Университет ИТМО

Факультет программной инженерии и компьютерной техники

**Лабораторная работа №4**

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнил:

Студент группы P3233

Фамилия И.О.

Шикунов Максим Евгеньевич

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2024

Задача №M «Цивилизация»

Код:

#include <iostream>  
#include <vector>  
#include <queue>  
using namespace std**;**const int MAX\_SIZE = (1 << 31) - 1**;**int main() {  
 int n**,** m**,** x1**,** y1**,** x2**,** y2**,** newDistance**;** string input**,** output = ""**;** bool check**;** cin >> n >> m >> x1 >> y1 >> x2 >> y2**;** int massive[n][m]**;** for (int i = 0**;** i < n**;** i++) {  
 cin >> input**;** for (int j = 0**;** j < m**;** j++) {  
 if (input[j] == '.') {  
 massive[i][j] = 1**;** } else if (input[j] == 'W') {  
 massive[i][j] = 2**;** } else {  
 massive[i][j] = -1**;** }  
 }  
 }  
 int size = n \* m**;** pair<int**,** int> minDistance[size]**;** vector<bool> visited(size**,** false)**;** vector<vector<pair<int**,** int>>> connections(size**,** vector<pair<int**,** int>>(4**,** pair(0**,** 0)))**;** for (int i = 0**;** i < n**;** i++) {  
 for (int j = 0**;** j < m**;** j++) {  
 if (i != 0) {  
 connections[i \* m + j][0] = pair((i - 1) \* m + j**,** massive[i - 1][j])**;** }  
 if (i != n - 1) {  
 connections[i \* m + j][1] = pair((i + 1) \* m + j**,** massive[i + 1][j])**;** }  
 if (j != 0) {  
 connections[i \* m + j][2] = pair(i \* m + j - 1**,** massive[i][j - 1])**;** }  
 if (j != m - 1) {  
 connections[i \* m + j][3] = pair(i \* m + j + 1**,** massive[i][j + 1])**;** }  
 }  
 }  
 int indexes[4] = {-m**,** m**,** -1**,** 1}**;** for (int i = 0**;** i < size**;** i++) {  
 minDistance[i] = pair(MAX\_SIZE**,** i)**;** }  
 int beginIndex = (x1 - 1) \* m + y1 - 1**;** minDistance[beginIndex] = pair(0**,** beginIndex)**;** priority\_queue<pair<int**,** int>**,** vector<pair<int**,** int>>**,** greater<pair<int**,** int>>> distances**;** distances.push(pair(0**,** beginIndex))**;** pair<int**,** int> now\_pair**;** do {  
 now\_pair = distances.top()**;** if (now\_pair.first != MAX\_SIZE) {  
 for (int i = 0**;** i < 4**;** i++) {  
 if (connections[now\_pair.second][i].second > 0) {  
 newDistance = now\_pair.first + connections[now\_pair.second][i].second**;** if (newDistance < minDistance[connections[now\_pair.second][i].first].first) {  
 minDistance[connections[now\_pair.second][i].first].first = newDistance**;** distances.push(pair(newDistance**,** now\_pair.second + indexes[i]))**;** }  
 }  
 }  
 distances.pop()**;** }  
 } while (!distances.empty())**;** int index = (x2 - 1) \* m + y2 - 1**;** if (minDistance[index].first < MAX\_SIZE) {  
 cout << minDistance[index].first << endl**;** while (minDistance[index].first > 0) {  
 check = true**;** if (index - m >= 0) {  
 if (check && minDistance[index - m].first + connections[index - m][1].second == minDistance[index].first) {  
 output = "S" + output**;** index -= m**;** check = false**;** }  
 }  
 if (index - 1 >= 0) {  
 if (check && minDistance[index - 1].first + connections[index - 1][3].second == minDistance[index].first) {  
 output = "E" + output**;** index--**;** check = false**;** }  
 }  
 if (index + 1 < size) {  
 if (check && minDistance[index + 1].first + connections[index + 1][2].second == minDistance[index].first) {  
 output = "W" + output**;** index++**;** check = false**;** }  
 }  
 if (index + m < size) {  
 if (check & minDistance[index + m].first + connections[index + m][0].second == minDistance[index].first){  
 output = "N" + output**;** index += m**;** }  
 }  
 }  
 cout << output**;** } else {  
 cout << -1**;** }  
 return 0**;**}

