МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное
образовательное учреждение высшего образования
“Национальный исследовательский университет ИТМО”

ФАКУЛЬТЕТ ПРОГРАММНОЙ ИНЖЕНЕРИИ И КОМПЬЮТЕРНОЙ ТЕХНИКИ

ЛАБОРАТОРНАЯ РАБОТА №4

по дисциплине
“АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ”
Базовые задачи.

выполнил:
Студент группы P32311
Птицын Максим Евгеньевич
Преподаватели:
Косяков М.С.
Тараканов Д.С.

г. Санкт-Петербург
2023 г.

1 M. Цивилизация

Условие задачи:

Карта мира в компьютерной игре «Цивилизация» версии 1 представляет собой прямоугольник, разбитый на квадратики. Каждый квадратик может иметь один из нескольких возможных рельефов, для простоты ограничимся тремя видами рельефов — поле, лес и вода. Поселенец перемещается по карте, при этом на перемещение в клетку, занятую полем, необходима одна единица времени, на перемещение в лес — две единицы времени, а перемещаться в клетку с водой нельзя.
У вас есть один поселенец, вы определили место, где нужно построить город, чтобы как можно скорее завладеть всем миром. Найдите маршрут переселенца, приводящий его в место строительства города, требующий минимального времени. На каждом ходе переселенец может перемещаться в клетку, имеющую общую сторону с той клеткой, где он сейчас находится.

Пояснение к примененному алгоритму:

Суть задачи - дан ориентированный граф с заданным весом рёбер, нужно найти минимальный путь от одной заданной вершины к другой. Наиболее популярные алгоритмы, решающие эту задачу: Дейкстра(O($nlog(n)$)), Беллман-Форд (O($|V| * |E|$)), A* (экспоненциальный рост сложности при большом количестве рёбер, оптимален, если исследуемый граф дерево), Флойд-Уоршелл (O(N^3)). Очевидно, что в нашем случае (количество рёбер в графе приближено к $2N^2$) оптимальным вариантом будет алгоритм Дейкстры.
Так же, чтобы оптимизировать память, заходя в каждую вершину, я сохраняю откуда я в неё пришел, а в конце прохожу маршрут в обратную сторону и собираю строку пути.
Сложность алгоритма: Дейкстра O($nlog(n)$) + Восстановление пути O(n) $\approx$ O($nlog(n)$)

