Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №4

по «Алгоритмам и структурам данных»

Базовые задачи (блок 4)

Выполнила:

Студентка группы P3230

Вавилина Е. А.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2025

Задача №1 «M. Цивилизация»

Пояснение к примененному алгоритму:
Нам надо найти минимальное расстояние между двумя вершинами. Для этого применим алгоритм Дейксты.

- Создадим приоритетную очередь и из стартовой точки (предка) будем обходить все связанные с ней вершины (сверху, снизу, слева и справа).
- Если расстояние для связанной вершины (потомка) получилось меньше, чем уже найденное расстояние до нее – заменим расстояние до вершины на новое меньшее и предшествующую ей вершину на предка. Потомка положим в очередь.
- Когда все 4 потомка обработаны – возьмем из очереди следующую вершину, с самым меньшим расстояние до нее, и повторим алгоритм.

Если встретим конечную вершину – минимальный путь найден. Восстановим маршрут, идя от потомка к предку (мы заранее сохранили для каждой вершины, из какой в нее надо прийти для меньшего маршрута).

Оценки:

1. Сложность по времени: **O(nm·log(nm))**, где n и m – число городов по вертикали и горизонтали. Работа с priority_queue занимает log(nm), т.к. максимум в нем лежит все вершины. При этом в while мы обработаем все возможные вершины в худшем случае.
2. Сложность по памяти: **O(nm)**. Т.к. у нас 4 структуры размера nm для хранения всех вершин, расстояний до вершины, предков и приоритетна очередь, в худшем случае состоящие из массивов константного размера.

Код:

```cpp
#include <algorithm>
#include <array>
#include <climits>
#include <iostream>
#include <queue>
#include <utility>
#include <vector>

const std::array<int, 4> DirectionX = {-1, 0, 1, 0};
const std::array<int, 4> DirectionY = {0, 1, 0, -1};
const std::array<char, 4> DirectionChars = {'N', 'E', 'S', 'W'};

using Coord = std::pair<int, int>;
using State = std::pair<int, Coord>;

Coord ReadCoord() {
  int x, y;
  std::cin >> x >> y;
  return {x - 1, y - 1};
}
```

```
std::string RecreatePath(Coord start, Coord end, std::vector<std::vector<Coord>>&
parent) {
  std::string path = "";
  int x = end.first, y = end.second;

  while (!(x == start.first && y == start.second)) {
    int px = parent[x][y].first;
    int py = parent[x][y].second;

    if (px == x - 1)
      path += 'S';
    else if (py == y + 1)
      path += 'W';
    else if (py == y - 1)
      path += 'E';
    else if (px == x + 1)
      path += 'N';

    x = px;
    y = py;
  }

  std::reverse(path.begin(), path.end());
  return path;
}

int main() {
  int n = 0, m = 0;
  std::cin >> n >> m;

  const Coord start = ReadCoord();
  const Coord end = ReadCoord();

  std::vector<std::vector<char>> matrix(n, std::vector<char>(m));
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      std::cin >> matrix[i][j];
    }
  }

  std::vector<std::vector<int>> dist(n, std::vector<int>(m, INT_MAX));
  std::vector<std::vector<Coord>> parent(n, std::vector<Coord>(m, {-1, -1}));
  std::priority_queue<State, std::vector<State>, std::greater<State>> pq;

  dist[start.first][start.second] = 0;
  pq.push({0, start});

  while (!pq.empty()) {
```

```
    const auto [cost, pos] = pq.top();
    pq.pop();

    if (pos == end)
      break;

    if (cost > dist[pos.first][pos.second])
      continue;

    for (int i = 0; i < 4; i++) {
      const Coord next = {pos.first + DirectionX[i], pos.second + DirectionY[i]};

      if (next.first < 0 || next.second < 0 || next.first >= n || next.second >= m)
        continue;

      if (matrix[next.first][next.second] == '#')
        continue;

      const int new_cost = cost + (matrix[next.first][next.second] == 'W' ? 2 : 1);

      if (new_cost < dist[next.first][next.second]) {
        dist[next.first][next.second] = new_cost;
        parent[next.first][next.second] = pos;
        pq.push({new_cost, next});
      }
    }
  }

  if (dist[end.first][end.second] == INT_MAX) {
    std::cout << -1;
  } else {
    std::cout << dist[end.first][end.second] << "\n";
    std::cout << RecreatePath(start, end, parent);
  }
}
```

