Министерство образования и науки Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
—
Институт компьютерных наук и технологий
Кафедра «Информационная безопасность компьютерных систем»

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2

по дисциплине «Дискретная Математика»

Выполнил
студент гр. 23508/4 Е.Г. Проценко

Проверила
ассистент Д.С. Лаврова

Санкт-Петербург
2016

1. 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 ε вершины – максимального расстояния от нее до всех остальных вершин;
   - определение диаметра δ графа – максимального эксцентриситета;
   - определение радиуса графа – минимального эксцентриситета;
   - определение центров графа – вершин с минимальным эксцентриситетом;
   - определение периферийных вершин – вершин с ε = δ.
2. Реализовать в классе методы построения остова: поиск в ширину и поиск в глубину.
3. 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.

$$G_5 = \begin{pmatrix} 0 & 1 & 0 & -1 & 0 & 2 & 3 & 0 & 0 & 0 & 4 & 0 & 0 \\ 1 & 0 & 2 & 0 & 0 & 0 & 0 & 1 & 0 & 8 & 0 & 7 & 0 \\ 0 & 2 & 0 & 1 & 0 & 0 & 4 & 0 & 2 & 0 & 0 & 0 & 1 \\ -1 & 0 & 1 & 0 & 5 & 0 & 0 & 9 & 12 & 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 5 & 0 & 2 & 0 & 0 & 0 & -1 & 1 & 0 & 3 \\ 2 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 4 & 0 & 0 & 2 & 0 & -1 & 2 & 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 9 & 0 & 0 & -1 & 0 & 1 & 3 & 0 & 8 & 0 \\ 0 & 0 & 2 & 12 & 0 & 0 & 2 & 1 & 0 & 2 & 2 & 0 & 0 \\ 0 & 8 & 0 & 0 & -1 & 0 & 0 & 3 & 2 & 0 & 2 & 0 & 4 \\ 4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 2 & 2 & 0 & 5 & 0 \\ 0 & 7 & 0 & 10 & 0 & 1 & 0 & 8 & 0 & 0 & 5 & 0 & 1 \\ 0 & 0 & 1 & 0 & 3 & 0 & 2 & 0 & 0 & 4 & 0 & 1 & 0 \end{pmatrix}$$

2. Ход работы

2.1. Работа с графов, полученном из лабораторной работы 1

Сам граф выглядит следующим образом:

```
0 1 1 0 1 0 0 0 1 1
1 0 0 0 0 0 0 0 1 1
1 0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0 1 0
1 0 0 0 0 1 1 1 1 1
0 0 0 1 1 0 0 1 1 1
0 0 0 0 1 0 0 0 1 0
0 0 0 0 1 1 0 0 0 1
1 1 0 1 1 1 1 0 0 1
1 1 1 0 1 1 0 1 1 0
```

Висячие и изолированные вершины отсутствуют:

```
Command: get_leaf
Leaf Verticies:
Command: get_isolated
Isolated Verticies:
```

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

Радиус и диаметры:

```
Command: get_radius
Radius: 2
Command: get_diameter
Diameter: 3
```

Центры и периферийные вершины:

```
Command: get_centers
Centers: 0 1 4 5 7 8 9
Command: get_peripherals
Peripheral Verticies: 2 3 6
```

2.2. Построение остава методом поиска в глубину

```
Command: print_am
Матрица Смежности:
0 1 1 0 1 0 0 0 1 1
1 0 0 0 0 0 0 0 1 1
1 0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0 1 0
1 0 0 0 0 1 1 1 1 1
0 0 0 1 1 0 0 1 1 1
0 0 0 0 1 0 0 0 1 0
0 0 0 0 1 1 0 0 0 1
1 1 0 1 1 1 1 0 0 1
1 1 1 0 1 1 0 1 1 0
Command: dfs
Command: Введите номер исходного графа (0 <= var <= 0): 0
Остав получен.
Command: print_am
Матрица Смежности:
0 1 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0 1 0
0 0 0 0 0 1 1 1 0 0
0 0 0 1 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0 1
0 1 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 1 0 0
```

