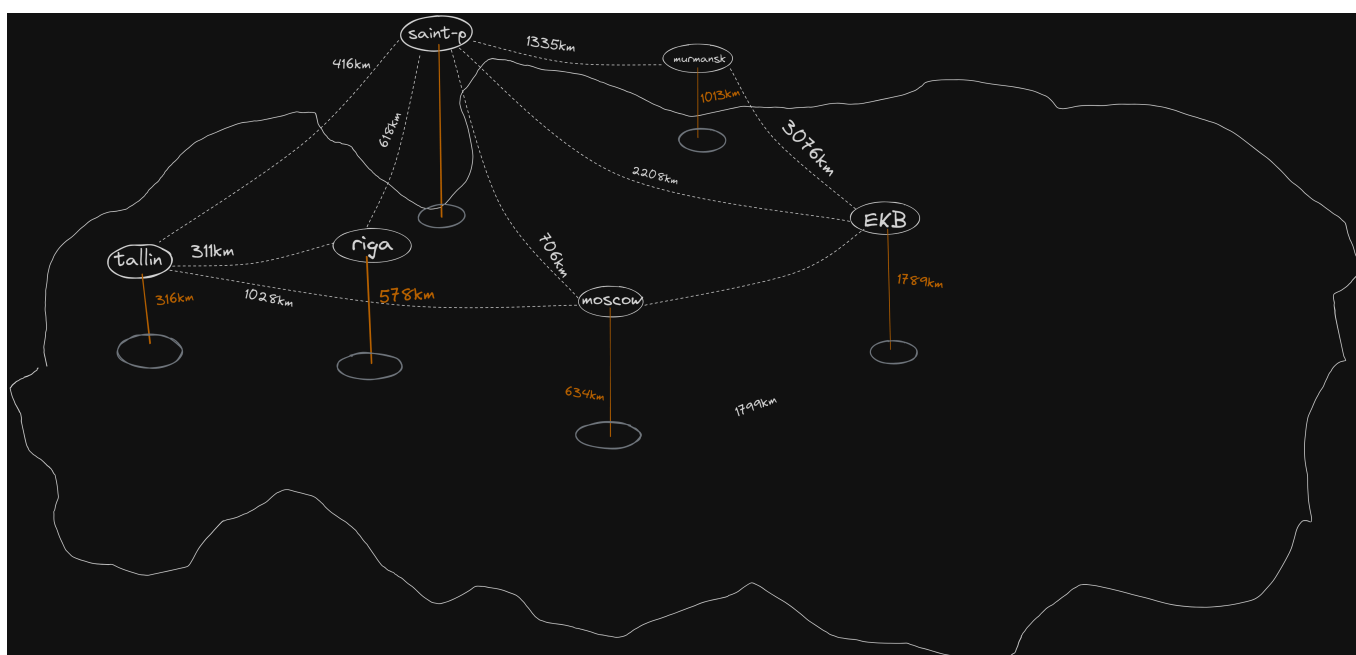
Ребра в графе определяют расстояние между городами. Задача — до боли знакома:

найти кратчайший путь до из некоторого стока в Санкт-Петербурге в некоторый сток в Екатеринбурге

Введем функцию-потенциал для каждого города: $h(x) \in \mathbb{Z}$, которая определяет **прямое** расстояние между Saint-P и городом x , граф для удобства перерисую в двух вариантах.



Я изобразил обычные пути пунктирной белой линией, а прямые пути сплошной оранжевой. На таком графе, должно быть, трудно разобраться не так ли? А если я перерисую ещё раз:



Потенциал возвышает города на карте как бы "подсказывая" куда нужно идти. Очень похоже на метод градиентного спуска, но им, конечно, не является.

При этом на функцию-потенциал возлагается ограничение: $h(x) \leq d(x, y) + h(y)$, что вполне логично, поскольку приоритет обхода иначе будет некорректен.

План алгоритма:

- Посмотрим на соседние вершины
- Рассматриваем их в порядке возрастания потенциала
 - Если можно прорелаксировать путь до u , то пересчитаем также и потенциалы

В остальном делаем все как в Дейкстре — ведем себя как ~~дебилы~~ программисты.

В самой Дейкстре меняется только одна строка

```
int getShortestPath(int from, int to){
    auto ans = [].fill(-inf)
    ans[from] = 0

    auto potentials = [].fill(inf) //
добавим это
    potentials[from] = h(from) // это

    var priorityQueue = []
    priorityQueue.enqueue( // поменяем это
    {
        potentials[from],
        ans[from],
        from
    })

    while priorityQueue is not empty{
        (potential, weight, vertex) =
priorityQueue.dequeue()

        if (ans[vertex] < weight)
```



```
        return ans[to]
    }
```

Negative cycles

Циклы отрицательного веса это большая проблема для Дейкстры. Алгоритм, который допускает циклы отрицательного веса — алгоритм Беллмана-Форда.

