МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

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 P3218
Рамеев Тимур Ильгизович
Преподаватель:
Косяков М. С.
Тараканов Д. С.

Санкт-Петербург 2024

Содержание

Цивилизация

Исходный код

```
#include <iostream>
#include <limits.h>
#include <queue>
#include <unordered_map>
#include <list>

using namespace std;

struct Node {
	int type;
	char previous_step;
	bool was_visit = false;
	int short_way = INT_MAX;
	list<int> neighbors = {};
};
unordered_map<int, Node> nodes;
class Compare {
public:
	bool operator()(int a, int b) {
		if (nodes[a].short_way > nodes[b].short_way) {
			return true;
		}
		return false;
	}
};
priority_queue<int, vector<int>, Compare> front_line;
int main() {
	int x, y, begin_x, begin_y, end_x, end_y;
	cin >> y >> x >> begin_y >> begin_x >> end_y >> end_x;
	for (int i = 0; i < y; i++) {
		string current_string;
		cin >> current_string;
		for (int j = 0; j < x; j++) {
			if (current_string[j] == '.') {
				nodes[i * x + j + 1] = { .type = 1 };
			}
			if (current_string[j] == 'W') {
				nodes[i * x + j + 1] = { .type = 2 };
			}
		}
	}
	for (int i = 1; i <= x * y; i++) {
		if (nodes.contains(i)) {
			if (nodes.contains(i - 1) && (i - 1) % x != 0) {
				nodes[i].neighbors.push_back(i - 1);
			}
			if (nodes.contains(i + 1) && (i + 1) % x != 1) {
				nodes[i].neighbors.push_back(i + 1);
			}
			if (nodes.contains(i + x)) {
				nodes[i].neighbors.push_back(i + x);
			}
			if (nodes.contains(i - x)) {
				nodes[i].neighbors.push_back(i - x);
			}
		}
	}
	front_line.push(begin_x + (begin_y - 1) * x);
	nodes[begin_x + (begin_y - 1) * x].short_way = 0;
	nodes[begin_x + (begin_y - 1) * x].was_visit = true;
	while (!front_line.empty()) {
		int current_top_node = front_line.top();
```

```
			front_line.pop();
			for (int node_number : nodes[current_top_node].neighbors) {
				if (nodes[node_number].short_way > nodes[current_top_node].short_way +
nodes[node_number].type) {
					nodes[node_number].short_way = nodes[current_top_node].short_way +
nodes[node_number].type;

					if (node_number - current_top_node == x) {
						nodes[node_number].previous_step = 'S';
					}
					if (node_number - current_top_node == -x) {
						nodes[node_number].previous_step = 'N';
					}
					if (node_number - current_top_node == -1) {
						nodes[node_number].previous_step = 'W';
					}
					if (node_number - current_top_node == 1) {
						nodes[node_number].previous_step = 'E';
					}
				}
				if (nodes[node_number].was_visit == false) {
					front_line.push(node_number);
					nodes[node_number].was_visit = true;
				}
			}
		}
		int current_node = end_x + (end_y - 1) * x;
		int result_way = nodes[current_node].short_way;
		string result_str = "";
		if (result_way == INT_MAX) {
			cout << -1;
			return 0;
		}
		while (current_node != begin_x + (begin_y - 1) * x) {
			switch (nodes[current_node].previous_step) {
			case 'S':
				result_str = 'S' + result_str;
				current_node -= x;
				break;
			case 'N':
				result_str = 'N' + result_str;
				current_node += x;
				break;
			case 'W':
				result_str = 'W' + result_str;
				current_node += 1;
				break;
			case 'E':
				result_str = 'E' + result_str;
				current_node -= 1;
				break;
			}
		}
		cout << result_way << endl;
		cout << result_str;
		return 0;
}
```

Пояснения к примененному алгоритму

Входные данные:

- x – количество клеток вдоль оси x
- y – количество клеток вдоль оси y
- begin_x – начальная координата по x
- begin_y – начальная координата по y
- end_x – конечная координата по x
- end_y – конечная координата по y
- nodes –содержит структуру Node, олицетворяющую клетку. Структура содержит поле type, для определения типа клетки, в ней могут быть только поле и лес; поле previous_step для определения минимального пути, а также стандартный набор любой вершины графа neighbors – соседи, was_visit для прохода по графу short_way – самый короткий путь к вершине от начальной.

