ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«Национальный исследовательский университет ИТМО»**

**Факультет программной инженерии и компьютерной техники**

«Алгоритмы и структуры данных»

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

**Выполнил:**

Студент группы P3215

Барсуков М.А.

**Преподаватель:**

Косяков М.С.

Санкт-Петербург, 2024

**Задачи из 4-го блока Яндекс.Контеста**

**Задача №13 «M. Цивилизация»**

Описание:

На прямоугольном поле *N* x *M* в клетках есть поле, лес и вода. Переходить можно только в 4 граничащие с текущим квадратика. Перейти на поле в два раза быстрее, чем в лес, а по воде ходить нельзя. Нужно найти самый быстрый маршрут из одной клетки в другую и вывести этот его.

***Доказательство:***

Для решения задачи нам нужно найти кратчайший путь во взвешенном (так как стоимость перемещения по разным типам клеток различается) графе. Ребер отрицательного веса у нас нет, так что будем использовать **алгоритм Дейкстры** с использованием очереди с приоритетом.

Если точки отправления и прибытия совпадают, то ответ будет 0. Иначе создаем матрицу стоимости достижения каждой клетки из начальной точки vector<vector<int>> cost. Изначально все клетки имеют стоимость INF, кроме начальной, стоимость которой равна 0. Кроме того, для восстановления пути поселенца в виде последовательности символов сохраняем информацию о том, откуда поселенец пришел в каждую клетку – будем использовать vector<vector<pair<int, int>>> parent чтобы для каждой клетки хранить «родителя».

Для обхода соседей в алгоритме используем приоритетную очередь с компаратором по возрастанию стоимости клеток. Каждая клетка в очереди представляет собой объект, содержащий текущую стоимость пути до нее, а также ее координаты. Вершина клетки отправления имеет цену 0, добавляем её в очередь.

В основном цикле алгоритма Дейкстры (пока очередь не опустеет) удаляем верхнюю вершину. Если она совпадает с вершиной назначения, то мы выходим из цикла – мы достигли точки назначения. Иначе проверяем соседние вершины: если клетка в пределах карты и не вода, то считаем её стоимость, учитывая местность перехода (если ‘.’, то поле 1, если ‘W’, то лес 2). Если текущий путь более дешевый – обновляем стоимость и «родителя», добавляем новую вершину в очередь. Готово, алгоритм Дейкстры выполнен.

Если стоимость вершины назначения осталась бесконечной, значит дойти из начальной клетки в конечную невозможно – выводим -1. Иначе выводим стоимость этой клетки (то есть искомое количество единиц времени) и формируем строку, задающую маршрут поселенца. Начиная от клетки назначения, переходим по соответствующим клеткам из parent, после чего инвертируем строку и получаем нужную последовательность переходов поселенца – **задача решена**.

Алгоритмическая сложность:

1. **По времени:** Каждая клетка карты представляет собой вершину графа. Всего вершин *V = N\*M*. Каждая вершина может иметь до четырёх рёбер (соседи по сторонам), таким образом, общее число рёбер *E* приблизительно равно *4V = 4N \* M*. Рассмотрим операции вставки и извлечения из очереди с приоритетами. В наихудшем случае каждая вершина (клетка карты) может быть добавлена в очередь при каждом рассмотрении её соседей: добавление в очередь (push) имеет сложность O(logV), где *V* — число вершин, извлечение из очереди (pop) также имеет сложность O(logV). В худшем случае, каждая клетка может быть обработана **до четырёх раз** (один раз за каждого из четырёх соседей), что приводит к частым обновлениям приоритетов. Итого, каждая push или pop операции стоят O(logV), и если учесть, что каждая вершина может быть рассмотрена несколько раз, мы имеем общую временную сложность порядка O((V + E) \* (log V)), что в худшем случае даёт O((NM + 4NM) \* log(NM)) = O(NM \* log(NM)).
2. **По памяти:** Программа хранит некоторое постоянное число переменных (n, m, src, dest, dx, dy), а также карту размера *N\*M* в виде вектора строк, что требует O(N\*M) памяти, матрицу стоимостей cost, требующую O(N\*M) памяти, матрицу родителей parent, которая также требует O(N\*M) памяти и очередь с приоритетами, которая в худшем может содержать все клетки карты, т.е. O(N\*M). Это в худшем случае даёт пространственную сложность O(N\*M).

**Итого:**

* **По времени**: O((V + E) \* log V) = O(NM \* log(NM)) (линейная логарифмическая сложность, если рассматривать *NM* как одну переменную, что является типичной оценкой для многих алгоритмов, работающих с графами).
* **По памяти**: O(V) = O(NM) (линейная сложность).

Код:

1. #include <bits/stdc++.h>
2. using namespace std;
3. #define repeat(times, i) for (int (i) = 0; (i) < (times); (i)++)
4. #define unless(cond) if (!(cond))
5. const int INF = numeric\_limits<int>::max();
6. int main() {
7. int n, m;
8. cin >> n >> m;
9. pair<int, int> src, dest; *// (x, y)*
10. cin >> src.first >> src.second >> dest.first >> dest.second;
11. --src.first; --src.second; --dest.first; --dest.second;
12. if (src.first == dest.first && src.second == dest.second) {
13. cout << 0 << endl;
14. return 0;
15. }
16. vector<string> map(n);
17. repeat(n, i) cin >> map[i];
18. vector<vector<int>> cost(n, vector<int>(m, INF));
19. vector<vector<pair<int, int>>> parent(n, vector<pair<int, int>>(m, {-1, -1}));
20. auto comp = [&](const pair<int, int>& a, const pair<int, int>& b) {
21. return cost[a.first][a.second] > cost[b.first][b.second];
22. };
23. priority\_queue<pair<int, int>, vector<pair<int, int>>, decltype(comp)> pq(comp);
24. cost[src.first][src.second] = 0;
25. pq.push(src);
26. int dx[] = {1, -1, 0, 0};
27. int dy[] = {0, 0, 1, -1};
28. char dir[] = {'N', 'S', 'W', 'E'};
29. while (!pq.empty()) {
30. auto [x, y] = pq.top();
31. pq.pop();
32. if (x == dest.first && y == dest.second) break;
33. repeat(4, d) {
34. int nx = x + dx[d], ny = y + dy[d];
35. if (nx >= 0 && nx < n && ny >= 0 && ny < m && map[nx][ny] != '#') {
36. int new\_cost = cost[x][y] + (map[nx][ny] == '.' ? 1 : (map[nx][ny] == 'W' ? 2 : 0));
37. if (new\_cost < cost[nx][ny]) {
38. cost[nx][ny] = new\_cost;
39. parent[nx][ny] = {x, y};
40. pq.push({nx, ny});
41. }
42. }
43. }
44. }
45. if (cost[dest.first][dest.second] == INF) {
46. cout << "-1" << endl;
47. return 0;
48. }
49. cout << cost[dest.first][dest.second] << endl;
50. string path;
51. int cx = dest.first, cy = dest.second;
52. while (cx != src.first || cy != src.second) {
53. auto [px, py] = parent[cx][cy];
54. repeat(4, d) {
55. if (px == cx + dx[d] && py == cy + dy[d]) {
56. path.push\_back(dir[d]);
57. break;
58. }
59. }
60. cx = px;
61. cy = py;
62. }
63. reverse(path.begin(), path.end());
64. cout << path << endl;
65. return 0;
66. }

**Задача №14 «N. Свинки-копилки»**

Описание:

Есть *n* копилок, которые можно разбить или открыть ключом. Ключи лежат в некоторых из копилок. Нужно получить содержимое всех копилок. Какое минимальное количество копилок необходимо разбить?

***Доказательство:***

По условию задачи копилки с ключами образуют некоторые группы, в которых, разбив одну копилку, можно получить ключ и, открывая всё новые и новые копилки, получить содержимое всей группы. Очевидно, что таких групп может быть несколько. Тогда минимальное число свинок-копилок, которые нужно разбить, чтобы достать ключи из всех копилок – это количество таких групп копилок, связи (то есть ключи от других копилок) между которыми есть только в этой группе.

Таким образом, задачу можно сформулировать как задачу о **поиске количества компонент связности в графе**, где узлы графа — это копилки, а ребра указывают, какой ключ где лежит. Каждая компонента связности обозначает группу копилок, в которой можно достать все ключи, начиная с разбивания одной копилки в этой компоненте. Итак, минимальное количество копилок, которые необходимо разбить, равно количеству компонент связности в графе, потому что в каждой компоненте связности достаточно «разбить» одну копилку, чтобы получить ключи для всех остальных копилок в той же компоненте.

Сначала мы считываем общее количество копилок *n* и создаем граф в виде списка смежности. Для каждой копилки создаётся вектор, где graph[i] содержит индексы копилок, с которыми копилка i связана ключами. То есть, если ключ от копилки i лежит в копилке j, то в граф добавляются два ребра: i -> j и j -> i, указывающие взаимосвязь между копилками. Также используем массив broken, индексируемый номерами копилок, который показывает, была ли копилка уже «разбита» (посещена).

Далее для каждой копилки (если она ещё не была «разбита»), мы будем обходить все связанные с ней копилки в глубину и разбивать их. Все разбитые за эту итерацию копилки – это копилки одной группы, то есть компонента связности графа. Таким образом **за каждую итерацию по неразбитым копилкам мы находим еще одну компоненту связности графа**. Очевидно, что, посчитав количество таких итераций, мы узнаем количество компонент связности в графе – это и есть искомое минимальное количество копилок, которые необходимо разбить.

Алгоритмическая сложность:

1. **По времени:** Во время ввода данных мы проходим через *n* копилок и для каждой устанавливает двунаправленное ребро между копилками. Это занимает константное время amortized O(1) на каждую итерацию (добавление элемента в вектор), и общее время для всех копилок будет O(n). Во время обхода графа и поиска компонент связности, каждая вершина (копилка) и каждое ребро графа будут рассмотрены ровно один раз в процессе поиска в глубину. Так как у каждой копилки максимум два связанных ребра (отношение ключей), общее количество рёбер в графе будет порядка O(n). Таким образом, общее время, затрачиваемое на поиск в глубину для обхода всех вершин и рёбер, составит O(V + E), где *V* — количество вершин (копилок), а *E* — количество рёбер. Поскольку *V = n* и *E ≈ n*, временная сложность в среднем и худшем случаях будет O(n).
2. **По памяти:** Программа хранит граф в виде вектора векторов, где каждый элемент верхнего уровня соответствует копилке. Поскольку каждая копилка связывается с двумя другими копилками (одна связь находится в другой копилке, одна — где ключ от этой копилки), общее количество элементов в структуре данных составляет примерно *2n*, но основное хранилище ограничивается *n* векторами. Также хранится массив broken, используемый для отметки посещенных копилок, который содержит *n* элементов. Кроме того, программа хранит некоторое постоянное число переменных (n, value, broken\_count, i), Таким образом, в среднем и худшем случае пространственная сложность составляет O(n).

**Итого:**

* По времени: O(n) (линейная сложность).
* По памяти: O(n) (линейная сложность).

Код:

1. #include <bits/stdc++.h>
2. using namespace std;
3. #define repeat(times, i) for (int (i) = 0; (i) < (times); (i)++)
4. #define unless(cond) if (!(cond))
5. void dfs(int v, bool\* broken, vector<vector<int>>\* graph) {
6. broken[v] = true;
7. for (int i : (\*graph)[v]) {
8. unless(broken[i]) dfs(i, broken, graph);
9. }
10. }
11. int main() {
12. int n, value, broken\_count = 0;
13. cin >> n;
14. vector<vector<int>> graph(n);
15. repeat(n, i) {
16. cin >> value;
17. graph[value - 1].push\_back(i);
18. graph[i].push\_back(value - 1);
19. }
20. bool broken[n];
21. fill(broken, broken + n, false);
22. repeat(n, i) {
23. unless (broken[i]) {
24. ++broken\_count;
25. dfs(i, broken, &graph);
26. }
27. }
28. cout << broken\_count << endl;
29. return 0;
30. }

**Задача №15 «O. Долой списывание!»**

Описание:

Есть пары лкшат, обменивающихся записками. Нужно определить, можно ли разделить их на две группы так, чтобы любой обмен записками осуществлялся от лкшонка одной группы лкшонку другой группы.

***Доказательство:***

По условию задачи у нас есть граф, где вершина – это лкшонок, а ребро – это обмен запиской. Требуется проверить, что можно разделить вершины на группы таким образом, чтобы между вершинами каждой группы не было ребер. Таким образом, задачу можно смоделировать как задачу о **проверке графа на двудольность**. *Двудольный граф* — это граф, вершины которого можно разделить на два множества таким образом, чтобы рёбра соединяли только вершины из разных групп. В контексте задачи это означает, что ученики могут быть разделены на две группы — одна группа «списывает», а другая «даёт списывать».

Часто в контексте двудольных графов используется понятие **цвета вершины**. *Хроматическим числом* графа называется минимальное количество цветов, в которые можно покрасить его вершины так, чтобы каждое ребро соединяло вершины различного цвета. Хроматическое число двудольных графов равно 2. Поэтому для проверки графа на двудольность и разбития его на доли будем использовать **покраску** его вершин в **два различных цвета**. Так, предположим, есть 4 ученика и 3 пары обменов записками (1–2, 2–3, 3–4). Граф будет двудольным, так как ученики могут быть разделены на два цвета: {1, 3} и {2, 4}, где обмены происходят между этими группами. Если добавить еще одну связь (4–1), граф станет недвудольным, так как образуется цикл с нечётным числом вершин, и код выведет "NO".

