Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №4
по “Алгоритмам и структурам данных”
Базовые задачи

Выполнил:
Студент группы P3206
Сорокин А.Н.
Преподаватели:
Косяков М.С.
Тараканов Д.С.

Санкт-Петербург
2024

Задача №M “Цивилизация”
Решение за O((V+E) log V).

Дана карта мира. Ее можно представить в виде графа $G=(V(G), E(G), c)$, где $\Delta(G)=4$ (любой вершине инцидентны максимум 4 ребра, т.к. двигаться можно только вверх, вниз, вправо и влево). Заметим следующее:

- все ребра графа G имеют положительный вес
- карта конечна ($1 \leq N, M \leq 1000$)

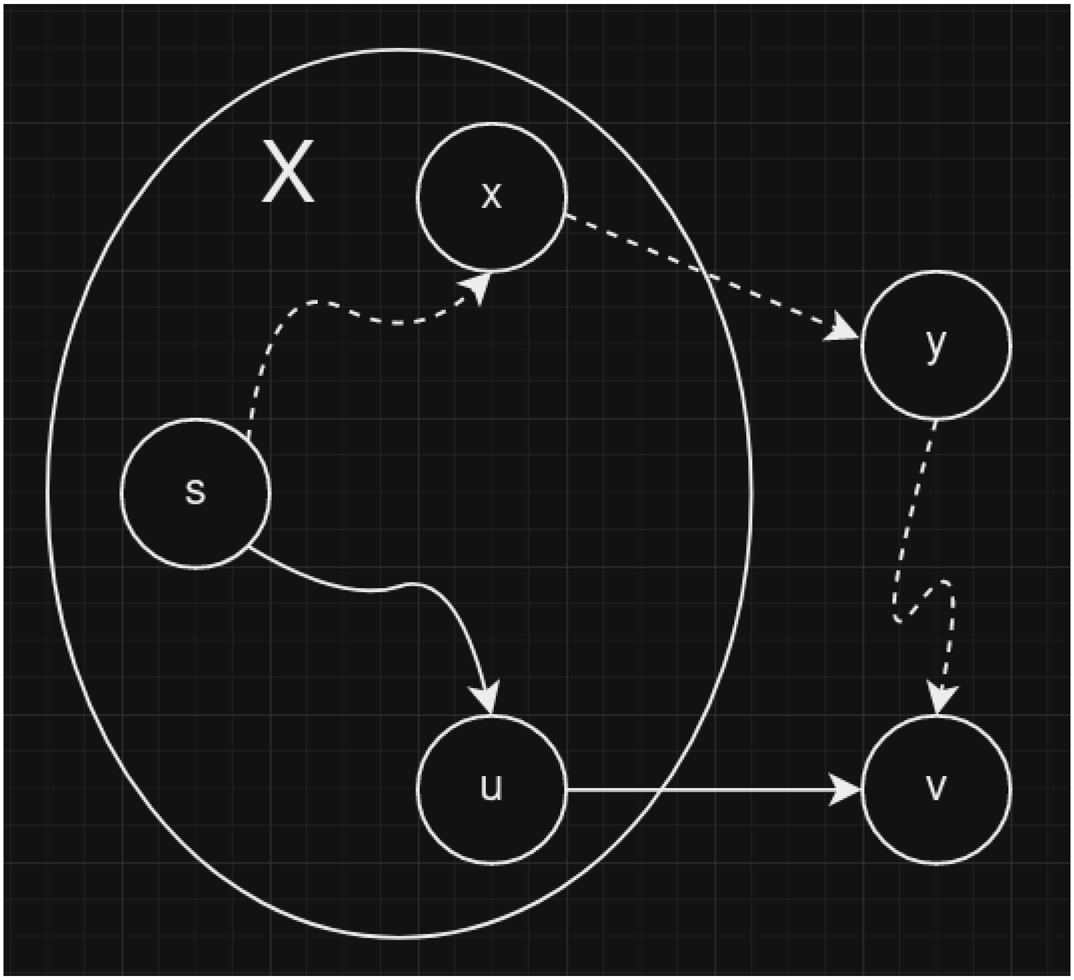
Следовательно, в качестве алгоритма поиска кратчайшего пути можно использовать **алгоритм Дейкстры**.

