1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский политехнический университет Петра Великого
3. —
4. Институт компьютерных наук и технологий
5. **Кафедра «Информационная безопасность компьютерных систем»**

**ОТЧЕТ**

**по лабораторной работе № 2**

1. по дисциплине «Дискретная Математика»
2. Выполнил
3. студент гр. 23508/4 Е.Г. Проценко
4. Проверила
5. ассистент Д.С. Лаврова
6. Санкт-Петербург
7. 2016
8. **Формулировка задания (Вариант 7)**

Цель работы – изучение алгоритмов обхода графов и алгоритмов поиска остова минимального веса.

1. Дополнить разработанный в лабораторной работе №1 класс следующими методами:

* определение висячих вершин, определение изолированных вершин;
* определение в орграфе истоков и стоков;
* определение расстояния между двумя вершинами;
* определение эксцентриситета вершины – максимального расстояния от нее до всех остальных вершин;
* определение диаметра графа – максимального эксцентриситета;
* определение радиуса графа – минимального эксцентриситета;
* определение центров графа – вершин с минимальным эксцентриситетом;
* определение периферийных вершин – вершин с = .

1. Реализовать в классе методы построения остова: поиск в ширину и поиск в глубину.
2. Реализовать алгоритм поиска остова минимального веса (Прима или Краскала в зависимости от варианта).

*Задание 1*. Для графа, полученного в последнем задании лабораторной работы №1, определить:

а) висячие и изолированные вершины;

б) радиус и диаметр графа;

в) центры и периферийные вершины.

Если в последнем задании лабораторной работы №1 граф построить не удалось, то подкорректировать исходные графы так, чтобы операции над ними можно было выполнить. Сделанные изменения указать в отчете.

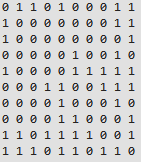
*Задание 2*. Для графа *G*4, полученного в лабораторной работе №1, построить остов. Четные варианты – методом обхода в ширину, нечетные варианты – методом обхода в глубину.

*Задание 3*. Найти остов минимального веса для взвешенного графа *G*5, заданного матрицей весов (выглядит как матрица смежности, только вместо 1 стоят веса; 0 – ребра нет). Вывести результат в виде матрицы смежности, указать общий вес полученного остова. Четные варианты решают задачу методом Прима, нечетные – методом Краскала.



1. **Ход работы**
   1. **Работа с графов, полученном из лабораторной работы 1**

Сам граф выглядит следующим образом:



Висячие и изолированные вершины отсутствуют:



Для нахождения радиуса, диаметра, центров, периферийных вершин был использован алгоритм Дейкстры поиска минимальных путей.

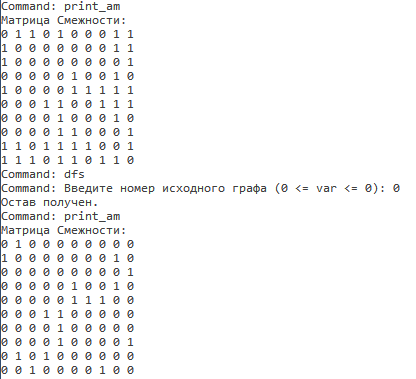
Радиус и диаметры:



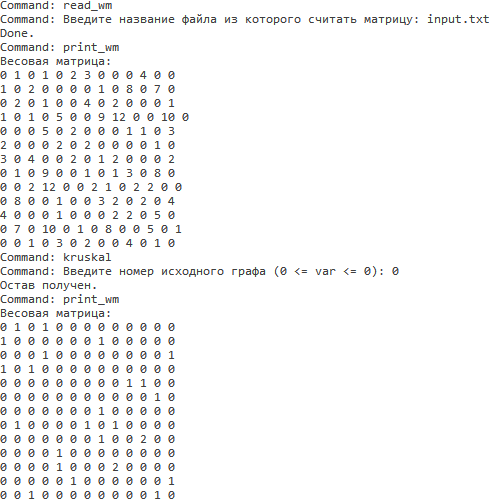
Центры и периферийные вершины:



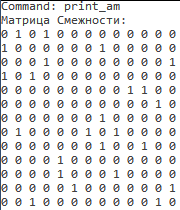
* 1. **Построение остава методом поиска в глубину**

****

* 1. **Нахождение остова минимального веса методом Крускала**

****

Результат в виде матрицы смежности:

****

1. **Контрольные вопросы**
   1. Как можно определить расстояние между двумя вершинами в невзвешенном графе?

