1 Алгоритм Деикстры

1.1 Описание алгоритма, доказательство корректности

Алгоритм Деикстры используется в произвольном графе для нахождения кратчайшего расстояния от вершины S до всех прочих вершин графа. **Важное условие работы алгоритма**: в графе должны отсутствовать циклы отрицательного веса. Алгоритм работает как для ориентированных, так и для не ориентированных графов.

Алгоритм Деикстры является оптимизацией алгоритма **bfs 0-k**. На самом деле я не знаю, какой алгоритм появился раньше, но так как мы изучили **bfs 0-k** раньше, то будем шагать от этого. Вспоминаем, как работает алгоритм **bfs 0-k**: создадим очереди вершин с расстоянием x и будем рассматривать их в порядке увеличения расстояния. Такой алгоритм работает за $O(k(n+m))$, ведь мы максимальный путь в графе равен $(n-1)k$.

Однако, много расстояний мы просто не посещаем, ведь каждую вершину мы посетим один раз, и часто мы проходим по такой длине пути x, что вершин с такой длиной просто нет. Всего вершин n, а расстояний $(n-1)k$, их намного больше. Алгоритм Деикстры говорит: давайте просто возьмем любую вершину с минимальным расстоянием и посмотрим все ребра из нее. Это то же самое, что и **bfs 0-k**, но при этом мы не посещаем несуществующие расстояние, мы сразу берем вершину с минимальным расстоянием. По факту, в алгоритме **bfs** мы делаем то же самое, мы всегда берем вершину с минимальным расстоянием и смотрим все пути из нее, просто в алгоритме **bfs** следующее расстояние всегда равно $x+1$, а здесь оно может быть любым, потому что ребро может иметь любой вес.

Доказательство: предположим, что алгоритм Деикстры нашел расстояние для первых k вершин правильно, тогда давайте докажем, что для следующей вершины всегда находится оптимальное расстояние. Предположим, что алгоритм нашел не самое кратчайшее расстояние W, а расстояние d_x. Тогда существует какой-то путь до текущей вершины x с расстоянием W. Рассмотрим последнюю помеченную вершину на этом пути, пусть это вершина y (такая вершина всегда есть). Но $d_y < W$, тогда вершина y не может быть последней помеченной вершиной, противоречие (иначе наш алгоритм сначала проверил все ребра из вершины y и пометил вершину z, так как она лежит на оптимальном пути до y, то $d_z \le d_y$).

1.2 Как закодить

Рассмотрим, как написать алгоритм Деикстры, чтобы найти расстояние до всех вершин в графе от вершины S. Обратите внимание, здесь ребро задано не только парой вершин, но еще и весом, поэтому для хранения графа мы создадим структуру:

Пример 1 (файл struct)

```
struct Edge{
    int y;
    long long w;
};
vector<vector<Edge>>e;

...

int main() {
    ...
    e = vector<vector<Edge>>(n+1);
    for(int i = 1; i <= m; i++) {
        long long x,y,w;
        cin >> x >> y >> w;
        e[x].pb({y,w});
        e[y].pb({x,w});
    }
    ...
}
```

Конец примера 1

Давайте выпишем шаги алгоритма:

I Выбрать нерасмотренную вершину с минимальным расстоянием

II Рассмотреть все ребра из найденной вершины

Понятно, что можно выбрать нерасмотренную вершину просто найдя минимум в массиве d (d — массив расстояний). Но можно воспользоваться структурами данных, например, двоичной кучей для поиска минимума в массиве. Мы рассмотрим также реализацию через красно-черное дерево.

Итак, приведем сам код через *set*:

Пример 2 (файл algo)

```
#include <bits/stdc++.h>
using namespace std;
struct Edge{
    int y;
    long long w;
};
const long long INF = 1'000'000'000'000'000'000;
vector<vector<Edge>>e;
int n,m;

void dij() {
    vector<long long>d(n+1,INF);
    int S = 1;
    set<pair<long long, int>>t;
    t.insert({d[S], S});
    while(t.size()) {
        auto [_, x] = *t.begin();
        t.erase(t.begin());
        for(auto&[y,w]:e[x]) {
            if(d[y] > d[x] + w) {
                t.erase({d[y], y});
                d[y] = d[x] + w;
                t.insert({d[y], y});
            }
        }
    }
}
```

Конец примера 2

Как только мы нашли лучшее расстояние до вершины, мы удаляем ее из сета и добавляем новое, обновленное расстояние. Можно написать реализацию и по-другому, которая оказывается даже быстрее. Видите ли, удалять из сета не очень легко, хоть это и делается за $O(\log n)$. Можно использовать двоичную кучу, которая умеет доставать максимум за $O(\log n)$. Видите ли, нам нужно доставать минимум, но мы можем складывать значения со знаком минус.