Заведем приоритетную очередь и добавим в нее начальную вершину. Начиная с начальной вершины, будем рассматривать 4 соседние клетки и рассчитывать путь до них по следующей формуле: путь от начала до v = путь от начала до u + расстояние от u до v. Гарантируется, что путь до вершины u является оптимальным. Если вершина не посещалась ранее или же прошлый путь до нее длиннее, обновляем этот путь и заносим вершину в очередь. Повторяем пока в очереди не останется вершин или пока не будет найден путь из начальной вершины в конечную. Далее с помощью bAcKtRaCkInGa восстанавливается путь из начальной вершины в конечную.

Формула алгоритма: $DS(v)=d(u)+c((u,v))$, где $DS(v)$ – кратчайший путь в v , $d(u)$ – длина кратчайшего пути в u , $c(u,v)$ – вес ребра из u в v .

Моя реализация считает длину пути в вершину с водой как ∞

Почему данный алгоритм работает:



Пусть мы ищем кратчайший путь из вершины s в вершину v, и существует $DS(v)$. Положим P - альтернативный путь из s в v и (x,y) – первое ребро, выходящее из множества X рассмотренных алгоритмом вершин.
Пусть P' – часть пути от s до x.
Тогда:
$c(P) \geq c(P') + c((x,y)) \geq d(x) + c((x,y)) = DS(y) \geq DS(v)$, таким образом, не существует альтернативного пути, который будет оптимальнее $DS(v)$

(P.S. данную задачу так же можно решить алгоритмом A*. Этот алгоритм использует эвристику для того, чтобы определить направление движения в сторону конечной вершины (в то время как Дейкстра просто расширяется, рассматривая все вершины подряд). Однако использование данного алгоритма в этой задаче – стрельба из пушки по воробьям).

Код решения:

```
#include <algorithm>
#include <climits>
#include <fstream>
#include <iostream>
#include <list>
#include <queue>
#include <set>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <utility>
#include <vector>

using namespace std;
const long INF = 1e7;

struct Node {
  long g;
  int num;
};
bool operator<(const Node &a, const Node &b) { return a.g > b.g; }

int main() {
  ifstream infile("input.txt");

  int n, m, y_start, x_start, y_end, x_end;
  infile >> n >> m >> y_start >> x_start >> y_end >> x_end;

  vector<string> grid(n);
  for (int i = 0; i < n; ++i) {
    infile >> grid[i];
  }

  vector<Node> visited(n * m);
```

```
for (int i = 0; i < n * m; ++i) {
  visited[i] = {-1, -1};
}

unordered_map<char, int> chmp;
chmp['.'] = 1;
chmp['W'] = 2;
chmp['#'] = INF;

int start = (x_start - 1) + (y_start - 1) * m;
int end = (x_end - 1) + (y_end - 1) * m;

visited[start] = {0, start};
priority_queue<Node> pq;
pq.push({0, start});

while (!pq.empty()) {
  auto curr_node = pq.top();
  pq.pop();

  if (curr_node.num == end)
    break;

  int x_cur = curr_node.num % m;
  int y_cur = curr_node.num / m;

  multiset<Node> v;

  if (x_cur - 1 >= 0) {
    v.insert({chmp[grid[y_cur][x_cur - 1]], curr_node.num - 1});
  }
  if (x_cur + 1 < m) {
    v.insert({chmp[grid[y_cur][x_cur + 1]], curr_node.num + 1});
  }
  if (y_cur - 1 >= 0) {
    v.insert({chmp[grid[y_cur - 1][x_cur]], curr_node.num - m});
```

```
  }
  if (y_cur + 1 < n) {
    v.insert({chmp[grid[y_cur + 1][x_cur]], curr_node.num + m});
  }

  for (Node next_node : v) {
    long DS =
        curr_node.g + next_node.g >= INF ? INF : curr_node.g + next_node.g;
    if (visited[next_node.num].g == -1 || visited[next_node.num].g > DS) {
      pq.push({DS, next_node.num});
      visited[next_node.num].g = DS;
      visited[next_node.num].num = curr_node.num;
    }
  }
}
if (visited[end].num == -1 || visited[end].g >= INF) {
  cout << -1;
  return 0;
}
cout << visited[end].g << "\n";
list<char> res;

unordered_map<int, char> direct;
direct[1] = 'E';
direct[-1] = 'W';
direct[m] = 'S';
direct[-m] = 'N';

int curr = end;
while (curr != start) {
  res.push_front(direct[curr - visited[curr].num]);
  curr = visited[curr].num;
}

for (char i : res) {
  cout << i;
```

```
  }
 return 0;
}
```