* Обход в ширину
* Алгоритм Дейкстры
  1. Какие задачи можно решить методом поиска в глубину, а какие задачи – методом поиска в ширину?

В глубину: Поиск лексикографически первого пути в графе, проверка графа на ацикличность и нахождение цикла, поиск мостов.

В ширину: Поиск кратчайшего пути в невзвешенном графе; Поиск компонент связности в графе за O(n+m).

* 1. Нет, этот остов уникален. Докажем от противного: пусть существуют остовы минимального веса *T1* и *T2* (*T1 ≠ T2*). Пусть . Тогда при добавлении ребра *e* в *T2* получим цикл, следовательно, существует хотя бы одно ребро . Т.к. мы выбирали *e* такое, что его вес минимальный, то дерево имеет меньший вес, чем *T2*. Получили противоречие, следовательно, *T2* не может быть остовом минимального веса.

1. **Приложение**

**int \* Graph::get\_leaf\_vertices(int \* count)**

**{**

**int \* arr = NULL;**

**int cnt = 0;**

**for (int i = 0; i < vertices; i++)**

**{**

**int \* deg = (int \*)malloc(2 \* sizeof(int));**

**get\_v\_degree(i, deg);**

**if (deg[0] == 1){**

**arr = (int \*)realloc((void \*)arr, (\*count + 1) \* sizeof(int));**

**arr[cnt++] = i;**

**}**

**}**

**\*count = cnt;**

**return arr;**

**}**

**int \* Graph::get\_isolated\_vertices(int \* count)**

**{**

**int \* arr = NULL;**

**int cnt = 0;**

**for (int i = 0; i < vertices; i++)**

**{**

**int \* deg = (int \*)malloc(2 \* sizeof(int));**

**get\_v\_degree(i, deg);**

**if (deg[0] == 0){**

**arr = (int \*)realloc((void \*)arr, (\*count + 1) \* sizeof(int));**

**arr[cnt++] = i;**

**}**

**}**

**\*count = cnt;**

**return arr;**

**}**

**int \*\* Graph::get\_ecc\_matrix()**

**{**

**int \*\*ecc\_matrix = (int \*\*)malloc(vertices \* sizeof(int \*));**

**for (int i = 0; i < vertices; i++) ecc\_matrix[i] = (int \*)malloc(vertices \* sizeof(int));**

**for (int i = 0; i < vertices; i++)**

**{**

**int \* ecc = dijkstra(i);**

**for (int j = 0; j < vertices; j++) ecc\_matrix[i][j] = ecc[j];**

**}**

**return ecc\_matrix;**

**}**

**int Graph::minDistance(int \* dist, bool \* sptSet)**

**{**

**// Initialize min value**

**int min = INT\_MAX, min\_index;**

**for (int v = 0; v < vertices; v++)**

**if (sptSet[v] == false && dist[v] <= min)**

**min = dist[v], min\_index = v;**

**return min\_index;**

**}**

**int \* Graph::dijkstra(int src)**

**{**

**// Funtion that implements Dijkstra's single source shortest path algorithm**

**// for a graph represented using adjacency matrix representation**

**int v = vertices;**

**int \*\* graph = (int \*\*)malloc(vertices \* sizeof(int\*));**

**for (int i = 0; i < vertices; i++) graph[i] = (int \*)malloc(vertices \* sizeof(int));**

**for (int i = 0; i < v; i++) for (int j = 0; j < v; j++) graph[i][j] = (adjacency\_matrix[i][j] && i != j) ? 1 : 0;**

**int \* dist = (int \*)malloc(vertices \* sizeof(int)); // The output array. dist[i] will hold the shortest**

**// distance from src to i**

**bool \* sptSet = (bool \*)malloc(vertices \* sizeof(bool)); // sptSet[i] will true if vertex i is included in shortest**