Представим граф списком смежности, где graph[i] содержит список учеников, с которыми ученик i обменивается записками. Введём вектор цветов по количеству учеников (вершин графа), изначально вершины не покрашены. Чтобы проверить, что граф двудольный, пройдемся по всем вершинам. Если вершина не покрашена, красим её в первый цвет и раскашиваем связанные с ней вершины поиском в глубину. В процессе поиска в глубину, при прохождении по каждому ребру красим следующую вершину в противоположный цвет. Если при переборе соседних вершин мы нашли вершину, уже покрашенную в тот же цвет, что и текущая, **то он не является двудольным** (так как образуется цикл с нечётным числом вершин), и функция возвращает false, то есть выводится "NO". Если таких «конфликтов» в раскрашивании не возникло, значит граф двудольный – учеников можно разделить на две группы так, чтобы любой обмен записками осуществлялся от лкшонка одной группы лкшонку другой группы – выводим "YES".

Алгоритмическая сложность:

* **По времени:** Для каждой из *m* пар происходит добавление в список смежности двух направлений (оба направления, так как граф неориентированный). Добавление каждой связи в вектор выполняется за amortized O(1), поэтому общая сложность этого шага будет O(m). Для проверки на двудольность используется обход в глубину (DFS). Поскольку каждая вершина и каждое ребро посещаются ровно один раз, сложность DFS зависит от количества вершин *n* и рёбер *m*, что дает общая временная сложность алгоритма O(n + m), где *n* — количество учеников (вершин), а *m* — количество пар обмена записками (рёбер).
* **По памяти:** Программа хранит некоторое постоянное число переменных (n, m, pupil1, pupil2, it, i), а также массив цветов размером n и граф, представленный списком смежности, максимальное количество элементов которого составляет 2m для каждого из n векторов, т.е. пространственная сложность графа O(n + 2m) ~ O(n + m). Значит, общая пространственная сложность алгоритма также составляет O(n + m).

**Итого:**

* По времени: O(n + m) (линейная сложность).
* По памяти: O(n + m) (линейная сложность).

Код:

1. #include <bits/stdc++.h>
2. using namespace std;
3. #define repeat(times, i) for (int (i) = 0; (i) < (times); (i)++)
4. #define unless(cond) if (!(cond))
5. bool bipartite\_dfs(vector<vector<int>>& graph, int node, vector<int>& color) {
6. for (auto it : graph[node]) {
7. if (color[it] == -1) {
8. color[it] = 1 - color[node];
9. unless (bipartite\_dfs(graph, it, color)) return false;
10. } else if(color[it] == color[node]) {
11. return false;
12. }
13. }
14. return true;
15. }
16. bool is\_bipartite(vector<vector<int>>& graph, int n) {
17. vector<int> color(n, -1);
18. repeat(n, i) {
19. if (color[i] == -1) {
20. color[i] = 1;
21. unless (bipartite\_dfs(graph, i, color)) return false;
22. }
23. }
24. return true;
25. }
26. int main() {
27. int n, m;
28. cin >> n >> m;
29. vector<vector<int>> graph(n);
30. int pupil1, pupil2;
31. repeat(m, i) {
32. cin >> pupil1 >> pupil2;
33. graph[pupil1 - 1].push\_back(pupil2 - 1);
34. graph[pupil2 - 1].push\_back(pupil1 - 1);
35. }
36. cout << (is\_bipartite(graph, n) ? "YES" : "NO") << endl;
37. return 0;
38. }

**Задача №16 «P. Авиаперелёты»**

Описание:

Для самолета существует определенный расход топлива для перелёта между каждой парой городов. Нужно найти минимально возможный размер бака, для которого самолёт сможет долететь от любого города в любой другой (возможно, с дозаправками в пути).

***Доказательство:***

Сначала считаем количество городов n и инициализируем двумерные вектора graph (для хранения расхода топлива между городами) и fits\_fuel (для флагов, показывающих, возможен ли перелет при текущем топливе).

Для того, чтобы определить минимальный размер бака, будем использовать **бинарный поиск** (как в задаче [«E. Коровы в стойла»](https://contest.yandex.ru/contest/35179/problems/E/)): минимумом в нем будет 0, а максимумом – **максимальное расстояние между парой городов** (очевидно, что в этом случае самолет может попасть из любого города в другой с дозаправкой, так как может преодолеть расстояние между любыми двумя связанными городами). Необходимо только написать функцию проверки достижимости всех городов при заданном уровне топлива.

Реализуем её так: заполним матрицу fits\_fuel, где указывается, возможен ли перелет между каждой парой городов при заданном объеме топлива. Учитывая это, **проверим достижимость всех городов** из одной точки (are\_all\_visited), сначала в одном направлении, а затем в другом, чтобы убедиться, что любой город может быть достигнут из любого другого. Чтобы это проверить, используем **поиск в глубину** начиная с первого города, рекурсивно переходя к новому городу только если перелет возможен по матрице fits\_fuel. Если все города посещены, то данное количество топлива подходит и можно попытаться уменьшить его в бинарном поиске, если не посещены – то нужно увеличить количество топлива.

Результатом бинарного поиска является **минимальный необходимый объем топлива**. Таким образом, этот подход обеспечивает эффективную проверку минимально необходимого объема топлива для перелетов между городами, учитывая возможность дозаправок в пути.

Алгоритмическая сложность:

1. **По времени:** Заполнение матриц graph и fits\_fuel выполняется за O(n2). Функция are\_all\_visited запускает DFS дважды (для проверки связности в прямом и обратном направлениях), каждый вызов DFS потенциально затрагивает все вершины и ребра, что занимает O(n2), так как в худшем случае нужно проверить все соединения. Бинарный поиск выполняется в диапазоне от 0 до *M*, где *M* – максимальное расстояние между двумя связанными городами, то есть имеет сложность O(log(M)). Поскольку в худшем случае max\_fuel могло быть равно максимальному значению в матрице graph, количество шагов бинарного поиска может быть *log(109) ≈ 30*. На каждом шаге бинарного поиска вызывается can\_travel\_with\_fuel, которая, в свою очередь, работает за O(n2). Таким образом, в среднем и худшем случаях временная сложность – O(log(M)\*n2).
2. **По памяти:** Программа использует фиксированное количество переменных (n, max\_fuel\_between\_cities, i, j, low, high, mid и т.д.), а также двумерные векторы graph и fits\_fuel, в которых *n2* элементов и массив visited длинной *n* символов, поэтому сложность по памяти составляет O(n2).

**Итого:**

* По времени: O(log(M)\*n2) (квадратичная сложность с поправкой на то, что M можно считать постоянной по максимальному её значению 109).
* По памяти: O(n2) (квадратичная сложность).

Код:

1. #include <bits/stdc++.h>
2. using namespace std;
3. #define repeat(times, i) for (int (i) = 0; (i) < (times); (i)++)
4. #define unless(cond) if (!(cond))
5. vector<vector<int>> graph;
6. vector<vector<int>> fits\_fuel;
7. void dfs(int v, vector<bool>& visited, bool direction, int n) {
8. visited[v] = true;
9. repeat(n, i) {
10. if (visited[i]) continue;
11. if (fits\_fuel[direction ? i : v][direction ? v : i]) {
12. dfs(i, visited, direction, n);
13. }
14. }
15. }
16. bool are\_all\_visited(int n, bool direction) {
17. vector<bool> visited(n, false);
18. dfs(0, visited, direction, n);
19. repeat(n, i) {
20. unless(visited[i]) return false;
21. }
22. return true;
23. }
24. bool can\_travel\_with\_fuel(int fuel, int n) {
25. repeat(n, i) {
26. repeat(n, j) fits\_fuel[i][j] = (graph[i][j] <= fuel);
27. }
28. if (are\_all\_visited(n, false)) {
29. return are\_all\_visited(n, true);
30. }
31. return false;
32. }
33. int binary\_search\_fuel(int max\_fuel, int n) {
34. int low = 0;
35. int high = max\_fuel;
36. while (low < high) {
37. int mid = (low + high) / 2;
38. if (can\_travel\_with\_fuel(mid, n)) high = mid;
39. else low = mid + 1;
40. }
41. return low;
42. }
43. int main() {
44. int n;
45. cin >> n;
46. graph.resize(n, vector<int>(n));
47. fits\_fuel.resize(n, vector<int>(n));
48. int max\_fuel = 0;
49. repeat(n, i) {
50. repeat(n, j) {
51. cin >> graph[i][j];
52. unless (i == j) max\_fuel = max(max\_fuel , graph[i][j]);
53. }
54. }
55. cout << binary\_search\_fuel(max\_fuel , n) << endl;
56. return 0;
57. }