Задача №N “Свинки-копилки”
Решение за O(E + V log V).

Вот копилка i. В ней лежат ключи от других копилок. Ключ от каждой копилки существует только один, рабитие копилки дает нам доступ ко всем копилкам, ключи которых находились в этой копилке. Необходимо разбить минимальное кол-во копилок.

Что нам позволит понять, какую копилку разбить?

Пусть мы смотрим на копилку i. В ней лежат некоторые ключи от копилок, но нас больше интересует то, в какой копилке лежит ключ от копилки i. И если мы найдем такую копилку (пусть это копилка j), то мы уже будем искать копилку, в которой лежит ключ от j.

Так мы проверим каждую копилку и для каждой копилки найдем ту копилку, разбитие которой автоматически приведет нас к открытию данной.

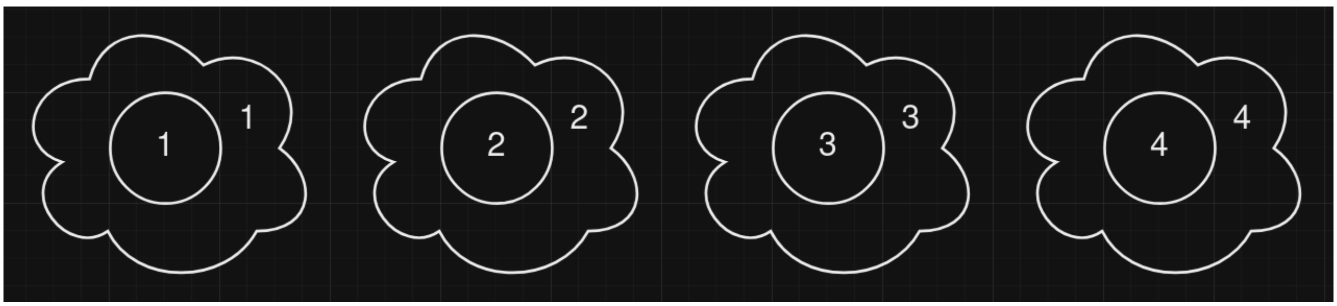
Воспользуемся **Union Find**.

Непересекающиеся множества (Disjoint Set) в данной задаче – множества копилок, где ни одна копилка одного множества не входит в другое множество. Для нашего случая: копилки i и j входят в разные множества, если не существует такой копилки k, разбитие которой позволит открыть обе копилки. Следовательно, для решения данной задачи необходимо посчитать количество непересекающихся множеств.

Union Find работает следующим образом: метод find двигается от вершины к ее родителю до тех пор, пока родитель вершины не станет равен самой вершине (то есть не существует родителей для этой вершины). Union использует find для вершин u и v, и если родители не совпали – делает родителя одной вершины родителем другой.

