|  |  |  |
| --- | --- | --- |
|  | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра прикладной математики | | |
| по дисциплине «Дискретная математика» | | |
| **Алгоритмы поиска в ширину и глубину** | | |
|  | | |
|  | Бригада 4 | Лезнёв Артём |
| Группа ПМ-15 | Дроздов Даниил |
| Вариант 7 | Мамаев Никита |
|  | Палий Никита |
|  |  |
|  |  |
| Преподаватель | Рояк Михаил эмМануилович |
|  |  |
| Новосибирск, 2022 | | |

1. **Задание**

Определить, существует ли маршрут между двумя заданными вершинами в неориентированном графе. Граф в памяти представлять в виде матрицы.

1. **Распределение задач в бригаде**

Создание тестов для поисков ошибок в программе – Лезнёв Артём

Создание тестов для оценки фактической сложности алгоритма – Мамаев Никита

Программа с использованием поиска в глубину – Дроздов Даниил

Программа с использованием поиска в ширину – Палий Никита

1. **Анализ задачи**

*Входные данные*: в первой строчке v1, v2 - вершины графа, между которыми требуется определить наличие маршрута, далее n строк по n элементов ‘1’ или ‘0’ – матричное представление графа, в котором ‘0’ означает отсутствие ребра, ’1’ – присутствие.

*Выходные данные*: результат проверки: “true”, если между введёнными вершинами графа есть маршрут, или ”false” в обратном случае.

*Метод решения*: решение строится из определения маршрута: «чередующаяся последовательность вершин и рёбер в которой любые два соседних элемента инцидентны». Записываем граф в виде матрицы, в которой на пересечениях строк n со столбцами m находятся значения true, указывающие на наличие ребра между вершинами графа n и m, либо false, указывающие на её отсутствие. Начнем из первой указанной вершины поиск второй указанной вершины. Если её удалось найти, то между вершинами есть маршрут. Чтобы избежать зацикливания, будем отмечать каждую проверенную вершину в коллекции сет.

*Поиск в глубину*: создаём стек для обхода графа. Начинаем обход с первой указанной вершины: проверяем, не является ли она искомой, отмечаем её и добавляем её в стек. Затем, пока стек не стал пуст или искомая вершина не найдена, обрабатываем текущую вершину. Сначала ищем первого неотмеченного соседа текущей вершины. Если он является искомой, то поиск завершен – маршрут найден, возвращаем «true». Если нет, отмечаем его, добавляем текущую вершину в стек и назначаем его новой текущей вершиной. В случае, если такового не найдено, выталкиваем из стека старую текущую вершину и назначаем её текущей. Если обход завершен, а искомая вершина не нашлась – то маршрута нет, возвращаем «false».

*Поиск в ширину*: для алгоритма потребуется очередь. На каждом шагу алгоритм берет из начала очереди вершину и добавляет все неотмеченные смежные с ней вершины в конец очереди, проверяя и отмечая их. Если среди них найдена искомая – маршрут найден, вернуть «true». Если очередь пуста, то алгоритм завершает работу и возвращает «false».

1. **Программа**

/\* DFS \*/

#include <iostream>

#include <stack>

#include <unordered\_set>

const int graphMaxSize = 10000;

struct Graph

{

bool graph[graphMaxSize][graphMaxSize] = {};

int a = 0;

int graphSize = 0;

bool neighbors(int v1, int v2)

{

return graph[v1][v2];

}

bool pathExistsDFS(int v1, int v2)

{

std::unordered\_set<int> visited;

if (v1 >= graphSize || v2 >= graphSize)

return false;

if (v1 == v2)

return true;

visited.insert(v1);

std::stack<int> s;

s.push(v1);

int current = v1;

bool pathFound = false;

while (!s.empty() && !pathFound)

{

int unvisited = 0;

for (int i = 0;

i < graphSize && !(neighbors(current, i) && !visited.count(i));

i++, unvisited++);

if (neighbors(current, unvisited) && !visited.count(unvisited))

{

if (unvisited == v2)

pathFound = true;

else

{

s.push(current);

current = unvisited;

visited.insert(unvisited);

}

} else

{

if (!s.empty())

{

current = s.top();

s.pop();

}

}

}

return pathFound;

}

void input()

{

graphSize = 0;

for (char c = 0;

std::cin.peek() != '\n'; graphSize++, std::cin >> c);

std::cin.ignore();

for (int i = 1; i < graphSize; i++)

{

for (int j = 0; j <= i; j++)

{

std::cin >> graph[i][j];

if (i == j)

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

else

graph[j][i] = graph[i][j];

}

}

}

};

int main()

{

Graph \*g = new Graph();

while (std::cin.peek() != EOF)

{

int v1, v2;

std::cin >> v1 >> v2;

std::cin.ignore();

g->input();

std::cout << g->graphSize << ": ";

bool result = false;

result = g->pathExistsDFS(v1, v2);

std::cout << "dfs: " << (result ? "true" : "false");

}

}

/\* BFS \*/

#include <iostream>

#include <queue>

#include <unordered\_set>

const int graphMaxSize = 10000;

struct Graph

{

bool graph[graphMaxSize][graphMaxSize] = {};

int a = 0;

int graphSize = 0;

bool neighbors(int v1, int v2)

{

return graph[v1][v2];

