Домашнє завдання №17

Скласти програму (C/C++), яка дозволяє знаходити мінімальне кістякове дерево*(англ. minimum spanning tree)* для заданого графу за допомогою алгоритму Крускала.

Вибір варіанту

(Nж + Nг + 1) % 2 + 1
де: Nж – порядковий номер студента в групі, а Nг – номер групи(1,2,3,4,5,6,7,8 або 9)

Варіанти завдань

Варіант	Кількість вершин графу
1	4
2	5

Приклад коду

Програма відображає заданий граф та мінімальне кістякове дерево(отримане після виконання алгоритму Крускала) у вигляді матриць суміжності(*англ.* adjacency matrix), в яких замість чисел 0 і 1(відсутність або присутність ребра), містяться ваги ребер(на відсутність ребра вказує значення NE – not exist).

Кількість вершин графу у прикладі	7
Макровизначення	`#define VERTEX_COUNT 7`

Лістинг

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define VERTEX_COUNT 7
#define MAX_VERTEX_COUNT 8

#define UNDIRECT_BEHAVIOR

#define NE (~0) // NOT EXIST
#define NA NE   // NOT AVAILABLE

#define EDGE_VALUES {\
/******V0**V1**V2**V3**V4**V5**V6**V7*/\
/*V0*/{NA, 7,  NE, 5,  NE, NE, NE, NE},\
/*V1*/{NA, NA, 8,  9,  7,  NE, NE, NE},\
/*V2*/{NA, NA, NA, NE, 5,  NE, NE, NE},\
/*V3*/{NA, NA, NA, NA, 15, 6,  NE, NE},\
/*V4*/{NA, NA, NA, NA, NA, 8,  9,  NE},\
/*V5*/{NA, NA, NA, NA, NA, NA, 11, NE},\
/*V6*/{NA, NA, NA, NA, NA, NA, NA, NE},\
/*V7*/{NA, NA, NA, NA, NA, NA, NA, NA}\
}
```

```
typedef struct EdgeStruct {
	unsigned int sourceVertexIndex;
	unsigned int destinationVertexIndex;
	int weight;
} Edge;

typedef struct EdgesStruct {
	unsigned int vertexCount;
	unsigned int edgeCount;
	unsigned int reservedEdgeCount;
	Edge* items;
} Edges;

Edges* createEdges(int edgeValues[MAX_VERTEX_COUNT][MAX_VERTEX_COUNT], int vertexCount) {
	int edgeIndex, iIndex, jIndex;

	Edges* edges = (Edges*)malloc(sizeof(Edges));
	edges->vertexCount = vertexCount;

	edges->edgeCount = 0;
	for (iIndex = 0; iIndex < vertexCount; ++iIndex) {
#ifdef UNDIRECT_BEHAVIOR
		jIndex = iIndex + 1;
#else
		jIndex = 0;
#endif
		for (jIndex = 0; jIndex < vertexCount; ++jIndex) {
			if (edgeValues[iIndex][jIndex] != NE) {
				++edges->edgeCount;
			}
		}
	}
	edges->reservedEdgeCount = edges->edgeCount;

	edges->items = (Edge*)malloc(edges->edgeCount * sizeof(Edge));

	edgeIndex = 0;
	for (iIndex = 0; iIndex < vertexCount; ++iIndex) {
#ifdef UNDIRECT_BEHAVIOR
		jIndex = iIndex + 1;
#else
		jIndex = 0;
#endif
		for (; jIndex < vertexCount; ++jIndex) {
			if (edgeValues[iIndex][jIndex] != NE) {
				edges->items[edgeIndex].sourceVertexIndex = iIndex;
				edges->items[edgeIndex].destinationVertexIndex = jIndex;
				edges->items[edgeIndex].weight = edgeValues[iIndex][jIndex];
				++edgeIndex;
			}
		}
	}

	return edges;
}

void destroyEdges(Edges* edges) {
	if (edges) {
		free(edges->items);
		free(edges);
	}
}

typedef struct SubsetStruct {
```

```
	unsigned int parent;
	unsigned int rank;
} Subset;

unsigned int find(Subset * subsets, unsigned int index) {
	if (subsets[index].parent != index) {
		subsets[index].parent = find(subsets, subsets[index].parent);
	}

	return subsets[index].parent;
}

void mergeSubsets(Subset * subsets, unsigned int first, unsigned int second) {
	unsigned int firstRoot = find(subsets, first);
	unsigned int secondRoot = find(subsets, second);

	if (subsets[firstRoot].rank < subsets[secondRoot].rank) {
		subsets[firstRoot].parent = secondRoot;
	}
	else if (subsets[firstRoot].rank > subsets[secondRoot].rank) {
		subsets[secondRoot].parent = firstRoot;
	}
	else {
		subsets[secondRoot].parent = firstRoot;
		++subsets[firstRoot].rank;
	}
}

int edgeCompare(const void* a, const void* b){
	return ((Edge*)a)->weight > ((Edge*)b)->weight;
}