Пример из задачи:
4
2
1
2
4

Имеем:

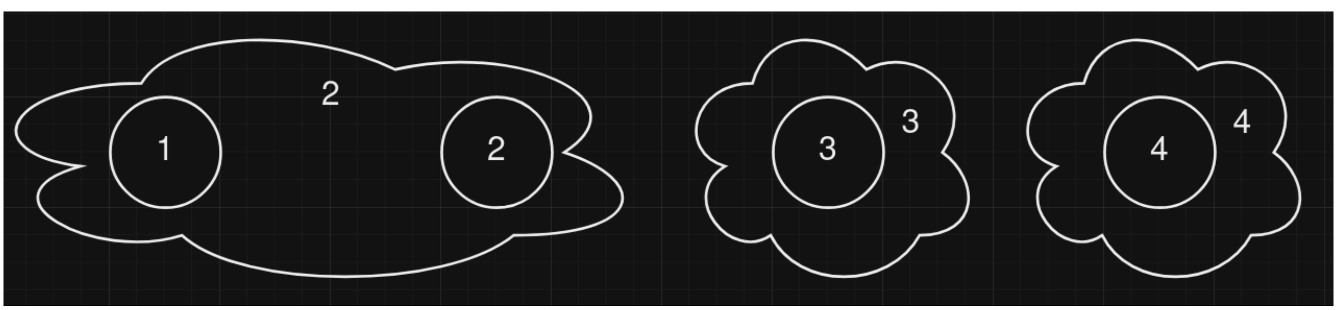


где i - копилка, $parent[i]$ – родитель (облачко). Изначально $parent[i]=i$

Рассмотрим 1 – 2 (ключ от 1 копилки лежит во 2 копилке).

Т. к. $parent[i]=i$, то find() найдет родителей 1 и 2, а затем, поскольку они различны, union() положит $parent[1]=2$

Получим:



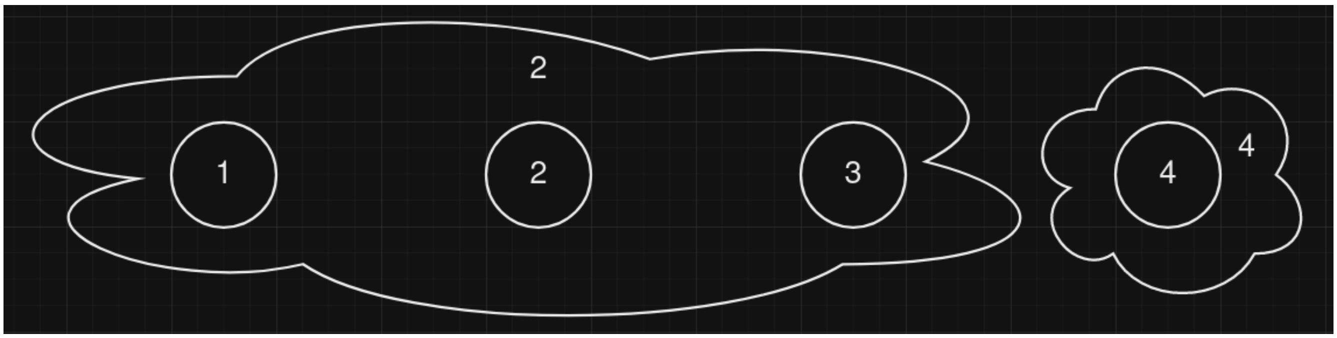
Рассмотрим 2 – 1 (ключ от 2 копилки лежит в 1 копилке).

Поскольку $parent[1]=2$ и $parent[2]=2$, обе копилки входят в одно множество, ничего изменять не нужно.

Рассмотрим 3 – 2 (ключ от 3 копилки лежит во 2 копилке).

find() найдет $parent[2]=2$ и $parent[3]=3$, а затем, поскольку они различны, union() положит $parent[1]=2$

Получим:



Рассмотрим 4 – 4 (ключ от 4 копилки в ней же)

Очевидно, у них один и тот же родитель, следовательно, без изменений.

Итого 2 множества, т. е. необходимо разбить только 2 копилки.

