Лекция 5.
Кратчайшие пути

179 Школа, Овчинников Андрей

26 октября 2022 г.

Аннотация

Перед тем, как вы прочитаете то, что написано ниже обращу внимание на то, что в данном конспекте могут быть допущены ошибки и опечатки, если вы находите подобное, то пишите мне в личку или группу в телеграмм с упоминанием (*то есть через* @). Буду рад, если вы поможете довести конспект до хорошего безошибочного состояния.

1 О теме

Мы рассмотрим одни из самых полезных, на мой взгляд, алгоритмов поиска кратчайших путей. Здесь не будет алгоритмов Форда-Беллмана и Флойда, так как они достаточно редко встречаются (по крайней мере пока что ...). Тем не менее, если возникает необходимость их использовать, то они по сути не заменимы, поэтому оставлю ссылку для дополнительного ознакомления с алгоритмами Форда-Беллмана и Флойда

2 BFS

Пусть дан взвешенный граф, построенный на n вершинах из m рёбрер (не важно ориентированный или нет, главнное что бы веса рёбер были $0 \leq w_i \leq k$, где k - известно заранее). Начнём с разбора более частных случаев.

2.1 Обычный BFS

Пусть $w_i = 1, \forall i$. Тогда найдём кратчайшее расстояние от заданной вершины до остальных. Расстояние до выбранной вершины 0. Будем поддерживать множество вершин, до которых уже посчитано минимальное расстояние (на старте это только изначальная вершина). На очередном шаге, выбираем ближайшую вершину, из тех, что являются *соседними* с множеством, и добавляем её в наше множество $\Rightarrow$ обновляем значения и множество вершин соседних с *минимальным* множеством.

2.1.1 Код

Реализуем оптимально. Используя факты озвученные на лекции.

```
void
bfs(int start_vertex, vector<int> &dist, vector<vector<int>> &graph)
{
    vector<bool> used(dist.size(), false);
    fill(dist.begin(), dist.end(), INT_MAX);

    used[start_vertex] = true;
    dist[start_vertex] = 0;

    deque<int> all;
    all.push_back(start_vertexr);

    while (!all.empty()) {
        int v = all[0];
        all.pop_front();

        for (auto to : graph[v]) {
            if (!used[to]) {
                used[to] = true;
                dist[to] = dist[v] + 1;
                all.push_back(to);
            }
        }
    }
}
```

2.1.2 Асимптотика

Заметим, что каждая вершина может побывать в очереди единожды (исходя из реализации). Для каждой вершины, мы перебираем все её соседей $\Rightarrow O(n + m)$.

2.2 0-1 BFS

Пусть $0 \leq w_i \leq 1$. Тогда при очередном обновлении нужно учитывать что ребро может быть как веса 0, так и веса 1. Соответственно, более лёгкие ребра имеют более высокий приоритет, так как им следует обновить своих соседей раньше. Воспользуемся тем, что в очереди одновременно могут находиться только вершины с расстоянием k и $k+1$ до стартовой вершины, и будем соседей по рёбрам нулевого веса ставить в начало очереди, так как там стоят такие же вершины с расстоянием k.

2.2.1 Код

Реализуем оптимально с учётом написанного выше.

```
void
bfs(int start_vertex, vector<int> &dist, vector<vector<pair<int, int>>> &graph)
{
    fill(dist.begin(), dist.end(), INT_MAX);
    dist[start_vertex] = 0;

    deque<int> q;
    q.push_back(start_vertexr);

    while (!q.empty()) {
        int v = q[0];
        q.pop_front();

        for (auto to : graph[v]) {
            if (dist[to.first] > dist[v] + to.second) {
                dist[to.first] = dist[v] + to.second;
                if (to.second == 0) {
                    all.push_front(to.first);
                } else {
                    all.push_back(to.first);
                }
            }
        }
    }
}
```

2.2.2 Асимптотика

Заметим, что каждая вершина может побывать в очереди на более 2-ух раз (если обновили по ребру веса - 0 и, если обновили по ребру веса - 1). Для каждой вершины мы пройдёмся по её ребрам не более двух раз $\Rightarrow O(2 * n + 2 * m) = O(n + m)$.

2.3 1-k BFS

Теперь перейдём к более общему случаю задачи, пусть $1 \leq w_i \leq k$. Заметим, что по сути в прошлой подзадаче, мы работали с двумя очередями, одна для вершин с расстоянием k, другая для - $k+1$ (иммитировали это с помощью *.push_front()*). Тогда давайте обобщим наш подход, а именно, так как веса в нашем случае веса бывают от 1 до $k+1$, то очевидно нам хватит $(k+1)$ очереди (этот факт более подробно пояснялся/будет прояснён на лекции).