Код:

```
int64_t n, m, sx, sy, fx, fy;
std::cin >> n >> m >> sy >> sx >> fy >> fx;
sy--;
sx--;
fy--;
fx--;
int64_t y = sy;
int64_t x = sx;
std::vector<std::vector<int64_t>> matrix(n, std::vector<int64_t>(m));
for (int64_t i = 0; i < n; i++) {
  for (int64_t j = 0; j < m; j++) {
    char c;
    std::cin >> c;
    if (c == '.')
      matrix[i][j] = 1;
    if (c == 'W')
      matrix[i][j] = 2;
    if (c == '#')
      matrix[i][j] = -1;
  }
}
matrix[y][x] = 0;
std::vector<std::vector<char>> paths_matrix(n, std::vector<char>(m));
std::priority_queue<
    std::pair<int64_t, std::pair<int64_t, int64_t>>,
    std::vector<std::pair<int64_t, std::pair<int64_t, int64_t>>>,
    std::greater<>>
    queue;
if (x > 0 && matrix[y][x - 1] != -1) {
  std::pair<int64_t, std::pair<int64_t, int64_t>> pair = {matrix[y][x - 1],
                                                          {y, x - 1}};
  matrix[y][x - 1] = 0;
  paths_matrix[y][x - 1] = 'W';
  queue.emplace(pair);
}
if ((x + 1) < m && matrix[y][x + 1] != -1) {
  std::pair<int64_t, std::pair<int64_t, int64_t>> pair = {matrix[y][x + 1],
                                                          {y, x + 1}};
  matrix[y][x + 1] = 0;
  paths_matrix[y][x + 1] = 'E';
  queue.emplace(pair);
}
if (y > 0 && matrix[y - 1][x] != -1) {
  std::pair<int64_t, std::pair<int64_t, int64_t>> pair = {matrix[y - 1][x],
                                                          {y - 1, x}};
  matrix[y - 1][x] = 0;
  paths_matrix[y - 1][x] = 'N';
  queue.emplace(pair);
}
```

```
if ((y + 1) < n && matrix[y + 1][x] != -1) {
  std::pair<int64_t, std::pair<int64_t, int64_t>> pair = {matrix[y + 1][x],
                                                          {y + 1, x}};
  matrix[y + 1][x] = 0;
  paths_matrix[y + 1][x] = 'S';
  queue.emplace(pair);
}
int64_t ans = -1;
while (!queue.empty()) {
  std::pair<int64_t, std::pair<int64_t, int64_t>> curr = queue.top();
  queue.pop();
  y = curr.second.first;
  x = curr.second.second;
  int64_t cost = curr.first;
  if (y == fy && x == fx) {
    ans = cost;
    break;
  }
  if (x > 0 && matrix[y][x - 1] != -1 && matrix[y][x - 1] != 0) {
    std::pair<int64_t, std::pair<int64_t, int64_t>> pair = {
        cost + matrix[y][x - 1], {y, x - 1}};
    matrix[y][x - 1] = 0;
    paths_matrix[y][x - 1] = 'W';
    queue.emplace(pair);
  }
  if ((x + 1) < m && matrix[y][x + 1] != -1 && matrix[y][x + 1] != 0) {
    std::pair<int64_t, std::pair<int64_t, int64_t>> pair = {
        cost + matrix[y][x + 1], {y, x + 1}};
    matrix[y][x + 1] = 0;
    paths_matrix[y][x + 1] = 'E';
    queue.emplace(pair);
  }
  if (y > 0 && matrix[y - 1][x] != -1 && matrix[y - 1][x] != 0) {
    std::pair<int64_t, std::pair<int64_t, int64_t>> pair = {
        cost + matrix[y - 1][x], {y - 1, x}};
    matrix[y - 1][x] = 0;
    paths_matrix[y - 1][x] = 'N';
    queue.emplace(pair);
  }
  if ((y + 1) < n && matrix[y + 1][x] != -1 && matrix[y + 1][x] != 0) {
    std::pair<int64_t, std::pair<int64_t, int64_t>> pair = {
        cost + matrix[y + 1][x], {y + 1, x}};
    matrix[y + 1][x] = 0;
    paths_matrix[y + 1][x] = 'S';
    queue.emplace(pair);
  }
}
if (ans == -1)
  std::cout << -1;
else {
  std::string path;
  x = fx;
  y = fy;
  while (!(x == sx && y == sy)) {
    path = paths_matrix[y][x] + path;
    if (paths_matrix[y][x] == 'N') {
      y += 1;
    } else if (paths_matrix[y][x] == 'S') {
      y -= 1;
    } else if (paths_matrix[y][x] == 'E') {
      x -= 1;
    } else if (paths_matrix[y][x] == 'W') {
      x += 1;
    }
  }
  std::cout << ans << std::endl << path;
}
return 0;
```

2 N. Свинки-копилки

Условие задачи:

У Васи есть n свинок-копилок, свинки занумерованы числами от 1 до n. Каждая копилка может быть открыта единственным соответствующим ей ключом или разбита. Вася положил ключи в некоторые из копилок (он помнит, какой ключ лежит в какой из копилок). Теперь Вася собрался купить машину, а для этого ему нужно достать деньги из всех копилок. При этом он хочет разбить как можно меньшее количество копилок (ведь ему еще нужно копить деньги на квартиру, дачу, вертолет...). Помогите Васе определить, какое минимальное количество копилок нужно разбить.