~~(Приятно, что в задаче не просят указать, какие)~~

Код решения:

```
#include <fstream>
#include <iostream>
#include <unordered_map>
#include <unordered_set>
#include <vector>
using namespace std;

int findParent(vector<int> &A, int n) {
  while (A[n] != n) {
    n = A[n];
```

```
  }
  return n;
}

void uniteParents(vector<int> &A, int u, int v) {
  int v1 = findParent(A, u);
  int v2 = findParent(A, v);
  if (v1 == v2) {
    // already checked
    return;
  }
  A[v1] = v2;
}
int countDistinct(vector<int> &A) {
  int c = 0;
  for (int i = 0; i < A.size(); ++i) {
    if (A[i] == i) {
      ++c;
    }
  }
  return c;
}

int main() {
  int n;
  ifstream infile("input.txt");
  infile >> n;
  vector<int> parents(n);
  for (int i = 0; i < n; ++i) {
    parents[i] = i;
  }

  for (int i = 0; i < n; ++i) {
    int p;
    infile >> p;
    uniteParents(parents, i, p - 1);
```

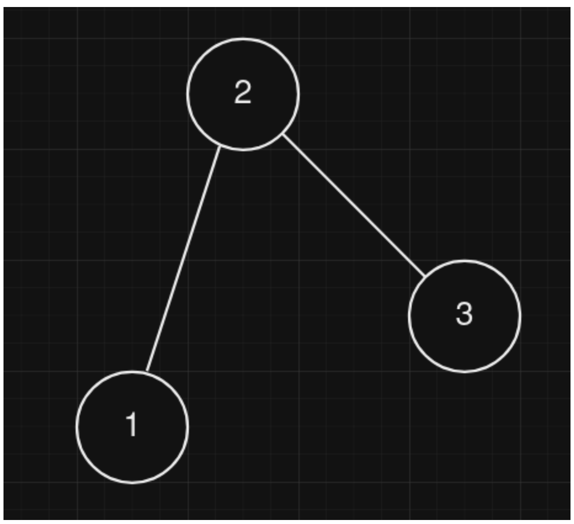
```
  }

  cout << countDistinct(parents);
  return 0;
}
```

Задача №O “Долой списывание!”

Решение за O(V + E).

Как четко разбить школьников на 2 группы?
Рассматривая пример из задания
3 2
1 2
2 3
Получаем граф:

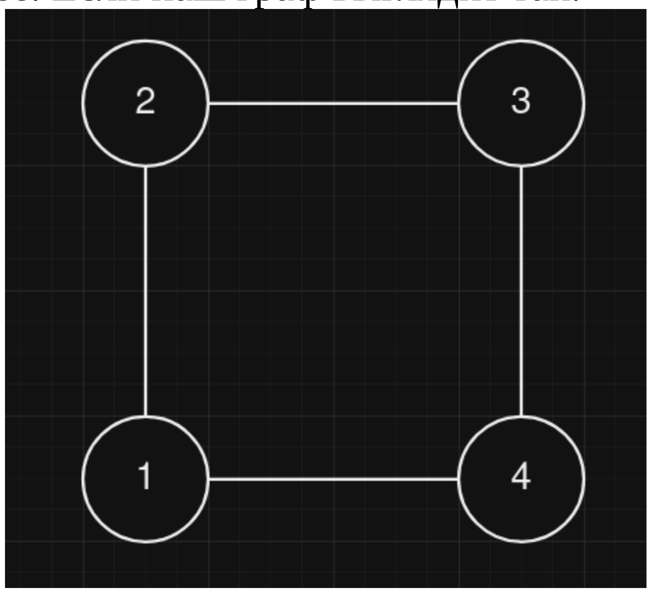


Из этого графа очевидно, что школьников можно разбить на 2 группы: пусть, например, 2 дает списывать, а 1 и 3 списывают. Однако если добавить ребро (1, 3), тогда их уже нельзя будет разбить – не понятно, кто у кого списывает.
Следовательно, разделение будет представлять из себя 2 множества вершин, где вершины одного множества соединены с вершинами другого и при этом не имеют связей между собой. Данная структура называется **двудольный граф**.