**// path tree or shortest distance from src to i is finalized**

**// Initialize all distances as INFINITE and stpSet[] as false**

**for (int i = 0; i < vertices; i++)**

**dist[i] = INT\_MAX, sptSet[i] = false;**

**// Distance of source vertex from itself is always 0**

**dist[src] = 0;**

**// Find shortest path for all vertices**

**for (int count = 0; count < vertices - 1; count++)**

**{**

**// Pick the minimum distance vertex from the set of vertices not**

**// yet processed. u is always equal to src in first iteration.**

**int u = minDistance(dist, sptSet);**

**// Mark the picked vertex as processed**

**sptSet[u] = true;**

**// Update dist value of the adjacent vertices of the picked vertex.**

**for (int v = 0; v < vertices; v++)**

**// Update dist[v] only if is not in sptSet, there is an edge from**

**// u to v, and total weight of path from src to v through u is**

**// smaller than current value of dist[v]**

**if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX**

**&& dist[u] + graph[u][v] < dist[v])**

**dist[v] = dist[u] + graph[u][v];**

**}**

**return dist;**

**}**

**int \* Graph::get\_max\_ecc\_for\_every\_vortex(int \*\* ecc\_matrix)**

**{**

**int \* eccs = (int \*)calloc(vertices , sizeof(int));**

**for (int i = 0; i < vertices; i++)**

**{**

**for (int j = 0; j < vertices; j++)**

**{**

**if (ecc\_matrix[i][j] > eccs[i]) eccs[i] = ecc\_matrix[i][j];**

**}**

**}**

**return eccs;**

**}**

**int Graph::get\_radius()**

**{**

**int \*\* ecc\_matrix = get\_ecc\_matrix();**

**int \* eccs = get\_max\_ecc\_for\_every\_vortex(ecc\_matrix);**

**int rad = INT\_MAX;**

**for (int i = 1; i < vertices; i++) if (rad > eccs[i] && i > 0) rad = eccs[i];**

**return rad;**

**}**

**int Graph::get\_diameter()**

**{**

**int \*\* ecc\_matrix = get\_ecc\_matrix();**

**int \* eccs = get\_max\_ecc\_for\_every\_vortex(ecc\_matrix);**

**int dia = 0;**

**for (int i = 1; i < vertices; i++) if (dia < eccs[i]) dia = eccs[i];**

**return dia;**

**}**

**int \* Graph::get\_vertices\_by\_ecc(int \*count, int ecc)**

**{**

**int \*\* ecc\_matrix = get\_ecc\_matrix();**

**int \* eccs = get\_max\_ecc\_for\_every\_vortex(ecc\_matrix);**

**int cnt = 0;**

**int \* arr = NULL;**

**for (int i = 0; i < vertices; i++)**

**{**

**if (eccs[i] == ecc)**

**{**

**arr = (int \*)realloc((void \*)arr, (cnt + 1) \* sizeof(int));**

**arr[cnt++] = i;**

**}**

**}**

**\*count = cnt;**

**return arr;**

**}**

**int \* Graph::get\_centers(int \* count)**

**{**

**int rad = get\_radius();**

**return get\_vertices\_by\_ecc(count, rad);**

**}**

**int \* Graph::get\_peripherals(int \* count)**

**{**

**int dia = get\_diameter();**

**return get\_vertices\_by\_ecc(count, dia);**

**}**

**void Graph::dfs(Graph \* G)**

**{**

**vertices = G->vertices;**

**int \*\* spanning\_tree = (int \*\*)malloc(vertices \* sizeof(int \*));**

**for (int i = 0; i < vertices; i++) spanning\_tree[i] = (int \*)calloc(vertices, sizeof(int));**

**bool \* broed = (bool \*)malloc(vertices \* sizeof(bool));**

**bool \* visited = (bool \*)malloc(vertices \* sizeof(bool));**

**for (int i = 0; i < vertices; i++) broed[i] = visited[i] = false;**

**for (int i = 0; i < vertices; i++) if (visited[i] == false) dfs\_works(i, broed, visited, spanning\_tree, G);**

**reinit\_adj\_matrix();**

**for (int i = 0; i < vertices; i++) for (int j = 0; j < vertices; j++) adjacency\_matrix[i][j] = spanning\_tree[i][j];**