Пояснение к примененному алгоритму:

Суть задачи сводится к нахождению количества компонент связности в ориентированном графе: в каждой компоненте есть начальная вершина (не обязательно одна, зайдя в которую (((разбив копилку))), потом можно дойти до остальных вершин в этой компоненте по рёбрам (((используя открытые ключи)))).
Мой алгоритм втупую обходит весь граф: беру любую вершину, обхожу рекурсивно всех её потомков, добавляя вершины в компоненту связности, далее рекурсивно обхожу всех её родителей, заходя в каждого родителя рекурсивно обхожу всех его потомков, если не делал этого ранее, так же добавляя родителей в компоненту. Дополнительно сохраняю вне компонент каждую посещённую вершину, чтобы не заходить в неё дважды. После обхода всей компоненты, сохраняю её и ищу, остались ли непосещённые вершины, пока такие вершины есть - так же обхожу рекурсивно вверх и вниз их. В конце получаю все имеющиеся компоненты связности.
Сложность алгоритма: $O(n)$, потому что в каждую вершину я могу зайти максимум дважды (при обходе вверх и вниз).

Код:

```
void round_down(const std::vector<std::vector<bool>> &list,
                std::vector<bool> &check_list,
                std::unordered_set<size_t> &cluster,
                size_t i) {
    cluster.emplace(i);
    check_list[i] = true;
    for (size_t j = 0; j < list.size(); j++) {
        if (j == i)
            continue;
        if (list[i][j] && !check_list[j])
            round_down(list, check_list, cluster, j);
    }
}

void round_up(const std::vector<std::vector<bool>> &list,
              std::vector<bool> &check_up_list,
              std::vector<bool> &check_list,
              std::unordered_set<size_t> &cluster,
              size_t i) {
    check_up_list[i] = true;
    round_down(list, check_list, cluster, i);
    for (size_t j=0; j<list.size(); j++) {
        if (list[j][i] && !check_up_list[j]) {
            round_up(list,check_up_list,check_list, cluster, j);
        }
    }
}

int main() {
    size_t N;
    std::cin >> N;
    std::vector<std::vector<bool>> list(N, std::vector<bool>(N));
    for (size_t i = 0; i < N; i++) {
        size_t j;
        std::cin >> j;
        list[j - 1][i] = true;
    }
    std::vector<std::unordered_set<size_t>> clusters;
    std::vector<bool> check_list(N);
    std::vector<bool> check_up_list(N);
    for (size_t i = 0; i < N; i++) {
        std::unordered_set<size_t> cluster;
        if (!check_list[i]) {
            round_up(list,check_up_list, check_list, cluster, i);
            clusters.push_back(cluster);
        }
    }
    std::cout << clusters.size() << std::endl;
    return 0;
}
```

3 O. Долой списывание!

Условие задачи:

Во время теста Михаил Дмитриевич заметил, что некоторые лкшата обмениваются записками. Сначала он хотел поставить им всем двойки, но в тот день Михаил Дмитриевич был добрым, а потому решил разделить лкшат на две группы: списывающих и дающих списывать, и поставить двойки только первым.

У Михаила Дмитриевича записаны все пары лкшат, обменявшихся записками. Требуется определить, сможет ли он разделить лкшат на две группы так, чтобы любой обмен записками осуществлялся от лкшонка одной группы лкшонку другой группы.

Пояснение к примененному алгоритму:

Задача: проверить, является ли заданный граф с алкашатами двудольным. Необходимый и достаточный критерий двудольности графа - его можно раскрасить в 2 цвета. Мой алгоритм подвешивает граф за какую-то вершину и раскрашивает её по уровням в два цвета обходом в ширину. Если все связные с корнем вершины пройдены, но есть ещё рассмотренные вершины, то алгоритм повторяется до тех пор, пока не будут рассмотрены все вершины.