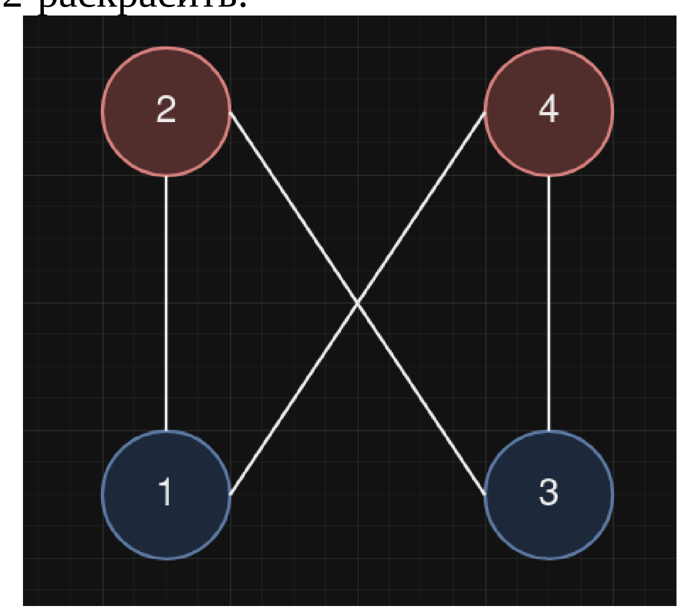
Пусть, чтобы определить, можно ли разбить школьников на 2 группы, мы их покрасим в 2 цвета. Если окажется, что их можно раскрасить так, что никакие 2 связанные вершины не будут иметь одинаковый цвет, значит, граф двудольный, и школьников можно разбить на 2 группы.

Более формально: **граф 2-раскрашиваемый тогда и только тогда, когда он двудольный**.

Заметим следующее. Если наш граф выглядит так:



то мы можем его 2-раскрасить:



Заметим, что если у нас таким образом связаны 6, 8, 10, … 2n вершин, то мы можем их 2-раскрасить. При этом, если у нас так связано 3, 5, 7 … 2n+1 вершин, такой граф 2-раскрасить не получится.

Теорема Кёнига: граф G является двудольным тогда и только тогда, когда все циклы в графе G имеют четную длину.

Данная теорема поможет нам попытаться 2-раскрасить граф.

Воспользуемся **поиском в глубину (DFS)** чтобы обойти все вершины графа. Пусть мы раскрасим первую вершину в цвет 1. Тогда попытаемся раскрасить смежные вершины цветом 2, а смежные с ними – цветом 1, и так далее. Покраска четного цикла приведет к ситуации, когда мы придем к вершине, уже покрашенной в нужный нам цвет. В свою очередь если мы, допустим, захотим покрасить вершину в цвет 2, а она уже была покрашена в цвет 1 – значит, был найден цикл нечетной длины, данный граф нельзя раскрасить в 2 цвета, и, следовательно, школьников нельзя разбить на 2 группы.

Следует помнить, что есть школьники, которые записками не обмениваются – их можно красить в любой цвет.

Код решения:

```
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;

bool dfs(vector<vector<int>> &graph, vector<int> &visited, int u, int color)
{
  if (visited[u] != 0) {
    if (visited[u] != color) {
      return false;
    } else {
      return true;
```

```
    }
  }
  visited[u] = color;
  color = (color == 1) ? 2 : 1;
  for (int v = 0; v < graph.size(); ++v) {
    if (graph[u][v] == 1) {
      if (!dfs(graph, visited, v, color)) {
        return false;
      }
    }
  }
  return true;
}

int main() {
  ifstream infile("input.txt");
  int n, m;
  infile >> n >> m;
  vector<int> visited(n, 0);
  vector<vector<int>> graph(n, vector<int>(n, 0));

  for (int i = 0; i < m; ++i) {
    int u, v;
    infile >> u >> v;
    graph[u - 1][v - 1] = 1;
    graph[v - 1][u - 1] = 1;
  }
  bool res = true;
  for (int u = 0; u < visited.size(); ++u) {
    if (visited[u] == 0) {
      res = res && dfs(graph, visited, u, 1);
    }
  }
  if (res) {
    cout << "YES";
  } else {
```

```
    cout << "NO";
  }
  return 0;
}
```