2.3. Нахождение остова минимального веса методом Крускала

```
Command: read_wm
Command: Введите название файла из которого считать матрицу: input.txt
Done.
Command: print_wm
Весовая матрица:
0 1 0 1 0 2 3 0 0 0 4 0 0
1 0 2 0 0 0 0 1 0 8 0 7 0
0 2 0 1 0 0 4 0 2 0 0 0 1
1 0 1 0 5 0 0 9 12 0 0 10 0
0 0 0 5 0 2 0 0 0 1 1 0 3
2 0 0 0 2 0 2 0 0 0 0 1 0
3 0 4 0 0 2 0 1 2 0 0 0 2
0 1 0 9 0 0 1 0 1 3 0 8 0
0 0 2 12 0 0 2 1 0 2 2 0 0
0 8 0 0 1 0 0 3 2 0 2 0 4
4 0 0 0 1 0 0 0 2 2 0 5 0
0 7 0 10 0 1 0 8 0 0 5 0 1
0 0 1 0 3 0 2 0 0 4 0 1 0
Command: kruskal
Command: Введите номер исходного графа (0 <= var <= 0): 0
Остав получен.
Command: print_wm
Весовая матрица:
0 1 0 1 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 1
1 0 1 0 0 0 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 0 0 0 0 1 0
0 0 0 0 0 0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 1 0 1 0 0 0
0 0 0 0 0 0 0 0 1 0 0 2 0
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 2 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 0 0 0 1 0
```

Результат в виде матрицы смежности:

```
Command: print_am
Матрица Смежности:
0 1 0 1 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 1
1 0 1 0 0 0 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 0 0 0 0 1 0
0 0 0 0 0 0 0 1 0 0 0 0 0
0 1 0 0 0 0 1 0 1 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 0 0 0 0 0
0 0 0 0 1 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 0 0 0 1 0
```

3. Контрольные вопросы

3.1. Как можно определить расстояние между двумя вершинами в невзвешенном графе?

- Обход в ширину
- Алгоритм Дейкстры

3.2. 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).

3.3. Нет, этот остов уникален. Докажем от противного: пусть существуют остовы минимального веса T_1 и T_2 ($T_1 \neq T_2$). Пусть $e \in T_1$ и $e \overline{\in} T_2 : c(e) = min$. Тогда при добавлении ребра e в T_2 получим цикл, следовательно, существует хотя бы одно ребро $f \in T_2 : f \overline{\in} T_1$. Т.к. мы выбирали e такое, что его вес минимальный, то дерево $T_2 \cup \{e\} \setminus \{f\}$ имеет меньший вес, чем T_2. Получили противоречие, следовательно, T_2 не может быть остовом минимального веса.

4. 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 ** connectivity = 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,  connectivity,  total_comps,
items_in_comp);
		int  camp_2  =  get_comp_numb(edges[i].v2,  connectivity,  total_comps,
items_in_comp);

		if (camp_1 == camp_2)
		{
			if (camp_1 == -1)
			{
				connectivity  =  (int  **)realloc((void  *)connectivity,
(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;
				connectivity[total_comps]    =    (int    *)malloc(2    *
sizeof(int));
				connectivity[total_comps][0] = _v1;
				connectivity[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++)
			{
				connectivity[camp_2]      =      (int      *)realloc((void
*)connectivity[camp_2], (items_in_comp[camp_2] + 1) * sizeof(int));
				connectivity[camp_2][items_in_comp[camp_2]++]            =
connectivity[camp_1][k];
			}
			free(connectivity[camp_1]);
			connectivity[camp_1] = connectivity[--total_comps];
```

```
                    items_in_comp[camp_1] = items_in_comp[total_comps];
                    connectivity    =    (int    **)realloc((void    *)connectivity,
(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)
             {
                    connectivity[camp_2]        =        (int        *)realloc((void
*)connectivity[camp_2], (items_in_comp[camp_2] + 1) * sizeof(int));
                    connectivity[camp_2][items_in_comp[camp_2]++] = _v1;

                    min_weight_matrix[_v1][_v2] = min_weight_matrix[_v2][_v1] = _w;
             }

             else
             {
                    connectivity[camp_1]        =        (int        *)realloc((void
*)connectivity[camp_1], (items_in_comp[camp_1] + 1) * sizeof(int));
                    connectivity[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();
}
```