План поиска циклов отрицательного веса и вывода кратчайшего пути:

- Переберем все вершины
- Если путь до u можно упростить, сделаем это

Для удобства граф можно хранить как вектор ребер:

(u, v, w) :

```
(int[], int?) getShortestPath(int from, int to)
{
    distances = [].fill(inf)
    distances[from] = 0

    int startOfCycle;
    for (int i = 0; i < quantityOfVertices;
++i)
    {
        startOfCycle = -1;
        for (auto e : dg)
```

```

        {
            if (distances[e.u] + e.w <
distances[e.v])
            {
                distances[e.v] = distances[e.u]
+ e.w;

                path[e.v] = e.u;
                startOfCycle = e.u;
            }
        }
    }

    if (startOfCycle == -1)
    {
        return (distances, distances[to]);
    }

    cycle = [];

    for (int i = 0; i < quantityOfVertices;
++i)
    {
        startOfCycle = path[startOfCycle];
    }

    for (int i = startOfCycle;; i = path[i])
    {
        cycle.push(i);

        if (i == startOfCycle && cycle.size() >
1)
            break;
    }

```

```
    return (cycle, null)
}
```

Кратчайший путь в графе не может содержать больше $V - 1$ вершины, иначе мы соединим все вершины ребрами, а значит получим цикл. Если это не совсем очевидно, то вспомните как связаны ребра в графе с вершинами.

Если и тут неочевидно, то попробуйте нарисовать граф на $E = V$

Поэтому если выполнить цикл ровно V раз, то мы сможем либо найти цикл, либо не сильно проигрывая по времени получить кратчайший путь.

Время работы очевидно, благодаря тому, как я переписал граф: $O(V * E)$

DP with graphs

Формально, алгоритм Беллмана-Форда использует матрицу дп: A_{ij} — длина кратчайшего пути из s в i , содержащего не более j ребер. Просто эту матрицу в прошлом примере я упростил.

Проблема Беллмана-Форда заключается в том, что, хоть он и находит отрицательные циклы и кратчайшие пути, время работы будет не самым оптимальным для графов с большим количеством ребер (много большим числа вершин).

Поэтому самый тупой алгоритм на сегодня — алгоритм Флойда-Уоршелла.

План поиска отрицательного цикла и поиска кратчайшего пути:

- Переберем все тройки вершин
- Будем релаксировать расстояние между двумя вершинами и смежной вершиной
- Поскольку любой граф можно рассмотреть тройками (иногда совпадающими), то мы прорелаксируем все ребра графа

```
(int[], int?) getShortestPath(int from, int to)
{
    distances = [[]].foreach(u => // для
каждой u
    {
        // запишем расстояния до u
        u.foreach(w => w = g[u][v])
    })

    path = [[]].foreach(u => // каждую u
    {
        // запишем как родителя для u
        u.foreach(parent => parent = u)
    })

    for i in g.size
        for j in g.size
            for k in g.size
                if
(distances[i][j] > distances[i][k] +
```

```

distances[k][j]){

distances[i][j] = distances[i][k] +
distances[k][j]

path[i]

[j] = path[i][k]

}

if any(distances[i][i] < 0){
    cycle = [from]
    cur = from
    while cur != to{
        cur = path[cur][from]
        cycle.push(cur)
    }

    return (cycle, null)
}

ans = [from]
cur = from
while (cur != to){
    cur = path[cur][from]
    ans.push(cur)
}

return (ans, distances[from][to])
}

```

Единственное, что отсюда непонятно, почему проверяется диагональ и, если на ней есть отрицательный значения, то есть и цикл.

Вообще по предусловию $distances[i][i]$ всегда 0. При этом мы всегда рассматриваем все ребра, значит, если путь $[i, i_1, \dots, i]$ был прорелаксирован — уменьшился, то на этом пути были отрицательные ребра, а значит это отрицательный цикл. Достаточно просто да ?

Время работы $O(V^3)$

What's next

Кратчайшие пути — это, конечно, классно..

Мы рассматривали примеры со стоками и логистикой Yandex. А что, если пойти в Yandex.Taxi?

Вы, должно быть видели, как таксисту приходят уведомления о заказах по всему городу, но у него всегда есть возможность выбрать какой-то район или какой-то конкретный заказ.

Представьте граф города (вершины — перекрестки, ребра — дороги). Все возможные заказы, поступающие таксисту в какой-то момент времени, можно ассоциировать с перекрестком для упрощения.

Образуется подобие сети, в центре которой находится таксист.

Система предоставляет возможность таксисту выбрать ближайший к нему заказ. Как можно

с моделировать такое поведение с помощью алгоритмов ?..