Сложность алгоритма: $O(|V| + |E|)$

Код:

```
size_t N, M;
std::cin >> N >> M;
std::unordered_multimap<size_t, size_t> edges;
std::unordered_set<size_t> unvisited_nodes;
for (size_t i = 0; i < M; i++) {
    size_t x, y;
    std::cin >> x >> y;
    edges.emplace(x, y);
    edges.emplace(y, x);
    unvisited_nodes.emplace(x);
    unvisited_nodes.emplace(y);
}
std::queue<std::pair<size_t, uint32_t>> queue;
std::unordered_set<size_t> red;
std::unordered_set<size_t> blue;
while(!unvisited_nodes.empty()) {
    auto item = unvisited_nodes.begin();
    queue.push({*item, 0});
    unvisited_nodes.erase(item);
    while (!queue.empty()) {
        std::pair<size_t, uint32_t> node = queue.front();
        queue.pop();
        unvisited_nodes.erase(node.first);
        auto iter = edges.find(node.first);
        while (iter != edges.end()) {
            if (unvisited_nodes.find(iter->second) != unvisited_nodes.end()) queue.push({iter->second, node.second+1});
            edges.erase(iter);
            iter = edges.find(node.first);
        }
        if (node.second % 2 == 0) {
          if (blue.find(node.first) != blue.end()) {
              std::cout << "NO";
              return 0;
          } else {
              red.emplace(node.first);
          }
        } else {
            if (red.find(node.first) != red.end()) {
                std::cout << "NO";
                return 0;
            } else {
                blue.emplace(node.first);
            }
        }
    }
}
std::cout << "YES";
return 0;
```

4 P. Авиаперелёты

Условие задачи:

Главного конструктора Петю попросили разработать новую модель самолёта для компании «Air Бубундия». Оказалось, что самая сложная часть заключается в подборе оптимального размера топливного бака.
Главный картограф «Air Бубундия» Вася составил подробную карту Бубундии. На этой карте он отметил расход топлива для перелёта между каждой парой городов.
Петя хочет сделать размер бака минимально возможным, для которого самолёт сможет долететь от любого города в любой другой (возможно, с дозаправками в пути).
Пояснение к примененному алгоритму:

Бинпоиск по ответу + N DFS'ов + оптимизация DFS'а (если я захожу в вершину, которую до этого уже проверил поиском в глубину и она связана со всеми, то и текущая проверяемая поиском в глубину вершина связана со всеми и я выхожу из поиска).
Сложность алгоритма: O($log(N)$ * N^2)

Код:

```
std::vector<bool> used_all;


void find_path(const std::vector<std::vector<uint32_t>> &matrix,
               std::vector<bool> &used,
               std::unordered_set<size_t> &nodes,
               size_t i, uint32_t dist) {
    used[i] = true;
    for (size_t j=0; j<matrix.size(); j++) {
        if (!used[j] && matrix[i][j] <= dist) {
            if (nodes.find(j) != nodes.end()) {
                used = used_all;
                return;
            } else {
                find_path(matrix,used, nodes, j,dist);
            }
        }
    }
}

int main() {
    size_t n;
    std::cin >> n;
    std::vector<std::vector<uint32_t>> matrix(n, std::vector<uint32_t>(n));
    uint32_t R = 0;
    for (size_t i = 0; i < n; i++) {
        for (size_t j = 0; j < n; j++) {
            uint32_t number;
            std::cin >> number;
            matrix[i][j] = number;
            R = std::max(number, R);
        }
    }

    used_all = std::vector<bool> (n, true);


    uint32_t L = 0;
    while (R - L > 1) {
        uint32_t dist = (R + L) / 2;
        bool isEnough = true;
        std::vector<bool> used(n);
        std::unordered_set<size_t> nodes;
        for (size_t i=0; i<n; i++) {
            find_path(matrix,used, nodes,i,dist);
            if (used != used_all) {
                isEnough = false;
                break;
            } else {
                nodes.emplace(i);
            }
            used = std::vector<bool> (n);
        }
        if (!isEnough) {
            L+= (R-L)/2;
        } else {
            R -= (R-L)/2;
        }
    }
    std::cout << R;

    return 0;
```