Пример 3 (файл algo2)

```
#include <bits/stdc++.h>
using namespace std;
struct Edge{
    int y;
    long long w;
};
const long long INF = 1'000'000'000'000'000'000;
vector<vector<Edge>>e;
int n,m;

void dij() {
    vector<long long>d(n+1,INF);
    int S = 1;
    priority_queue<pair<long long, int>>qu;
    qu.push({-d[S], S});
    while(qu.size()) {
        auto [cur_dist, x] = qu.top();
        qu.pop();
```

```
        cur_dist = -cur_dist;
        if(d[x] < cur_dist) continue;
        for(auto&[y,w]:e[x]) {
            if(d[y] > d[x] + w) {
                d[y] = d[x] + w;
                qu.push({-d[y], y});
            }
        }
    }
}
```

Конец примера 3

В среднем, решение через очередь работает быстрее.

Обратите внимание. Максимальное расстояние может достигнуть порядка nw, где обычно $n \approx 10^5, w \approx 10^9$, тогда 10^{14} не помещается в тип *int*, но помещается в тип *long long*, поэтому будьте внимательнее. Сохраняйте ответ в подходящем типе данных.

1.3 Итоговая асимптотика

Оба алгоритма достают для каждого из m ребер вершину за $\log n$, итого асимптотика $O(m \log n)$.

1.4 Восстановление ответа

Восстановить ответ в Алгоритме Деикстры так же просто, как и в **bfs**. Заведем массив p, где p_x — из какой вершины мы попали в вершину x по оптимальному пути, когда мы обновляем ответ для вершины x мы обновим значение p_x.

Пример 4 (файл algo_p)

```
#include <bits/stdc++.h>
using namespace std;
struct Edge{
    int y;
    long long w;
};
const long long INF = 1'000'000'000'000'000'000;
vector<vector<Edge>>e;
int n,m;

void dij() {
    vector<long long>d(n+1,INF);
    int S = 1;
    set<pair<long long, int>>t;
    vector<int>p(n+1, -1);
    t.insert({d[S], S});
    while(t.size()) {
        auto [_, x] = *t.begin();
        t.erase(t.begin());
        for(auto&[y,w]:e[x]) {
            if(d[y] > d[x] + w) {
                t.erase({d[y], y});
                d[y] = d[x] + w;
                p[y] = x;
                t.insert({d[y], y});
            }
        }
    }
    int T = n;
    vector<int>path;
    int x = T;
    while(x != -1) {
```

```
        path.push_back(x);
        x = p[x];
    }
    reverse(path.begin(), path.end());
}
```

Конец примера 4

1.5 Еще один пример применения алгоритма

Часто может встретиться похожая задача: есть расписание поездов, из города x поезд уходит в момент времени t_i и через w_i секунд добирается до города y_i. Таким образом, у нас есть взвешенный ориентированный граф, где есть ребра вида $(\text{откуда}, \text{куда}, t_i, w_i)$. Мы можем ждать в городах, например, если мы приехали во время 10 и поезд уходит в момент времени 16, то мы можем подождать до 16 и уехать на поезде. Нас просят найти минимальное время прибытия в город T из города S. Как решить задачу? Допустим, мы можем приехать в город в моменты времени: $10, 16, 25, 30, 60$. Но что нам выгоднее? Заметим, что чем раньше мы приедем в город, тем больше рейсов нам будет доступно. Например, если мы приедем в 30, то мы пропустим поезд, который ушел в 20 секунд, но если мы приедем раньше, то у нас больше шансов попасть на этот поезд. Поэтому нам просто нужно найти минимальный путь из S в T, но при этом по ребрам мы ходим немного по-другому. Если мы приедем в город x в момент d_x, то мы можем только рассматривать такие ребра, у которых $t_i \geq d_x$. Понимаете, как это написать? Здесь оставлю небольшую подсказку:

Пример 5 (файл algo_train)

```
struct Edge{
    int y;
    long long t;
    long long w;
};
...
for(auto&[y, t, w]:e[x]) {
    if(d[x] <= t && d[y] > t + w) {
        ...
    }
}
```

Конец примера 5

1.6 Подведение итогов

I Алгоритм Деикстры работает за $O(m \log n)$

II Используем алгоритм на взвешенных графах без циклов отрицательного веса

III Не используем алгоритм, когда веса равны 1, используем **bfs** вместо этого

IV Практикуемся в алгоритме Деикстры