Алгоритм выполнения:

Так как граф ориентированный, будет удобно применить алгоритм Дейкстры для поиска кратчайшего пути. Собственно все решение задачи сводится именно к этому. Для реализации алгоритма использовал очередь PriorityQueue с переопределенным компаратором, который располагал вершины в порядке не убывания кратчайшего расстояния до них. После нахождения размера кратчайшего пути пробегаемся по всем предкам благодаря полю previous_step и благодаря этому строим строку resultstr, содержащую крачайший путь от начального поля до конечного. В целом все.

Сложность:

$$O((x * y)^2 \log(x * y))$$

Свинки-копилки

Исходный код

```cpp
#include <iostream>
#include <unordered_map>
#include <vector>

using namespace std;

struct Node {
	int color = 1000;
	int parent;
};

int main() {

	unordered_map<int, Node> nodes;
	int n, colors_counter = 0;
	cin >> n;

	for (int i = 1; i <= n; i++) {
		int current_number;
		cin >> current_number;
		nodes[i] = { .parent = current_number };
	}

	for (int i = 1; i <= n; i++) {
		if (nodes[i].color == 1000) {
			colors_counter++;
			nodes[i].color = colors_counter;
			vector<int> childs = { i };
			int current_parent = nodes[i].parent;
			while (nodes[current_parent].color == 1000) {
				nodes[current_parent].color = colors_counter;
				childs.push_back(current_parent);
				current_parent = nodes[current_parent].parent;
			}
			if (nodes[current_parent].color < colors_counter) {
				int new_color = nodes[current_parent].color;
				for (int x : childs) {
					nodes[x].color = new_color;
				}
				colors_counter--;
			}
		}
	}
	cout << colors_counter;
	return 0;
}
```

Пояснения к примененному алгоритму

Входные данные:

- n – количество копилок
- nodes – map размера n, содержащая структуры Node – копилки

Алгоритм выполнения:

Алгоритм прост. Задача сводится к поиску компонент ориентированного графа. Делаем это как нас учили в лекциях через BFS, закрашивая компоненты разным цветом. Стоит заметить, что здесь мы пользуемся тем фактом, что у копилок может быть только один родитель. Под родителем я подразумеваю копилку, содержащую ключик от данной копилки. Результатом работы будет число – количество различных цветов в графе.

Сложность:

$$O(n^2)$$

Долой списывание

Исходный код

```
#include <iostream>
#include <list>
#include <unordered_map>
#include <queue>

using namespace std;

struct Node {
	int color = -1;
	list<int> neighbors = {};
	bool was_visited = false;
};

unordered_map<int, Node> main_map;
queue<int> front_line;

int main() {
	int n, m;
	cin >> n >> m;
	for (int i = 0; i < m; i++) {
		int first, second;
		cin >> first >> second;
		main_map[first].neighbors.push_back(second);
		main_map[second].neighbors.push_back(first);
	}
	for (auto [key, value] : main_map) {
		if (value.was_visited == true) {
			continue;
		}
		front_line.push(key);
		main_map[key].was_visited = true;
		main_map[key].color = 0;
		while (!front_line.empty()) {
			int current_top = front_line.front();
			front_line.pop();
			for (int x : main_map[current_top].neighbors) {
				if (main_map[x].was_visited == false) {
					main_map[x].color = (main_map[current_top].color + 1) % 2;
					main_map[x].was_visited = true;
					front_line.push(x);
				}
			}
		}
	}
	for (auto [key, value] : main_map) {
		for (int x : value.neighbors) {
			if (value.color == main_map[x].color) {
				cout << "NO";
				return 0;
			}
		}
	}
	cout << "YES";
	return 0;
}
```

Пояснения к примененному алгоритму

Входные данные:

- n – количество учеников
- m – количество пар
- main_map – map с ключом – номером ученика и значением – структура Node, которая содержит цвет ученика, лист его соседей и статус was_visited.