2.3.1 Код

Код был позаимствован, с небезызвестного сайта codeforces из этой статьи.

```
void
bfs(int start_vertex, vector<int> &dist,
    vector<vector<pair<int, int>>> &graph, int max_w)
{
    fill(dist.begin(), dist.end(), INT_MAX);
    vector<bool> used(dist.size(), false);
    vector<deque<int>> q(max_w + 1);

    int current_q = 0, cnt_in_queue = 1;
    dist[start_vertex] = 0;
    q[current_q].push_back(start_vertex);

    while (0 < cnt_in_queue) {
        //пропускаем пустые очереди
        while (q[current_q % (max_w + 1)].empty()) {
            current_q++;
        }
        //пропускаем пустые очереди
        int v = q[current_q][0];
        q[current_q].pop_front();
        cnt_in_queue--;

        if (used[v]) {
            continue;
        } else {
            used[v] = true;
            for (auto e : graph[v]) {
                if (dist[e.first] > dist[v] + e.second) {
                    dist[e.first] = dist[v] + e.second;
                    q[dist[e.first] % (max_w + 1)].push_back(e.first);
                    cnt_in_queue++;
                }
            }
        }
    }
}
```

2.3.2 Асимптотика

Заметим, что каждая аершина может побывать в каждой очереди единжды (так как может быть обновлена по рёбрам веса: $1, 2, \ldots, k$). Соответственно благодаря массиву *used* по соседям текущей вершины мы пройдёмся лишь единожды, окуда $\Rightarrow O(k * n + m)$

3 Дейкстра

Рассмотрим более общую задачу. Дан граф на n вершинах и m рёбрах. У каждого ребра есть вес, причём проивольный (ну почти ☺) положительный.

3.1 Дейкстра за $O(n^2)$

Аналогично предыдущему алгоритму, будем поддерживать множество вершин, до которых посчитано кратчайшее расстояние. И в очередной момент времени искать ближайшую вершину **нележащую** в этом множестве. Добавим её в множество и пересчитаем её соседей.

3.1.1 Код

Реализуем описанный выше алгоритм.

```
void
dijkstra(int start_vertex, vector<int> &dist, vector<vector<pair<int, int>>> &graph)
{
    fill(dist.begin(), dist.end(), INT_MAX);
    dist[start_vertex] = 0;

    vector<bool> used(dist.size(), false);

    for (int i = 0; i < (int)dist.size(); ++i) {
        //ищем минимальную не в нашем множестве
        int current_v = -1;
        for (int u = 0; u < (int)dist.size(); ++u) {
            if ((current_v == -1 || dist[current_v] > dist[u]) && !used[u]) {
                current_v = u;
            }
        }
        if (dist[current_v] == INT_MAX) {
            break; //если граф не связен, то эта проверка необходима
        }          //чтобы не получить переполнения прии пересчёте
                   //соседей => мусор в dist

        used[current_v] = true; //добавляем current_v в множество
        //обновляем соседей
        for (auto e : graph[current_v]) {
            if (!used[e.first]) {
                dist[e.first] = min(dist[e.first], dist[current_v] + e.second);
            }
        }
    }
}
```

3.2 Дейкстра за $O(m * log_2(n))$

Мы знаем множество структур данных, давайте воспользуемся ими, для того чтобы ускорить самую медленную часть (иногда не самую медленную) предыдущего подхода. А именно, внутренний цикл поиска кандидата работает за весь алгоритм $O(n^2)$, а вот обновление соседей за весь алгоритм отработает $O(m)$ раз $\Rightarrow$ если граф разреженный (не полный), то тогда мы будем сильно проигрывать по времени работы на цикле поиска кандидата. Используем *set* для того, чтобы искать кандидата за $O(leg_2(v))$. Однако мы усложним обновление соседей, так как теперь будет необходимо удалять вершину из *set'a* и добавлять заново, если расстояние до неё обновилось $\Rightarrow O(m * (log_2(n) + log_2(n))) = O(m * log_2(n))$

3.2.1 Код

По описаннуму выше алгоритму напишем код.

```
void
dijkstra(int start_vertex, vector<int> &dist, vector<vector<pair<int, int>>> &graph)
{
    fill(dist.begin(), dist.end(), INT_MAX);
    dist[start_vertex] = 0;

    vector<bool> used(dist.size(), false);
    set<pair<int, int>> q;
    q.insert({dist[start_vertex], start_vertex});

    while (!q.empty()) {
        int v = (*q.begin()).second;
        q.erase(q.begin());
        used[v] = true;

        for (auto u : graph[v]) {
            if (!used[u.first] && dist[u.first] > dist[v] + u.second) {
                q.erase({dist[u.first], u.first});
                dist[u.first] = dist[v] + u.second;
                q.insert({dist[u.first], u.first});
            }
        }
    }
}
```

3.3 Об использовании

Важно заметить, что Дейкстра за $O(n^2)$ вовсе не бесполезна, как может показаться неопытному читателю, так как если вам дан полный граф, то

$$m \cdot log_2 n = n^2 \cdot log_2 n > n^2 \tag{1}$$

Следовательно, в случае полного графа, Дейкстра за $O(n^2)$ работает лучше.