Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

Отчет по блоку №4 (яндекс контест)

По Алгоритмам и структурам данных

Выполнил:

Ступин Тимур Русланович

Группа № P3108

Поток № 1.3

Преподаватель:

Тараканов Денис Сергеевич

Санкт-Петербург 2025

Содержание

Задача 1 Цивилизация **3**

Задача 2 Свинки-копилки **7**

Задача 3 Долой списывание! **9**

Задача 4 Авиаперелёты **11**

Задача 1 Цивилизация

Решение

Отметим, что задача является классической задачей о поиске минимального пути между двумя вершинами в графе. Очевидным решением является алгоритм Дейкстры, однако в данной задаче можно заметить, что веса рёбер имеют очень ограниченный диапазон значений: `1` или `2`. Это позволяет решить данную задачу иначе: используя `1-k` bfs.

Идея данного алгоритма в том, чтобы имея граф, в котором рёбра могу иметь вес от `1` до `k` (в данном случае `k=2`), преобразовать его к графу с рёбрами единичной длинны, путём добавления фиктивных вершин. После чего на новом графе запустить обычный bfs, который очевидно позволяет найти кратчайший путь между двумя вершинами.

Естественно, реальное добавление вершин неэффективно по памяти, поэтому реализация алгоритма немного отлична от его идеи. Вместо того чтобы явно добавлять фиктивные вершины, можно заметить, что bfs на преобразованном графе обрабатывает вершины в порядке возрастания расстояния. Поэтому можно просто завести отдельную очередь для каждого расстояния и добавлять вершину в очередь, соответствующую расстоянию её текущему. Но поскольку расстояния растут линейно, а рёбра имеют вес не больше `k`, достаточно хранить `k+1` очередей. Само же минимальное расстояние будет хранится в массиве `d`, в котором значение в `i`-й ячейке является минимальным расстоянием от стартовой вершины до вершины `i`. Также понадобится вспомогательный массив `used` (для хранения посещённых вершин) и `history` (для восстановления пути). Сам алгоритм заключается в следующем:

Будем поддерживать указатель на текущую очередь (`q_ptr`) и счётчик вершин, которые лежат во всех очередях в данный момент (`cnt`) для того что понять когда пора остановиться.

Пока в очередях лежит хотя бы один элемент будем выполнять следующие действия:

1. Из первой непустой очереди выберем первый элемент `u`. Это будет текущая рассматриваемая вершина.

2. Если эта вершина уже использована, то просто переходит к следующей итерации. Иначе помечаем её как пройденную.

3. Обходим всех соседей вершины и в случае если они ещё не были посещены выполняем релаксацию, пытаясь улучшить расстояние до соседей используя текущую вершину `u`. Если улучшение возможно, добавляем вершину в соответствующую очередь и обновляем `history`

В конце происходит вывод искомого расстояние и стандартное восстановление пути.

Сложность алгоритма

- По времени: Из идейного описания алгоритма его сложность очевидно будет $O(k|V| + |E|)$, что в случае данной задачи можно оценить как $O(2NM + NM)$ что эквивалентно $O(NM)$
- По памяти: В памяти хранится входная таблица, массив очередей, и вспомогательные массивы. Каждый из них может содержать не более NM элементов, что даёт сложность $O(NM)$

Итого:

- По времени: $O(NM)$
- По памяти: $O(NM)$

Код

```cpp
#include <bits/stdc++.h>
using namespace std;

constexpr int INF = 1e9;
constexpr int k = 2;

struct point {
  int x;
  int y;
};

struct path_node {
  int x;
  int y;
  char c;
};
constexpr int dx[4] = {0, -1, 0, 1};
constexpr int dy[4] = {-1, 0, 1, 0};
constexpr char dir[4] = {'W', 'N', 'E', 'S'};

int main() {
  int n, m;
  cin >> n >> m;
  vector g(n, vector<char>(m));

  point p1, p2;
  cin >> p1.x >> p1.y >> p2.x >> p2.y;
  p1.x--;
  p1.y--;
  p2.x--;
```

```
  p2.y--;

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      cin >> g[i][j];
    }
  }

  vector<queue<point>> bfs(k + 1);
  vector d(n, vector(m, INF));
  vector used(n, vector(m, false));
  vector history(n, vector<path_node>(m));
  int q_ptr = 0;
  int cnt = 1;  // Счётчик вершин в очередях

  d[p1.x][p1.y] = 0;
  bfs[0].push(p1);
  history[p1.x][p1.y] = {-1, -1, ' '};

  while (cnt > 0) {
    while (bfs[q_ptr % (k + 1)].empty()) {
      q_ptr++;
    }

    const point u = bfs[q_ptr % (k + 1)].front();
    bfs[q_ptr % (k + 1)].pop();
    cnt--;

    if (used[u.x][u.y]) {
      continue;
    }

    used[u.x][u.y] = true;

    for (int i = 0; i < 4; i++) {
      point v = {u.x + dx[i], u.y + dy[i]};
      if (v.x < 0 || v.x >= n || v.y < 0 || v.y >= m) {
        continue;
      }

      if (g[v.x][v.y] == '#') {
        continue;
      }

      int weight = g[v.x][v.y] == '.' ? 1 : 2;

      if (d[u.x][u.y] + weight < d[v.x][v.y]) {
        d[v.x][v.y] = d[u.x][u.y] + weight;
        bfs[d[v.x][v.y] % (k + 1)].push(v);
        cnt++;
```

```
81          history[v.x][v.y] = {u.x, u.y, dir[i]};
82        }
83      }
84    }
85    if (d[p2.x][p2.y] == INF) {
86      cout << -1 << endl;
87      return 0;
88    }
89    cout << d[p2.x][p2.y] << endl;
90    string ans;
91    path_node p = history[p2.x][p2.y];
92    while (p.x != -1) {
93      ans += p.c;
94      p = history[p.x][p.y];
95    }
96    reverse(ans.begin(), ans.end());
97    cout << ans << endl;
98  }
```

Задача 2 Свинки-копилки

Решение

Заметим, что соотношения между копилками и ключами можно представить в виде ориентированного графа, в котором вершинами являются копилки, а ребро ориентированное ребро `u->v` говорит о том, что в копилке `u` лежит ключ от копилки `v` (следовательно разбив копилку `u` можно открыть (переход по графу), копилку `v`).

Заметим также, что если в графе существует цикл, то достаточно разбить всего одну копилку в нём чтобы открыть все копилки в цикле. Также стоит отметить, что в соответствие с условием задачи ключ от некоторой копилки существует единственном экземпляре, а значит в построенной модели графа в некоторую вершину может входить только одно ребро. Из этого следует, что из цикла не может выходить никакое ребро, а значит одна компонента связности графа всегда будет не более одного цикла. Наконец можно заметить, цикл будет ровно один, так как если предположить что его нет, что ключ от одной из копилок будет некуда положить, что не соответствует условию задачи. Это значит что достаточно развить ровно одну вершину в компоненте связности чтобы открыть все копилки в ней. Таким образом задача сводится к поиску количества компонент связности в графе, причём при ориентацию рёбер можно не учитывать.

В итоге решение заключается в том, чтобы на основе входных данных построить неориентированный граф, после чего выполнить подсчёт компонент связности с использованием dfs.

Сложность алгоритма

- По времени: В ходе подсчёта компонент связности каждая вершина просматривается один раз, что даёт сложность $O(N)$.
- По памяти: В памяти хранится граф в виде списков смежности. Количество элементов в списке равно `n`, что даёт сложность $O(N)$

Итого:

- По времени: $O(N)$
- По памяти: $O(N)$

Код

```cpp
#include <bits/stdc++.h>
using namespace std;

void dfs(const int v, vector<bool>& used, vector<vector<int>>& g) {
  used[v] = true;
  for (const auto u : g[v]) {
    if (!used[u]) {
      dfs(u, used, g);
    }
  }
}

int main() {
  int n;
  cin >> n;
  vector<vector<int>> g(n);
  for (int i = 0; i < n; i++) {
    int a;
    cin >> a;
    g[--a].push_back(i);
    g[i].push_back(a);
  }
  vector used(n, false);
  int ans = 0;
  for (int i = 0; i < n; i++) {
    if (!used[i]) {
      dfs(i, used, g);
      ans++;
    }
  }
  cout << ans << endl;
  return 0;
}
```

Задача 3 Долой списывание!

Решение

Заметим, что можно представить задачу в виде неориентированного графа, в котором вершинами являются ученики, а ребро говорит о том что два ученика обмениваются записками. Таким образом задача сводится к раскраске графа в два цвета, такой что никакое ребро не соединят две вершины одного цвета. Это классическая задача на графах и для её решения можно использовать поиск в глубину. Рекурсивная функция поиска в глубину будет принимать параметр цвета, в которых необходимо покрасить данную вершину. Цвета будет два: `0` и `1`. При этом информация о присвоенном цвете будет храниться в массиве `used`, который будет изначально инициализирован значениями `-1`. В самой функции `dfs` будет выполнятся рекурсивный вызов с цветом, противоположным полученному. При этом если в какой-то момент среди просматриваемых вершин будет встречена вершина с цветом аналогичным текущему, это будет значить что раскраска графа в два цвета невозможна. Для того чтобы понимать была ли раскраска успешной функция `dfs` будет возвращать значений `true` или `false`. Так как в графе может быть несколько компонент связности, функция `dfs` будет вызвана для каждой. При этом начальный цвет при вызове не играет роли и будет установлен в `0`.