Задача №2 «N. Свинки-копилки»

Пояснение к примененному алгоритму:

Для решения задачи надо найти количество компонент связанности (т.е. сколько копилок мы можем открыть, разбив одну в цепочке. Для этого у нас вершины должны быть связаны). Из одной компоненты связанности надо будет минимум разбить 1 копилку.

Для поиска компонент связанности воспользуемся следующим алгоритмом:

- Будем, начиная от вершины 1, помечать все вершины, которые с ней связаны сначала на расстоянии 1, потом на расстоянии 2 и т.д. (тут реализуется вспомогательный обход в ширину)

- Когда мы дошли до максимального уровня глубины – мы обработали все элементы в 1 компоненте связанности.
- Теперь возьмем какую-то не посещенную вершину и повторим для нее поиск компоненты.

Когда мы посетили все вершины – полученное число компонент связанности и есть итоговое количество разбитых копилок.

Оценки:

1. Сложность по времени: **O(n)**. Т.к. мы для каждой вершины проверяем, посещена она или нет, и только если она не посещена – запускаем обход в ширину. Т.е. суммарно каждую вершину мы будем обрабатывать 1 раз.
2. Сложность по памяти: **O(n)**. Т.к. у нас не более 2n ребер, которые мы храним (т.к. нужен неориентированный граф) и в очереди тоже максимум n вершин.

Код:

```
#include <iostream>
#include <queue>
#include <vector>

int main() {
  int n = 0;
  std::cin >> n;

  std::vector<std::vector<int>> connections(n + 1);

  for (int i = 1; i <= n; ++i) {
    int u = 0;
    std::cin >> u;

    connections[i].push_back(u);
    connections[u].push_back(i);
  }

  std::vector<bool> visited(n + 1, false);
  int component_count = 0;

  for (int i = 1; i <= n; i++) {
    if (!visited[i]) {
      component_count++;
      std::queue<int> bfs_queue;
      bfs_queue.push(i);

      while (!bfs_queue.empty()) {
        const int next = bfs_queue.front();
        bfs_queue.pop();

        for (const int j : connections[next]) {
```

```
          if (!visited[j]) {
            visited[j] = true;
            bfs_queue.push(j);
          }
        }
      }
    }
  }

  std::cout << component_count;
}
```

Задача №3 «O. Долой списывание!»

Пояснение к примененному алгоритму:

Для решения задачи нам надо проверить, является ли граф двудольным. Вершины одного цвета – одна доля, вершины другого цвета – вторая доля.

- Для этого будем проводить обход в глубину и начиная с первой еще не обработанной вершины и будем красить каждую следующую вершину в цвет, отличный от предыдущей.
- Если при этом у текущей вершины есть потомок того же цвета – разделить нельзя.
- Закончили обход для одной вершины – пойдем к следующей.

Граф заранее сделаем неориентированным.

Оценки:

1. Сложность по времени: **O(n+m)**. Т.к. мы для каждой вершины проверяем, посещена она или нет, и только если она не посещена – запускаем обход в глубину, который как раз и помечает вершины как пройденные. Т.е. суммарно каждую вершину мы будем обрабатывать 1 раз. При этом обход в глубину занимает n+m (т.к. у нас в худшем случае полный граф).
2. Сложность по памяти: **O(n+m)**. Т.к. у нас граф с m ребрами, сложность хранения которого O(m), массив для хранения цветов размером n и стек, хранящий максимум n вершин.

