Университет ИТМО

Факультет программной инженерии и компьютерной техники

«Алгоритмы и структуры данных»

**отчет по блоку задач №4 (Яндекс.Контест)**

**Выполнил:**

Студент группы P3210

Федоров Евгений Константинович

**Преподаватель:**

Тараканов Денис Сергеевич

Санкт-Петербург, 2025г.

**Задача №1 «M. Цивилизация»**

**Пояснение к примененному алгоритму:**

Алгоритм решает задачу нахождения кратчайшего пути до пункта назначения, учитывая тип рельефа и его стоимость. То есть у каждого ребра есть свой вес, а значит нужно использовать алгоритм Дейкстры. Начальная клетка добавляется в хип (приоритетную очередь) и на каждом шаге выбирается клетка с наименьшим временем достижения. Из нее производятся переходы во все возможные направления, кроме диагональных, если находим более короткое расстояние, то расстояние и направление обновляются. Как только достигаем нужной точки, из массива восстанавливается весь путь, если до точки добраться невозможно, возвращается -1.

**Алгоритмическая сложность:**

* **По времени**: Алгоритм выполняется со скорость **O(NM log(NM)),** так как имеем матрицу размером N \* M, операции с приоритетной очередью занимают O(log(NM)) времени
* **По памяти**: Так как в худшем случае нам придется обойти каждое ребро, то в очередь попадет вся матрица, а значит - сложность **O(NM)**

Код:

|  |
| --- |
| #include <algorithm>  #include <climits>  #include <iostream>  #include <queue>  #include <vector>  using namespace std;  using Node = pair<int, pair<int, int>>;  int N, M, startRow, startCol, endRow, endCol;  vector<pair<int, int>> dirs;  const vector<char> move\_chars = {'N', 'S', 'W', 'E'};  void dijkstra(  vector<vector<char>>& grid, vector<vector<int>>& dist, vector<vector<char>>& prev\_dir  ) {  priority\_queue<Node, vector<Node>, greater<Node>> pq;  dist[startRow][startCol] = 0;  pq.push({0, make\_pair(startRow, startCol)});  while (!pq.empty()) {  auto [time, pos] = pq.top();  auto [row, col] = pos;  pq.pop();  if (row == endRow && col == endCol)  break;  if (time > dist[row][col])  continue;  for (int i = 0; i < 4; ++i) {  int new\_row = row + dirs[i].first;  int new\_col = col + dirs[i].second;  if (new\_row < 0 || new\_row >= N || new\_col < 0 || new\_col >= M)  continue;  if (grid[new\_row][new\_col] == '#')  continue;  int move\_time = (grid[new\_row][new\_col] == 'W') ? 2 : 1;  int new\_time = time + move\_time;  if (new\_time < dist[new\_row][new\_col]) {  dist[new\_row][new\_col] = new\_time;  prev\_dir[new\_row][new\_col] = i;  pq.push({new\_time, make\_pair(new\_row, new\_col)});  }  }  }  }  string reconstruct\_path(const vector<vector<char>>& prev\_dir) {  string path;  int row = endRow, col = endCol;  while (row != startRow || col != startCol) {  int dir\_idx = prev\_dir[row][col];  auto el = move\_chars[dir\_idx];  path.push\_back(el);  row -= dirs[dir\_idx].first;  col -= dirs[dir\_idx].second;  }  reverse(path.begin(), path.end());  return path;  }  int main() {  cin >> N >> M >> startRow >> startCol >> endRow >> endCol;  dirs.emplace\_back(-1, 0);  dirs.emplace\_back(1, 0);  dirs.emplace\_back(0, -1);  dirs.emplace\_back(0, 1);  startRow--;  startCol--;  endRow--;  endCol--;  vector<vector<char>> grid(N, vector<char>(M));  for (auto& row : grid) {  for (auto& cell : row) {  cin >> cell;  }  }  if (grid[startRow][startCol] == '#' || grid[endRow][endCol] == '#') {  cout << -1 << endl;  return 0;  }  vector<vector<int>> dist(N, vector<int>(M, INT\_MAX));  vector<vector<char>> prev\_dir(N, vector<char>(M, -1));  dijkstra(grid, dist, prev\_dir);  if (dist[endRow][endCol] == INT\_MAX) {  cout << -1 << endl;  } else {  cout << dist[endRow][endCol] << endl;  cout << reconstruct\_path(prev\_dir) << endl;  }  return 0;  } |

