Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №4

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнил:

Студент группы P3216

Брагин Р.А.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2025

M. Цивилизация

```cpp
#include <algorithm>
#include <iostream>
#include <queue>
#include <string>
#include <vector>
using namespace std;

struct Node {
  int weight, index;
  Node(int w = -1, int i = -1) : weight(w), index(i) {
  }
  bool operator<(const Node& other) const {
    return weight > other.weight;
  }
};

struct InputData {
  int rows, cols, start, goal;
  vector<string> grid;
};

InputData read_input() {
  int N, M, sx, sy, gx, gy;
  cin >> N >> M >> sx >> sy >> gx >> gy;
  vector<string> field(N);
  for (auto& row : field)
    cin >> row;
  int s = (sx - 1) * M + (sy - 1);
  int g = (gx - 1) * M + (gy - 1);
  return {N, M, s, g, field};
}

vector<Node> find_shortest_path(const InputData& data) {
  int total = data.rows * data.cols;
  vector<Node> trace(total);
  priority_queue<Node> q;
  q.emplace(0, data.start);

  vector<pair<int, int>> directions = {
```

```
      { 0,  1},
      { 0, -1},
      { 1,  0},
      {-1,  0}
  };

  while (!q.empty()) {
    Node current = q.top();
    q.pop();

    int x = current.index / data.cols;
    int y = current.index % data.cols;

    if (data.grid[x][y] == '#')
      continue;
    if (current.index == data.goal)
      break;

    for (auto [dx, dy] : directions) {
      int nx = x + dx, ny = y + dy;
      if (nx < 0 || ny < 0 || nx >= data.rows || ny >= data.cols)
        continue;
      if (data.grid[nx][ny] == '#')
        continue;

      int nid = nx * data.cols + ny;
      int cost = (data.grid[nx][ny] == 'W') ? 2 : 1;
      int new_weight = current.weight + cost;

      if (trace[nid].weight == -1 || trace[nid].weight > new_weight) {
        q.emplace(new_weight, nid);
        trace[nid] = {new_weight, current.index};
      }
    }
  }

  return trace;
}

string reconstruct_path(const vector<Node>& trace, int start, int
goal, int cols) {
  if (trace[goal].weight == -1)
```

```
    return "";

  string result;
  int current = goal;
  while (current != start) {
    int prev = trace[current].index;
    int diff = current - prev;
    if (diff == 1)
      result += 'E';
    else if (diff == -1)
      result += 'W';
    else if (diff == cols)
      result += 'S';
    else if (diff == -cols)
      result += 'N';
    current = prev;
  }

  reverse(result.begin(), result.end());
  return result;
}

int main() {
  InputData data = read_input();
  vector<Node> trace = find_shortest_path(data);

  cout << trace[data.goal].weight << endl;
  if (trace[data.goal].weight == -1)
    return 0;

  string path = reconstruct_path(trace, data.start, data.goal,
data.cols);
  cout << path << endl;

  return 0;
}
```

Описание:

Для решения задачи поиска кратчайшего маршрута переселенца используется модифицированный алгоритм Дейкстры. Карта

представляется в виде ориентированного взвешенного графа, где каждая клетка — это вершина, а возможные переходы в соседние клетки считаются рёбрами с весами. Вес перехода зависит от типа рельефа: поле (.) имеет вес 1, лес (W) — вес 2, а вода (#) недоступна для перемещения, поэтому такие клетки просто игнорируются.

Чтобы упростить работу с клетками и избежать двойной индексации, координаты карты преобразуются в линейные индексы. Далее инициализируются все клетки: расстояния до них устанавливаются в -1, что означает, что клетка ещё не была посещена. В приоритетную очередь помещается стартовая клетка с нулевой стоимостью, после чего запускается основной цикл алгоритма.

На каждой итерации из очереди извлекается клетка с наименьшей накопленной стоимостью. Для этой клетки проверяются все четыре соседние направления (вверх, вниз, влево, вправо). Если соседняя клетка доступна (не вода) и новая стоимость перехода меньше текущей сохраненной, то данные о стоимости и предыдущей клетке обновляются, а сама клетка добавляется в очередь.