Код:

```
#include <iostream>
#include <stack>
#include <utility>
#include <vector>

bool static DFS(
    int start, const std::vector<std::vector<int>>& connections, std::vector<int>&
colors
) {
```

```cpp
  std::stack<std::pair<int, int>> dfs_stack;
  dfs_stack.emplace(start, 0);
  colors[start] = 0;

  while (!dfs_stack.empty()) {
    auto [next, color] = dfs_stack.top();
    dfs_stack.pop();

    for (const int j : connections[next]) {
      if (colors[j] == -1) {
        colors[j] = 1 - color;
        dfs_stack.emplace(j, colors[j]);
      }

      if (colors[j] == colors[next]) {
        return false;
      }
    }
  }

  return true;
}

int main() {
  int n = 0;
  int m = 0;
  std::cin >> n >> m;

  std::vector<std::vector<int>> connections(n + 1);

  for (int i = 0; i < m; ++i) {
    int u = 0;
    int v = 0;
    std::cin >> u >> v;
    connections[u].push_back(v);
    connections[v].push_back(u);
  }

  std::vector<int> colors(n + 1, -1);
  bool res = true;

  for (int i = 1; i <= n; i++) {
    if (colors[i] == -1) {
      if (!DFS(i, connections, colors)) {
        res = false;
        break;
      }
    }
  }
```

```
  if (res) {
    std::cout << "YES";
  } else {
    std::cout << "NO";
  }
}
```

Задача №4 «P. Авиаперелёты»

Для составления карты авиаперелетов нам надо найти минимальное число, при котором у нас будет сильно связный граф, состоящий из ребер меньше или равно тому самому минимальному числу.

Для нахождения такого числа будем перебирать бинарным поиском все числа от минимального до максимального. Для выбора интервала будем выбирать по результатам проверки связанности алгоритм Косараю.

Т.е. сначала проведем обход графа в глубину, а потом обход обратно ориентированного графа в глубину. Если смогли в обоих случаях обойти все вершины из 1 – значит критерий выполнен и можем поискать число меньше. Иначе – ищем число больше.

Оценки:

1. Сложность по времени: **O(n·n·log(m))**, где m – значение максимального ребра. Т.к. мы реализуем бинарный поиск с логарифмической сложностью вокруг поиска в глубину по полному графу, с квадратичной сложностью.
2. Сложность по памяти: **O(n·n)**. Т.к. у нас полный граф, сложность хранения которого O(n·n), массив для хранения посещений размером n и стек, хранящий максимум n вершин.

Код:

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

bool static DFS(int distance, const std::vector<std::vector<int>>& matrix, int n,
bool revers) {
  std::vector<bool> visited(n, false);

  std::stack<int> dfs_stack;
  dfs_stack.push(0);
  visited[0] = true;

  while (!dfs_stack.empty()) {
```

```
    const int i = dfs_stack.top();
    dfs_stack.pop();

    for (int j = n - 1; j >= 0; j--) {
      if (!revers && !visited[j] && matrix[i][j] <= distance) {
        visited[j] = true;
        dfs_stack.push(j);
      }

      if (revers && !visited[j] && matrix[j][i] <= distance) {
        visited[j] = true;
        dfs_stack.push(j);
      }
    }
  }

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

  return true;
}

bool static IsConnected(int distance, const std::vector<std::vector<int>>& matrix,
int n) {
  return DFS(distance, matrix, n, false) && DFS(distance, matrix, n, true);
}

int main() {
  int n = 0;
  int high = 0;
  int low = 0;

  std::cin >> n;

  std::vector<std::vector<int>> matrix(n, std::vector<int>(n, 0));

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      std::cin >> matrix[i][j];
      high = std::max(high, matrix[i][j]);
    }
  }

  while (high - 1 > low) {
    const int mid = low + ((high - low) / 2);
```

```
    if (IsConnected(mid, matrix, n)) {
      high = mid;
    } else {
      low = mid;
    }
  }

  std::cout << high;
}
```