Задача №P “Авиаперелеты”
Решение за O(VE).

~~*Решим данную задачу алгоритмом Флойда-Уоршелла…*~~

Алгоритм Флойда-Уоршелла слишком медленный для данной задачи.

Необходимо найти минимальный размер топливного бака такой, чтобы можно было добраться из любого города в любой другой, учитывая, что в городах можно заправляться. В качестве входных данных дана матрица смежности, что очень удобно.

Как будем искать?

Обратимся к решению задачи №E “Коровы в стойла”. В решении использовался алгоритм **бинарного поиска** – в качестве условия подсчитанным значением предпринимались попытки расставить коров. Воспользуемся той же логикой для данной задачи: найдем максимальный вес ребра, найдем среднее между максимумом и минимумом, возьмем его за размер топливного бака и попробуем добраться до каждого города. Если с данным размером топливного бака получилось пометить все вершины, то с большим тем более получится, если нет – с меньшим можно не рассматривать.

Как посетить все вершины?

Нам достаточно просто пройтись в цикле по ним. По горизонтали и вертикали. Нет необходимости рассматривать все возможные пути из v в u с целью найти ребро наименьшего веса – бинарный поиск делает всю работу за нас. Данный обход можно реализовать через **поиск в глубину (DFS)**, помечая вершины на ходу. Необходимо проходить по вершинам как по горизонтали, так и по вертикали – по неизвестным причинам, ребро (A, B) может быть тяжелее ребра (B, A). Если после DFS хотя бы одна вершина осталась непомеченной – значит, емкости топливного бака не хватает для облета всех городов.

Код решения:

```
#include <algorithm>
#include <ios>
#include <iostream>
#include <vector>

using namespace std;
void DFS_horizontal(vector<vector<int>> &grid, vector<bool> &visited, int
v,
                    int target) {
  visited[v] = true;
  for (int i = 0; i < grid.size(); ++i) {
    if (!visited[i] && grid[v][i] <= target) {
      DFS_horizontal(grid, visited, i, target);
    }
  }
}

void DFS_vertical(vector<vector<int>> &grid, vector<bool> &visited, int v,
                  int target) {
  visited[v] = true;
  for (int i = 0; i < grid.size(); ++i) {
    if (!visited[i] && grid[i][v] <= target) {
      DFS_vertical(grid, visited, i, target);
    }
  }
}

bool tryToPass(vector<vector<int>> &matrix, int fuelCapacity) {
  int n = matrix.size();
```

```
  vector<bool> visited(n, false);

  DFS_horizontal(matrix, visited, 0, fuelCapacity);

  for (int i = 0; i < n; ++i) {
    if (!visited[i])
      return false;
  }

  vector<bool> visited2(n, false);

  DFS_vertical(matrix, visited2, 0, fuelCapacity);

  for (int i = 0; i < n; ++i) {
    if (!visited2[i])
      return false;
  }
  return true;
}

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  int n;
  cin >> n;
  int r = 0;
  vector<vector<int>> matrix(n, vector<int>(n, 0));
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      cin >> matrix[i][j];
      r = max(r, matrix[i][j]);
    }
  }

  int l = -1;
  ++r;
```

```
  while (r - l > 1) {
    int mid = (l + r) / 2;
    if (tryToPass(matrix, mid)) {
      r = mid;
    } else {
      l = mid;
    }
  }
  cout << r;
  return 0;
}
```