Когда обработка завершена, проверяется, удалось ли достигнуть целевой клетки. Если нет — выводится -1. Если путь найден, начинается восстановление маршрута. Для этого используется массив, в котором хранятся предки клеток. Сравнивая текущую и предыдущую клетку, вычисляется направление движения: разница в индексах определяет, была ли это команда N (вверх), S (вниз), E (вправо) или W (влево). Таким образом, формируется строка маршрута, которую и нужно вывести в конце вместе со стоимостью пути.

Сложность:

Время: O(N*M*log(NM)), Так как каждая клетка может быть добавлена в очередь, а операции с priority_queue происходят за log.

Память: O(N * M)

N. Свинки-копилки

```cpp
#include <iostream>
#include <list>
#include <vector>
using namespace std;
```

```
void explore(int node, vector<pair<list<int>, int>>& g, int& total) {
  g[node].second = 1;
  for (int neighbor : g[node].first) {
    if (g[neighbor].second == 0) {
      explore(neighbor, g, total);
    } else if (g[neighbor].second == 1) {
      total++;
    }
  }
  g[node].second = 2;
}

int main() {
  int n, t, res = 0;
  cin >> n;
  vector<pair<list<int>, int>> g(n);
  for (int i = 0; i < n; ++i) {
    cin >> t;
    g[t - 1].first.push_back(i);
  }

  for (int i = 0; i < n; ++i) {
    if (g[i].second == 0) {
      explore(i, g, res);
    }
  }

  cout << res;
  return 0;
}
```

Описание:

Для решения задачи строится обратный граф: для каждой копилки i, в которой лежит ключ от копилки p, добавляется ребро p → i. Это значит, что открыв копилку p, можно получить доступ к копилке i. Далее выполняется обход в глубину (DFS) по всем вершинам графа. Каждая вершина при обходе помечается одним из трёх состояний: 0 — не посещена, 1 — находится в стеке вызовов (в процессе обхода), 2 — полностью обработана.

Если во время обхода мы попадаем в вершину, которая уже находится в стеке (статус 1), значит обнаружен цикл. Каждый такой цикл требует разбить хотя бы одну копилку, чтобы получить доступ ко всем копилкам в цикле. Алгоритм подсчитывает количество таких циклов — это и есть минимальное количество копилок, которые необходимо разбить.

Сложность:

Время: O(n), так как каждая вершина и ребро обрабатываются не более одного раза.

Память: O(n), поскольку нужно хранить обратный граф и состояние каждой вершины.

O. Долой списывание!

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

bool checkBipartite(int start, const vector<vector<int>>& graph,
vector<int>& color) {
  queue<int> q;
  color[start] = 1;
  q.push(start);

  while (!q.empty()) {
    int v = q.front();
    q.pop();
    for (int u : graph[v]) {
      if (color[u] == 0) {
        color[u] = 3 - color[v];
        q.push(u);
      } else if (color[u] == color[v]) {
        return false;
      }
    }
  }
  return true;
}
```

```cpp
int main() {
  int n, m;
  cin >> n >> m;
  vector<vector<int>> graph(n);
  vector<int> color(n, 0);

  for (int i = 0; i < m; i++) {
    int a, b;
    cin >> a >> b;
    a--;
    b--;
    graph[a].push_back(b);
    graph[b].push_back(a);
  }

  bool isBipartite = true;
  for (int i = 0; i < n; i++) {
    if (color[i] == 0 && !checkBipartite(i, graph, color)) {
      isBipartite = false;
      break;
    }
  }

  cout << (isBipartite ? "YES" : "NO");
  return 0;
}
```

Описание:

Нужно проверить, можно ли разделить вершины графа на две группы так, чтобы каждое ребро соединяло вершины из разных групп, то есть проверить двудольность графа.

Сначала считывается количество вершин (N) и рёбер (M), затем строится неориентированный граф в виде списка смежности. Для каждой вершины хранится цвет: 0 — не окрашена, 1 и 2 — два разных цвета групп.

Далее для каждой связной компоненты запускается обход в ширину (BFS). Начальная вершина окрашивается в 1, её соседи — в 2, и так далее. Если во время обхода встречается ребро, соединяющее вершины одного цвета, граф не двудолен — выводится «NO».

Если обход проходит без конфликтов, значит граф двудолен и выводится «YES».

Сложность:

Время: O(N + M) , Каждый узел и каждое ребро обрабатываются ровно один раз при обходе в ширину (BFS).

Память: O(N + M), хранение графа в виде списка смежности и массива цветов для вершин.