Алгоритм выполнения:

Задача простая. Алгоритм сводится к проверке графа на двудольность. Для этого и нужно поле color в структуре Node. Проверка осуществляется алгоритмом BFS, как нас учили в лекциях через очередь. В нем вы пробегаемся по всем вершинам и раскрашиваем их. Далее пробегаемся по всем ребрам и если вдруг мы найдем ребро, которое соединяет вершины одинаковых цветов, выводим NO.

Сложность:

$$O(n * (n + m))$$

Авиаперелеты

Исходный код

```
#include <iostream>
#include <unordered_set>

using namespace std;

int main_arr[1010][1010] = {};
int current_main_arr[1010][1010] = {};
unordered_set<int> passed = {};

int n;

static void dfs(int from) {
	passed.insert(from);
	for (int i = 1; i <= n; i++) {
		if (!passed.contains(i) && current_main_arr[from][i] != -1) {
			dfs(i);
		}
	}
}

static void reverse_dfs(int from) {
	passed.insert(from);
	for (int i = 1; i <= n; i++) {
		if (current_main_arr[i][from] != -1 && !passed.contains(i)) {
			reverse_dfs(i);
		}
	}
}

int main() {

	cin >> n;

	for (int i = 1; i <= n; i++) {
		for (int j = 1; j <= n; j++) {
			cin >> main_arr[i][j];
		}
	}

	int left = 0;
	int right = 1000000000;

	while (left != right) {
		int middle = (left + right) / 2;
		for (int i = 1; i <= n; i++) {
			for (int j = 1; j <= n; j++) {
				if (main_arr[i][j] <= middle) {
					current_main_arr[i][j] = main_arr[i][j];
				}
				else {
					current_main_arr[i][j] = -1;
				}
			}
		}
		passed.clear();
		dfs(1);
		if (passed.size() != n) {
			left = middle + 1;
			continue;
		}
		passed.clear();
		reverse_dfs(1);
```

```
            if (passed.size() != n) {
                left = middle + 1;
                continue;
            }
            right = middle;
        }
        cout << right;

        return 0;
}
```

Пояснения к примененному алгоритму

Входные данные:

- n – количество городов
- main_arr – двумерный массив, содержащий пути из городов в другие города размера n*n

Алгоритм выполнения:

Несмотря на 55 попыток решить эту задачу, алгоритм оказался не таким уж и сложным. В нем мы пользуемся свойством сильной связности ориентированного графа о том, что если из одной вершины есть путь в другие вершины и из всех остальных вершин графа есть путь в эту вершину, то граф сильно связан.

Далее мы бинарным поиском ищем минимально подходящую длину ребра графа, чтобы он оставался сильно связанным. В цикле рассматривается вершина с индексом 1 (самая первая). Для перебора всех путей из нее используется DFS. Для учета пройденных вершин используем unordered_set passed, в который кладем индексы пройденных вершин. Если по окончании DFS размер passed совпадает с n ребро валидно, иначе – ребро должно быть больше. Но это только первая часть, дальше нужно проверить, доступность вершины 1 из остальных вершин. Для этого используем так называемый reverse_DFS в котором мы рекурсивно бежим не от родителя к сыну, а от сына к родителю. Таким образом мы пройдемся по всем вершинам, из которых можно попасть в вершину 1. Для учета опять используем passed. Если размер совпадает – размер ребра валиден. Вот и все!

Сложность:

$$O(\log(10^9) \cdot n^2)$$