**Задача №2 «N. Свинки-Копилки»**

**Пояснение к примененному алгоритму:**

По условию задачи мы имеем функциональный граф. Чтобы открыть какую-либо копилку, нужно найти ключ от нее, который в свою очередь может храниться в другой копилке. Соответственно, задача сводится к поиску цикла в функциональном графе. То есть, если несколько копилок «ссылаются» друг на друга, мы не можем открыть никакую из них, пока не разобьем хотя бы одну.

**Алгоритмическая сложность:**

1. **По времени:** **O(n)**, так как обходим весь граф, посещая каждый узел ровно один раз.
2. **По памяти:** **O(n),** так как в худшем случае для хранения всех узлов.

Код:

|  |
| --- |
| #include <iostream>  #include <stack>  #include <unordered\_set>  #include <vector>  using namespace std;  bool dfs(const vector<int>& graph, unordered\_set<int>& visited, int start) {  stack<int> st;  unordered\_set<int> currentPath;  st.push(start);  currentPath.insert(start);  bool cycleFound = false;  while (!st.empty()) {  int node = st.top();  int next = graph[node] - 1;  if (currentPath.count(next)) {  while (!st.empty()) {  int cycle\_node = st.top();  visited.insert(cycle\_node);  st.pop();  if (cycle\_node == next)  break;  }  cycleFound = true;  break;  }  if (visited.count(next)) {  visited.insert(node);  break;  }  currentPath.insert(next);  st.push(next);  }  return cycleFound;  }  int main() {  int n;  cin >> n;  vector<int> graph(n);  unordered\_set<int> visited;  int cycles = 0;  for (int i = 0; i < n; ++i) {  cin >> graph[i];  }  for (int i = 0; i < n; ++i) {  if (!visited.count(i)) {  if (dfs(graph, visited, i)) {  cycles++;  }  }  }  cout << cycles << endl;  return 0;  } |

**Задача №3 «O. Долой списывание»**

**Пояснение к примененному алгоритму:**

Суть задачи сводится к построению двудольного графа. То есть каждый узел относится к какой-либо группе и не может относиться к другой. Достаточно раскрашивать каждый узел в противоположный относительно его соседа цвет, пока не встретим конфликт, либо не раскрасим все узлы.  
Так как нужно смотреть на соседей, то воспользуемся обходом в ширину.

**Алгоритмическая сложность:**

1. **По времени:** **O(n + m)**, так как это стандартный обход в ширину.
2. **По памяти:** **O(n + m),** так как массив graph – список смежности размером n\*m.

Код:

|  |
| --- |
| #include <iostream>  #include <queue>  #include <vector>  using namespace std;  bool hasCycle(vector<vector<int>> graph) {  size\_t n = graph.size();  vector<int> color(n, -1);  for (size\_t i = 0; i < n; i++) {  if (color[i] == -1) {  color[i] = 0;  queue<int> q;  q.push(i);  while (!q.empty()) {  int val = q.front();  q.pop();  for (int j : graph[val]) {  if (color[j] == -1) {  color[j] = color[val] ^ 1;  q.push(j);  } else {  if (color[val] == color[j]) {  return false;  }  }  }  }  }  }  return true;  }  int main() {  int n, m;  cin >> n >> m;  vector<vector<int>> graph(n);  for (int i = 0; i < m; ++i) {  int u, v;  cin >> u >> v;  u--;  v--;  graph[u].push\_back(v);  graph[v].push\_back(u);  }  if (hasCycle(graph)) {  cout << "YES" << endl;  } else {  cout << "NO" << endl;  }  return 0;  } |