**converting\_from\_am();**

**}**

**void Graph::dfs\_works(int work, bool \* broed, bool \* visited, int \*\* spanning\_tree, Graph \* G)**

**{**

**visited[work] = true;**

**for (int i = 0; i < vertices; i++)**

**{**

**if (G->adjacency\_matrix[work][i] && visited[i] == false)**

**{**

**spanning\_tree[work][i] = spanning\_tree[i][work] = 1;**

**//broed[work] = broed[i] = true;**

**dfs\_works(i, broed, visited, spanning\_tree, G);**

**}**

**}**

**}**

**void Graph::read\_weight\_matrix(std::string \* str)**

**{**

**clear\_all();**

**vertices = calculate\_vertices(str);**

**init\_weight\_matrix();**

**is\_weight\_graph = true;**

**std::ifstream fin(str->c\_str());**

**for (int i = 0; i < vertices; i++)**

**{**

**for (int j = 0; j < vertices; j++)**

**{**

**fin >> weight\_matrix[i][j];**

**if (i > j && (weight\_matrix[i][j] != weight\_matrix[j][i])) is\_orgraph = true;**

**}**

**}**

**fin.close();**

**init\_adj\_matrix();**

**clear\_adj\_matrix();**

**convert\_weight\_into\_adj();**

**calculate\_edges();**

**init\_inc\_matrix();**

**convert\_adj\_into\_inc();**

**init\_adj\_list();**

**fill\_adjacency\_list();**

**}**

**void Graph::init\_weight\_matrix()**

**{**

**weight\_matrix = (int \*\*)malloc(vertices \* sizeof(int \*));**

**for (int i = 0; i < vertices; i++) weight\_matrix[i] = (int \*)malloc(vertices \* sizeof(int));**

**wm\_v = vertices;**

**}**

**void Graph::convert\_weight\_into\_adj()**

**{**

**for (int i = 0; i < vertices; i++)**

**{**

**for (int j = 0; j < vertices; j++)**

**{**

**if (weight\_matrix[i][j]) adjacency\_matrix[i][j] = 1;**

**}**

**}**

**}**

**void Graph::print\_weight\_matrix()**

**{**

**for (int i = 0; i < vertices; i++)**

**{**

**//printf("%d\t|", names[i]);**

**for (int j = 0; j < vertices; j++)**

**{**

**printf("%d ", weight\_matrix[i][j]);**

**}**

**printf("\n");**

**}**

**}**

**void Graph::sort\_edges(struct edge \* edges, int size)**

**{**

**int i, j;**

**int \_v1, \_v2, \_weight;**

**for (i = 1; i < size; i++)**

**{**

**\_v1 = edges[i].v1;**

**\_v2 = edges[i].v2;**

**\_weight = edges[i].weight;**

**for (j = i - 1; j >= 0; j--)**

**{**

**if (edges[j].weight < \_weight)**

**break;**

**edges[j + 1].v1 = edges[j].v1;**

**edges[j].v1 = \_v1;**

**edges[j + 1].v2 = edges[j].v2;**

**edges[j].v2 = \_v2;**

**edges[j + 1].weight = edges[j].weight;**

**edges[j].weight = \_weight;**

**}**

**}**

**}**

**int Graph::get\_comp\_numb(int work, int \*\* connectivity, int total\_comps, int \* items\_in\_comp)**

**{**

**for (int i = 0; i < total\_comps; i++)**

**{**

**for (int j = 0; j < items\_in\_comp[i]; j++)**

**{**

**if (connectivity[i][j] == work) return i;**

**}**

**}**

**return -1;**

**}**

**void Graph::kruskal(Graph \* G)**

**{**

**vertices = G->vertices;**

**int \*\* min\_weight\_matrix = (int \*\*)malloc(vertices \* sizeof(int \*));**

**for (int i = 0; i < vertices; i++) min\_weight\_matrix[i] = (int \*)calloc(vertices, sizeof(int));**

**struct edge \* edges = NULL;**

**int count = 0;**

**for (int i = 0; i < vertices; i++)**

**{**

**for (int j = i + 1; j < vertices; j++)**

**{**

**int \_weight = G->weight\_matrix[i][j];**

**if (\_weight)**

**{**