Edges* KruskalMST(Edges* edges) {
	unsigned int resultEdgeIndex;
	unsigned int edgeIndex;
	unsigned int first;
	unsigned int second;
	unsigned int vertexIndex;
	Subset* subsets;
	Edge next_edge;
	Edges* result;

	if (!edges) {
		return NULL;
	}

	result = (Edges*)malloc(sizeof(Edges));
	if (!result) {
		return NULL;
	}
	result->vertexCount = edges->vertexCount;
	result->reservedEdgeCount = edges->vertexCount;
	result->items = (Edge*)malloc(edges->vertexCount * sizeof(Edge));

	qsort(edges->items, edges->edgeCount, sizeof(Edge), edgeCompare);

	subsets = (Subset*)malloc(edges->vertexCount * sizeof(Subset));

	for (vertexIndex = 0; vertexIndex < edges->vertexCount; ++vertexIndex){
		subsets[vertexIndex].parent = vertexIndex;
		subsets[vertexIndex].rank = 0;
	}

	for (resultEdgeIndex = 0, edgeIndex = 0; resultEdgeIndex + 1 < edges->vertexCount
&& edgeIndex < edges->edgeCount;) {
```

```
		next_edge = edges->items[edgeIndex++];

		first = find(subsets, next_edge.sourceVertexIndex);
		second = find(subsets, next_edge.destinationVertexIndex);

		if (first != second){
			result->items[resultEdgeIndex++] = next_edge;
			mergeSubsets(subsets, first, second);
		}
	}

	result->edgeCount = resultEdgeIndex;

	free(subsets);

	return result;
}

void printGraphEdge(const char* title, Edges* edges) {
	int printEdgeValues[MAX_VERTEX_COUNT][MAX_VERTEX_COUNT] = { 0 };

	unsigned int edgeIndex, iIndex, jIndex;

	if (!edges || !edges->items) {
		return;
	}

	for (iIndex = 0; iIndex < edges->vertexCount; ++iIndex) {
		for (jIndex = 0; jIndex < edges->vertexCount; ++jIndex) {
			printEdgeValues[iIndex][jIndex] = NE;
		}
	}

	for (edgeIndex = 0; edgeIndex < edges->edgeCount; ++edgeIndex) {
		printEdgeValues[edges->items[edgeIndex].sourceVertexIndex %
MAX_VERTEX_COUNT][edges->items[edgeIndex].destinationVertexIndex % MAX_VERTEX_COUNT] =
edges->items[edgeIndex].weight;
	}

	printf("%s\r\n   ", title);
	for (jIndex = 0; jIndex < edges->vertexCount; ++jIndex) {
		printf(" V%-2d", jIndex);
	}
	printf("\r\n");
	for (iIndex = 0; iIndex < edges->vertexCount; ++iIndex) {
		printf("V%-2d", iIndex);
		for (jIndex = 0; jIndex < edges->vertexCount; ++jIndex) {
			if (jIndex) {
				printf(",");
			}
			printf(" ");
#ifdef UNDIRECT_BEHAVIOR
			if (iIndex < jIndex) {
#endif
				if (printEdgeValues[iIndex][jIndex] != NE) {
					printf("%-2d", printEdgeValues[iIndex][jIndex]);
				}
				else {
					printf("NE");
				}
#ifdef UNDIRECT_BEHAVIOR
			}
			else {
				printf("NA");
			}
```

```
#endif
			}
			printf("\r\n");
		}
		printf("\r\n");
}

int main() {
		int edgeValues[MAX_VERTEX_COUNT][MAX_VERTEX_COUNT] = EDGE_VALUES;
		Edges* edges = createEdges(edgeValues, VERTEX_COUNT), *spanningTreeEdges;
		if (!edges) {
			return 1;
		}

		spanningTreeEdges = KruskalMST(edges);
		if (!spanningTreeEdges) {
			destroyEdges(edges);
			return 1;
		}

		printGraphEdge("Graph:", edges);

		printGraphEdge("Spanning tree:", spanningTreeEdges);

		destroyEdges(edges);
		destroyEdges(spanningTreeEdges);

#ifdef __linux__
		(void)getchar();
#elif defined(_WIN32)
		system("pause");
#else
#endif

		return 0;
}
```