Сложность алгоритма

- По времени: В процессе обхода каждое ребро и каждая вершина будут просмотрены не более одного раза. Получаем сложность $O(N + M)$
- По памяти: Граф будет храниться в виде матрицы смежности, что даёт сложность $O(N^2)$

Итого:

- По времени: $O(N + M)$
- По памяти: $O(N^2)$

Код

```cpp
#include <bits/stdc++.h>
using namespace std;

bool dfs(const int v, const int color, vector<int>& used,
 ↪  vector<vector<int>>& g) {
  used[v] = color;
  for (int i = 0; i < static_cast<int>(g[v].size()); i++) {
    if (g[v][i] == 0)
      continue;
    if (used[i] == color)
      return false;
    if (used[i] == -1 && !dfs(i, (color + 1) % 2, used, g))
      return false;
  }
  return true;
}

int main() {
  int n, m;
  cin >> n >> m;
  vector g(n, vector(n, 0));
  for (int i = 0; i < m; i++) {
    int a, b;
    cin >> a >> b;
    g[a - 1][b - 1] = 1;
    g[b - 1][a - 1] = 1;
  }
  vector used(n, -1);
  for (int i = 0; i < n; i++) {
    if (used[i] == -1 && !dfs(i, 0, used, g)) {
      cout << "NO" << endl;
      return 0;
    }
  }
  cout << "YES" << endl;
  return 0;
}
```

Задача 4 Авиаперелёты

Решение

Представим задачу в виде ориентированного взвешенного графа, в котором вершиной является город, а взвешенной ребро говорит и стоимости перелёта из одного города в другой. Заметим, что если у нас есть некоторое значение ёмкости топливного бака, то можно проверить выполнение условия задачи (достижимость каждого города из каждого). Для этого достаточно выполнить обход (например в глубину) графа, игнорируя при этом рёбра, вес которых больше размера топливного бака. После этого достаточно проверить что все вершины были посещены. Наконец заметим, что если бак некоторого размера является подходящим, то любой бак большего размера также подходит. Напротив, если бак некоторого размера не подходит, то и любой бак меньшего размера не подходит. Таким образом подбор значения размера бака может быть выполнен при помощи бинарного поиска. Также важно отметить, что в данной задаче обход возможен в двух направлениях, так что при проверке нужно запустить два обхода: сначала для одного направления, потому для другого. Для этого достаточно использовать сначала верхнюю половину матрицы смежности, потом нижнюю.

Сложность алгоритма

- По времени: Сложность бинарного поиска $O(\log A)$, где A - максимальный возможный размер бака. При этом на каждом шаге поиска, выполняется обход графа, который занимает время $O(N^2)$. В итоге получаем $O(N^2 \log A)$
- По памяти: В памяти хранится матрица смежности, которая занимает $O(N^2)$ по памяти

Итого:

- По времени: $O(N^2 \log A)$
- По памяти: $O(N^2)$

Код

```cpp
#include <bits/stdc++.h>
using namespace std;

void dfs(const int v, const int c, const bool dir, vector<bool>& used,
 ↪  vector<vector<int>>& g) {
  used[v] = true;
  for (int i = 0; i < static_cast<int>(g[v].size()); i++) {
    if (!used[i] && (dir ? g[v][i] : g[i][v]) <= c)
      dfs(i, c, dir, used, g);
  }
}
bool is_all_visited(vector<bool>& used) {
  for (const auto a : used)
    if (!a)
      return false;
  return true;
}
bool check(const int c, vector<vector<int>>& g) {
  const int n = static_cast<int>(g.size());
  vector used(n, false);
  dfs(0, c, true, used, g);
  if (!is_all_visited(used))
    return false;
  used = vector(n, false);
  dfs(0, c, false, used, g);
  return is_all_visited(used);
}

int main() {
  int n;
  cin >> n;
  vector g(n, vector<int>(n));
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      cin >> g[i][j];
    }
  }
  int l = -1, r = static_cast<int>(1e9) + 1;
  while (r - l > 1) {
    int c = (l + r) / 2;
    if (check(c, g))
      r = c;
    else
      l = c;
  }
  cout << r << endl;
  return 0;
}
```