}

bool pathExistsBFS(int v1, int v2)

{

if (v1 >= graphSize || v2 >= graphSize)

return false;

if (v1 == v2)

return true;

std::unordered\_set<int> visited;

std::queue<int> s;

visited.insert(v1);

int current = v1;

bool pathFound = false;

s.push(v1);

while (!s.empty() && !pathFound)

{

current = s.front();

s.pop();

for (int i = 0; i < graphSize; i++)

{

if (neighbors(current, i) && !visited.count(i))

{

if (i == v2)

pathFound = true;

else

{

s.push(i);

visited.insert(i);

}

}

}

}

return pathFound;

}

void input()

{

graphSize = 0;

for (char c = 0;

std::cin.peek() != '\n'; graphSize++, std::cin >> c);

std::cin.ignore();

for (int i = 1; i < graphSize; i++)

{

for (int j = 0; j <= i; j++)

{

std::cin >> graph[i][j];

if (i == j)

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

else

graph[j][i] = graph[i][j];

}

}

}

};

int main()

{

Graph \*g = new Graph();

while (std::cin.peek() != EOF)

{

int v1, v2;

std::cin >> v1 >> v2;

std::cin.ignore();

g->input();

std::cout << g->graphSize << ": ";

bool result = false;

result = g->pathExistsBFS(v1, v2);

std::cout << "bfs: " << (result ? "true" : "false");

}

}

1. **Тесты для поиска ошибок**

|  |  |  |
| --- | --- | --- |
| **№** | **Входные данные** | **Комментарий** |
| **1** | 0 1 0 0 0 0 0 0 0 0 0 0  1 0 1 0 0 0 0 0 0 0 0 0  0 1 0 1 1 0 0 0 0 0 0 0  0 0 1 0 0 0 0 0 0 0 0 0  0 0 1 0 0 1 0 1 0 0 0 0  0 0 0 0 1 0 1 0 0 0 0 0  0 0 0 0 0 1 0 0 1 1 0 0  0 0 0 0 1 0 0 0 1 0 0 0  0 0 0 0 0 0 1 1 0 0 0 0  0 0 0 0 0 0 1 0 0 0 1 0  0 0 0 0 0 0 0 0 0 1 0 0  0 0 0 0 0 0 0 0 0 0 0 0 |  |
| **1.1** | 7 7 | Маршрут существует |
| **1.2** | 1 11 |
| **1.3** | 4 9 |
| **1.4** | 12 12 |
| **2** | 0 0 0 1 1 1 0 0 0 0 0 0  0 0 0 1 1 1 0 0 0 0 0 0  0 0 0 1 1 1 0 0 0 0 0 0  1 1 1 0 0 0 0 0 0 0 0 0  1 1 1 0 0 0 0 0 0 0 0 0  1 1 1 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 1 0 1 1 1  0 0 0 0 0 0 1 0 1 0 0 0  0 0 0 0 0 0 0 1 0 1 0 0  0 0 0 0 0 0 1 0 1 0 1 0  0 0 0 0 0 0 1 0 0 1 0 1  0 0 0 0 0 0 1 0 0 0 1 0 |  |
| **2.1** | 7 9 | Маршрут существует |
| **2.2** | 3 1 |
| **2.3** | 11 3 | Маршрут не существует |
| **2.4** | 1 10 |
| **3** | 1 0 0 0 0 0 0 0 0 0 0 0 0  0 0 0 0 1 1 0 0 1 0 0 0 1  0 0 1 0 0 0 0 1 0 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 1 0  0 1 0 0 0 0 0 0 1 0 0 0 1  0 1 0 0 0 0 0 0 1 0 0 0 0  0 0 0 0 0 0 0 0 0 0 0 1 0  0 0 1 0 0 0 0 1 0 0 1 0 0  0 1 0 0 1 1 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0 1 1 0 0  0 0 0 0 0 0 0 1 0 1 1 0 0  0 0 0 1 0 0 1 0 0 0 0 0 0  0 1 0 0 1 0 0 0 0 0 0 0 0 |  |
| **3.1** | 7 11 | Маршрут не существует |
| **3.2** | 1 5 |
| **3.3** | 1 1 | Маршрут существует |
| **3.4** | 13 6 |
| **3.5** | 3 10 |
| **4** | 0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 0  0 0 0 0 0 1 0 1 1  0 0 0 0 0 0 1 1 1  0 0 0 0 0 1 1 1 0  0 0 0 0 0 1 1 0 1 |  |
| **4.1** | 6 7 | Маршрут существует |
| **4.2** | 4 5 | Маршрут не существует |
| **4.3** | 1 2 |
| **4.4** | 3 1 |
| **4.5** | 3 8 |
| **4.6** | 4 6 |

1. **Тесты для оценки сложности**

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Входные данные** | **Время выполнения (секунды)** | |
|  | **Число вершин(n)** | **Поиск в глубину** | **Поиск в ширину** |
| **1** | n = 1000 | 0.07 | 0.04 |
| **2** | n = 2000 | 0.08 | 0.06 |
| **3** | n = 4000 | 0.15 | 0.13 |
| **4** | n = 8000 | 0.23 | 0.27 |

Вывод: при увеличении количества вершин вдвое наблюдается двойной увеличение времени работы алгоритмов. Это говорит о линейной зависимости времени выполнения от количества вершин.