Пояснение к примененному алгоритму:

Для решения задачи пользуемся алгоритмом Дейкстры, в котом мы поочередно, начиная с изначальной клетки закрашиваем другие и считаем, сколько минимально времени потребуется для достижения каждой клетки на поле. Потом уже выводим значение конечной клетки. А для нахождения пути мы делаем обратный метод с конца до начала, мы берем клетку и сравниваем ее значение с прилегающими, смотрим от куда мы в принципе могли прийти (время предыдущей + время перехода с нее) и так идем до самого конца

Сложность:

Задача №N «Свинки-копилки»

Код:

#include <iostream>  
#include <vector>  
#include <set>  
#include <map>  
using namespace std**;**struct dot {  
 int number**;** struct dot\* keyTo**;** int size = 0**;** vector<struct dot\*> keyFrom**;**}**;**struct dot\* find(set<int> visited**,** struct dot\* dot) {  
 if (dot->keyTo != nullptr && visited.find(dot->keyTo->number) == visited.end()) {  
 visited.insert(dot->number)**;** return find(visited**,** dot->keyTo)**;** } else {  
 return dot**;** }  
}  
  
  
set<int> visitAllDots(set<int> visited**,** struct dot\* dot) {  
 if (visited.find(dot->number) == visited.end()) {  
 visited.insert(dot->number)**;** } else {  
 return visited**;** }  
 set<int> visits**;** for (int i = 0**;** i < dot->size**;** i++) {  
 visits = visitAllDots(visited**,** dot->keyFrom[i])**;** for (int j : visits) {  
 visited.insert(j)**;** }  
 }  
 return visited**;**}  
  
  
int main() {  
 int dotCount**,** index**,** breakCount = 0**;** cin >> dotCount**;** struct dot\* dotMassive[dotCount]**;** map<int**,** bool> dotAlive**;** for (int i = 0**;** i < dotCount**;** i++) {  
 dotMassive[i] = new struct dot**;** dotAlive[i] = true**;** }  
 for (int i = 0**;** i < dotCount**;** i++) {  
 cin >> index**;** index--**;** dotMassive[i]->keyTo = dotMassive[index]**;** dotMassive[i]->number = i**;** dotMassive[index]->keyFrom.push\_back(dotMassive[i])**;** dotMassive[index]->size++**;** }  
 for (int i = 0**;** i < dotCount**;** i++) {  
 if (dotAlive[i]) {  
 struct dot\* deleteFrom = find(set<int> {}**,** dotMassive[i])**;** set<int> toDelete = visitAllDots(set<int> {}**,** deleteFrom)**;** for (int j : toDelete) {  
 dotAlive[j] = false**;** }  
 breakCount++**;** }  
 }  
 cout << breakCount**;** return 0**;**}

Пояснение к примененному алгоритму:

Так как по задачи у нас от одной свинки в одном направлении идет ребро к другой (от свинки в которой находится ключ другой, от нее ведется к ней ребро), то граф у нас получается такой, что у него не может быть больше одного конечного ребра, от которого можно посетить все графы. Тем самым мы просто находим такие графы, удаляем все найденные цепочки, а потом считаем, сколько делали удалений. Это и есть ответ.

Сложность:

Задача №O «Долой списывание!»

Код:

#include <iostream>  
#include <set>  
#include <vector>  
using namespace std**;**vector<int> colors(100**,** 0)**;**struct dot {  
 int number**;** set<struct dot\*> dotsTo**;**}**;**bool checkGraph(struct dot\* dot**,** int color) {  
 colors[dot->number] = color**;** for (struct dot\* child : dot->dotsTo) {  
 if (colors[child->number] == 0) {  
 if (color == 1) {  
 if (!checkGraph(child**,** 2)) {  
 return false**;** }  
 } else {  
 if (!checkGraph(child**,** 1)) {  
 return false**;** }  
 }  
 } else if (colors[child->number] == color) {  
 return false**;** }  
 }  
 return true**;**}  
  