**edges = (struct edge \*)realloc((void \*)edges, (count + 1) \* sizeof(struct edge));**

**edges[count].v1 = i;**

**edges[count].v2 = j;**

**edges[count].weight = \_weight;**

**count++;**

**}**

**}**

**}**

**sort\_edges(edges, count);**

**int \*\* сonnectivity = NULL; // компоненты связности**

**int total\_comps = 0;**

**int \* items\_in\_comp = NULL;**

**for (int i = 0; i < count; i++) // 'i' is for 'edges'**

**{**

**int \_v1 = edges[i].v1;**

**int \_v2 = edges[i].v2;**

**int \_w = edges[i].weight;**

**int camp\_1 = get\_comp\_numb(edges[i].v1, сonnectivity, total\_comps, items\_in\_comp);**

**int camp\_2 = get\_comp\_numb(edges[i].v2, сonnectivity, total\_comps, items\_in\_comp);**

**if (camp\_1 == camp\_2)**

**{**

**if (camp\_1 == -1)**

**{**

**сonnectivity = (int \*\*)realloc((void \*)сonnectivity, (total\_comps + 1) \* sizeof(int \*));**

**items\_in\_comp = (int \*)realloc((void \*)items\_in\_comp, (total\_comps + 1) \* sizeof(int));**

**items\_in\_comp[total\_comps] = 2;**

**сonnectivity[total\_comps] = (int \*)malloc(2 \* sizeof(int));**

**сonnectivity[total\_comps][0] = \_v1;**

**сonnectivity[total\_comps][1] = \_v2;**

**total\_comps++;**

**min\_weight\_matrix[\_v1][\_v2] = min\_weight\_matrix[\_v2][\_v1] = \_w;**

**}**

**else continue;**

**}**

**else if (camp\_1 != -1 && camp\_2 != -1)**

**{**

**for (int k = 0; k < items\_in\_comp[camp\_1]; k++)**

**{**

**сonnectivity[camp\_2] = (int \*)realloc((void \*)сonnectivity[camp\_2], (items\_in\_comp[camp\_2] + 1) \* sizeof(int));**

**сonnectivity[camp\_2][items\_in\_comp[camp\_2]++] = сonnectivity[camp\_1][k];**

**}**

**free(сonnectivity[camp\_1]);**

**сonnectivity[camp\_1] = сonnectivity[--total\_comps];**

**items\_in\_comp[camp\_1] = items\_in\_comp[total\_comps];**

**сonnectivity = (int \*\*)realloc((void \*)сonnectivity, (total\_comps) \* sizeof(int\*));**

**items\_in\_comp = (int \*)realloc((void \*)items\_in\_comp, (total\_comps) \* sizeof(int));**

**min\_weight\_matrix[\_v1][\_v2] = min\_weight\_matrix[\_v2][\_v1] = \_w;**

**}**

**else if (camp\_1 == -1)**

**{**

**сonnectivity[camp\_2] = (int \*)realloc((void \*)сonnectivity[camp\_2], (items\_in\_comp[camp\_2] + 1) \* sizeof(int));**

**сonnectivity[camp\_2][items\_in\_comp[camp\_2]++] = \_v1;**

**min\_weight\_matrix[\_v1][\_v2] = min\_weight\_matrix[\_v2][\_v1] = \_w;**

**}**

**else**

**{**

**сonnectivity[camp\_1] = (int \*)realloc((void \*)сonnectivity[camp\_1], (items\_in\_comp[camp\_1] + 1) \* sizeof(int));**

**сonnectivity[camp\_1][items\_in\_comp[camp\_1]++] = \_v2;**

**min\_weight\_matrix[\_v1][\_v2] = min\_weight\_matrix[\_v2][\_v1] = \_w;**

**}**

**if (items\_in\_comp[0] == vertices) break;**

**}**

**for (int i = 0; i < vertices; i++)**

**{**

**for (int j = 0; j < vertices; j++)**

**{**

**weight\_matrix[i][j] = min\_weight\_matrix[i][j];**

**}**

**}**

**converting\_from\_wm();**

**}**

**void Graph::converting\_from\_wm()**

**{**

**reinit\_adj\_matrix();**

**clear\_adj\_matrix();**

**convert\_weight\_into\_adj();**

**calculate\_edges();**

**reinit\_inc\_matrix();**

**convert\_adj\_into\_inc();**

**//print\_incidence\_matrix();**

**reinit\_adj\_list();**

**fill\_adjacency\_list();**

**}**