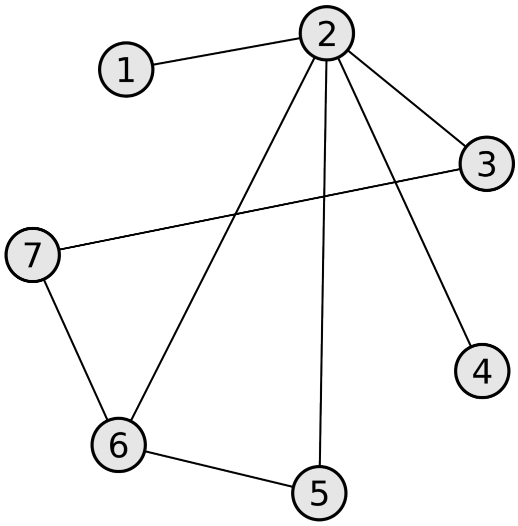
Графы: DFS. BFS.

Сапожников Денис

Contents

1 Хранение графов

Чтобы решать задачки на графы, вы рисуете их на бумажке в виде вершин и рёбер.



Но как же сохранить графы в памяти? Существуют два подхода.

1.1 Матрица смежности

Вы можете сознать матрицу $n \times n$, состоящую из 0 и 1, где 1 в позиции (i, j) обозначает наличие ребра из i-й вершины в j-ю. У данного подхода есть масса преимуществ:

1. Простота. Действительно, заполнить матрицу очень просто, а хранить вам нужно лишь двумерный массив.

2. Легко проверять наличие ребра между любыми двумя вершинами.

3. Легко делать граф ориентированным/неориентированным, взвешенным/не взвешенным (для взвешенного графа можно хранить не 1 при наличии ребра, а вес ребра между вершинами).

Например, хранение неориентированного взвешенного графа будет следующим:

```
int n, m; // vertexes, edges
cin >> n >> m;
vector<vector<int>> adj(n, vector<int>(n));
for (int i = 0; i < m; ++i) {
  int a, b, w;
  cin >> a >> b >> w; // weight of edge between a and b is w
  adj[a - 1][b - 1] = adj[b - 1][a - 1] = w;
}
```

Однако, есть очень большой недостаток: если в графе 10^5 вершин и 10^5 рёбер, то вам придется сохранить таблицу размера $10^5 \times 10^5$, при этом единицы в такой таблице будет очень мало. Такие графы называются разреженными и очень часто в задачах даны именно разреженные графы. Как хранить разреженные графы?

1.2 Список смежности

Вместо того, чтобы хранить всю матрицу смежности давайте для каждой вершины хранить список её соседей — список смежности.

Такой подход, очевидно, занимает $O(n + m)$ памяти, где n – количество вершин, m – количество рёбер.

Но есть и пара проблем:

1. Неудобно проверять наличие ребра между парой вершин. Для этого придется хранить не список смежных вершин, а множество смежных вершин, что увеличивает асимптотику. Благо, в задачах почти никогда не надо проверять наличие ребра между конкретными двумя вершинами.

2. Не очень удобно хранить веса рёбер: вместе с соседом вершины придется хранить ещё и вес ребра (то есть хранить пару).

То есть теперь для хранения неориентированного взвешенного графа вам придется написать следующий код:

```cpp
int n, m;
cin >> n >> m;
vector<vector<pair<int, int>>> gr(n); // from - { {to[1], w[1]}, ...}
for (int i = 0; i < m; ++i) {
  int a, b, w;
  cin >> a >> b >> w;
  --a, --b;
  gr[a].push_back({ b, w });
  gr[b].push_back({ a, w });
}
```

2 DFS

2.1 Алгоритм

Скорее всего, все уже знакомы с этим алгоритмом обхода графа. Напомню, что этот алгоритм «идёт, пока может», то есть:

```
bool used[N];
void dfs(int v) {
  used[v] = true;
  for (int u : gr[v])
    if (!used[u])
      dfs(u);
}
```

2.2 Красим в три цвета

Казалось бы, говорить об этих 7 строках кода нечего, но на самом деле тут есть потаенный смысл. Пусть ещё непосещённые вершины будут белыми, серыми те, которые лежат в стеке вызова, а чёрные — те, которые мы посетили и удалили из стека.

То есть:

```
enum { WHITE, GREY, BLACK };
int used[N];
void dfs(int v) {
  used[v] = GREY;
    for (int u : gr[v])
      if (used[u] == WHITE)
        dfs(u);
  used[v] = BLACK;
}
```

Лемма 1. *Не существует такого момента выполнения поиска в глубину, в который бы существовало ребро из чёрной вершины в белую.*

Proof. Пусть в процессе выполнения процедуры dfs нашлось ребро из чёрной вершины v в белую вершину u. Рассмотрим момент времени, когда мы запустили $dfs(v)$. В этот момент вершина v была перекрашена из белого в серый, а вершина u была белая. Далее в ходе выполнения алгоритма будет запущен $dfs(u)$, поскольку обход в глубину обязан посетить все белые вершины, в которые есть ребро из v. По алгоритму вершина v будет покрашена в чёрный цвет тогда, когда завершится обход всех вершин, достижимых из неё по одному ребру, кроме тех, что были рассмотрены раньше неё. Таким образом, вершина v может стать чёрной только тогда, когда dfs выйдет из вершины u, и она будет покрашена в чёрный цвет. Получаем противоречие. □

2.3 Лемма о белом пути

Лемма 2 (о белом пути)**.** *Пусть дан граф G. Запустим $dfs(G)$. Остановим выполнение процедуры dfs от какой-то вершины u графа G в тот момент, когда вершина u была выкрашена в серый цвет (назовём его первым моментом времени). Заметим, что в данный момент в графе G есть как белые, так и чёрные, и серые вершины. Продолжим выполнение процедуры $dfs(u)$ до того момента, когда вершина u станет чёрной (второй момент времени). Тогда* *вершины графа $G \setminus u$, бывшие чёрными и серыми в первый момент времени, не поменяют свой цвет ко второму моменту времени, а белые вершины либо останутся белыми, либо станут чёрными, причём чёрными станут те, что были достижимы от вершины u по белым путям.*

Proof. Чёрные вершины останутся чёрными, потому что цвет может меняться только по схеме белый $\to$ серый $\to$ чёрный. Серые останутся серыми, потому что они лежат в стеке рекурсии и там и останутся.

Далее докажем два факта:

Утверждение. *Если вершина была достижима по белому пути в первый момент времени, то она стала чёрной ко второму моменту времени.*

Proof. Если вершина v была достижима по белому пути из u, но осталась белой, это значит, что во второй момент времени на пути из u в v встретится ребро из черной вершины в белую, чего не может быть по лемме, доказанной выше. □

Утверждение. *Если вершина стала чёрной ко второму моменту времени, то она была достижима по белому пути в первый момент времени.*

Proof. Рассмотрим момент, когда вершина v стала чёрной: в этот момент существует серый путь из u в v, а это значит, что в первый момент времени существовал белый путь из u в v, что и требовалось доказать. □

Отсюда следует, что если вершина была перекрашена из белой в чёрную, то она была достижима по белому пути, и что если вершина как была, так и осталась белой, она не была достижима по белому пути, что и требовалось доказать. □

2.4 Поиск цикла

Утверждение. *В ориентированном графе существует цикл тогда и только тогда, когда при обходе dfs-ом найдется момент времени, когда мы посмотрим из серой вершины в серую.*

```
bool has_cycle(int v) { // return true if has cycle
  used[v] = GRAY;
  for (int u : gr[v]) {
    if (used[u] == GRAY || used[u] == WHITE && dfs(u)) {
      return true;
    }
  }
  used[v] = BLACK;
  return false;
}
```

Подумайте, как можно восстановить этот цикл.

2.5 Прямые и обратные рёбра

Определение. Назовём ребро **прямым**, если мы прошли по нему во время обхода dfs.

Определение. Прямые ребра образуют **дерево обхода** dfs.

Определение. Ребра (u, v), соединяющие вершину u с её предком v в дереве обхода в глубину назовём **обратными рёбрами** (для неориентированного графа предок должен быть не родителем, так как иначе ребро будет являться ребром дерева).

Определение. Все остальные ребра назовём **перекрёстными рёбрами**.

Задача. Докажите, что при обходе неориентированного графа в глубину **не существует перекрёстных рёбер**.

3 BFS

3.1 Обычный BFS

Наверняка многие знают, что это, но напомнить стоит. Типичная задача — задача о коне. Дан шахматный конь, нужно сказать, за какое минимальное количество ходов он доберётся из стартовой клетки в заданную. Для этого мы заведём структуру данных queue — очередь, которая умеет в 2 типа запросов, которые равны обычной очереди в супермаркете.

- pop — кассир обслужил первого в очереди человека, который после сразу ушёл.
- push — пришёл новый человек в конец очереди.

С помощью такой структуры мы можем легко решить задачу.

Будем поддерживать полуинвариант: в очереди сначала лежат клетки на расстоянии k от заданной, а затем на расстоянии $k+1$. Берём первую клетку из очереди (она на расстоянии k) и добавляем в конец очереди всех её непосещённых соседей (действительно, они на расстоянии $k+1$ и полуинвариант сохранился). Так обходим всё поле и решаем задачу сразу для любой финальной клетки. Сложность решения — $O(nm)$

3.2 1-k BFS

Бывает такое, что в задаче нужно найти кратчайшее расстояние от s до всех остальных вершин в неориентированном взвешенном графе, и при этом веса рёбер — целые числа от 1 до k. В таком случае иногда быстрее будет работать 1-k BFS.

Так как веса рёбер ограниченны, то максимальный по весу путь равен $(n-1)k$. Заведём $(n-1)k$ очередей. Каждая очередь будет означать список вершин на этом расстоянии. Изначально в 0-й очереди находится стартовая вершина. Кроме того, мы поддерживаем вершин, расстояние до которых уже точно посчитано.

Пусть мы обработали все вершины на расстоянии меньше L. Тогда будем вытаскивать из очереди все вершины на расстоянии L, если расстояние до неё уже посчитано, то пропускаем вершину иначе расстояние до вершины строго равно L (идея аналогична Дейкстре). От всех вершин v, расстояние до которых равно L, добавим в очередь всех соседей u с ребром веса w в очередь номер $L+w$.

Оптимизация: заметим, что в каждый момент времени у нас используется не больше $k+1$ очереди, поэтому мы можем создать всего $k+1$ очередь и засовывать новые вершины в очередь номер $(L+w) \bmod (k+1)$.

Время работы алгоритма — $O(nk+m)$ и $O(n+k)$ памяти.

```
vector<queue<int>> q(k + 1);
q[0].push(st);
vector<int> dist(n, -1);
for (int i = 0; i < n * k; ++i) {
  int nq = i % (k + 1);
  while(q[nq].size()) {
    int v = q[nq].front();
    q[nq].pop_front();

    if (dist[v] == -1)
      dist[v] = i;
    else
      continue;

```

```
    for (auto [u, w] : gr[v])
      if (dist[u] == -1)
        q[(nq + w) % (k + 1)].push(u);
  }
}
```

Частным случаем 1-k BFS является 0-1 bfs, он пишется ещё проще и более распространён.