  
int main() {  
 int input1**,** input2**,** n**,** m**;** bool check**;** cin >> n >> m**;** struct dot\* dotMassive[n]**;** for (int i = 0**;** i < n**;** i++) {  
 dotMassive[i] = new struct dot**;** dotMassive[i]->number = i**;** }  
 for (int i = 0**;** i < m**;** i++) {  
 cin >> input1 >> input2**;** input1--**;** input2--**;** dotMassive[input1]->dotsTo.insert(dotMassive[input2])**;** dotMassive[input2]->dotsTo.insert(dotMassive[input1])**;** }  
  
 for (int i = 0**;** i < n**;** i++) {  
 if (colors[i] == 0) {  
 check = checkGraph(dotMassive[i]**,** 1)**;** }  
 if (!check) {  
 cout << "NO"**;** return 0**;** }  
 }  
 cout << "YES"**;** return 0**;**}

Пояснение к примененному алгоритму:

В данной задаче мы должны построить граф и проверить его на двудольность. Если он двудольный, то выводим “Yes”, иначе “No”. Проверку реализую через алгоритм поиска в глубину. Запускаем проверку с первой вершиной с первым цветом, дальше запускаем проверку всех детей данной вершины уже со вторым цветом, а так же проверяем покрашены ли детские вершины, если они вдруг покрашены в тот же цвет, который сейчас подался в метод, то граф не двудольный.

Сложность:

Задача №P «Авиаперелёты»

Код:

#include <iostream>  
#include <vector>  
#include <set>  
using namespace std**;**int counter**;**void checkProgressNormal(vector<vector<int>>\* edges**,** int checkDistance**,** int nowVertex**,** set<int>\* visited**,** int size) {  
 counter++**;** (\*visited).insert(nowVertex)**;** for (int i = 0**;** i < size**;** i++) {  
 if ((\*visited).find(i) == (\*visited).end() && (\*edges)[nowVertex][i] <= checkDistance) {  
 checkProgressNormal(edges**,** checkDistance**,** i**,** visited**,** size)**;** }  
 }  
}  
  
void checkProgressInverted(vector<vector<int>>\* edges**,** int checkDistance**,** int nowVertex**,** set<int>\* visited**,** int size) {  
 counter++**;** (\*visited).insert(nowVertex)**;** for (int i = 0**;** i < size**;** i++) {  
 if ((\*visited).find(i) == (\*visited).end() && (\*edges)[i][nowVertex] <= checkDistance) {  
 checkProgressInverted(edges**,** checkDistance**,** i**,** visited**,** size)**;** }  
 }  
}  
  
  
bool checkDistance(vector<vector<int>>\* edges**,** set<int>\* visited**,** int checkDistance**,** int size) {  
 counter = 0**;** (\*visited).clear()**;** checkProgressNormal(edges**,** checkDistance**,** 0**,** visited**,** size)**;** if (counter == size) {  
 counter = 0**;** (\*visited).clear()**;** checkProgressInverted(edges**,** checkDistance**,** 0**,** visited**,** size)**;** if (counter == size) {  
 return true**;** }  
 }  
 return false**;**}  
  
  
int main() {  
 int n**,** input**,** left = 0**,** right = 1000000000**,** distance**;** cin >> n**;** vector<vector<int>> edges(n**,** vector<int> (n**,** 0))**;** set<int> visited**;** for (int i = 0**;** i < n**;** i++) {  
 for (int j = 0**;** j < n**;** j ++) {  
 cin >> input**;** edges[i][j] = input**;** }  
 }  
 while (left < right - 1) {  
 distance = (left + right) / 2**;** if (checkDistance(&edges**,** &visited**,** distance**,** n)) {  
 right = distance**;** } else {  
 left = distance**;** }  
 }  
 if (checkDistance(&edges**,** &visited**,** left**,** n)) {  
 cout << left**;** } else {  
 cout << right**;** }  
 return 0**;**}

Пояснение к примененному алгоритму:

Используя бин поиска находим минимальное количество топлива, которого будет достаточно для перелетов по представленным городам. Для проверки, подходим ли нам так бак или нет, мы сначала пробуем пройти в глубину по графу и посетить все вершины, если не удастся, то мы сразу выходим и переписываем левую границу. Если первую проверку прошли, то потом мы проверяем тоже самое, только уже по транспортированной матрице. Данные две проверки гарантируют нам, что мы сможем добраться из любой вершины в любую с таким топливом.

Сложность: