Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

"Белгородский государственный технологический университет им. В. Г. Шухова"

(БГТУ им. В.Г. Шухова)

Институт энергетики, информационных технологий и управляющих систем

Кафедра программного обеспечения вычислительной техники

и автоматизированных систем

**Лабораторная работа № 4.3**

**по дисциплине дискретная математика**

**тема: Связность**

**Выполнил: студент группы ПВ-223**

**Игнатьев Артур Олегович**

**Проверил: доцент   
Рязанов Юрий Дмитриевич**

Белгород 2023

**Цель занятия:** изучить алгоритм Краскала построения покрывающего леса, научиться использовать его при решении различных задач.

**Задания**

1. Реализовать алгоритм Краскала построения покрывающего леса.

2. Используя алгоритм Краскала, разработать и реализовать алгоритм решения задачи.

3. Подобрать тестовые данные. Результат представить в виде диаграммы графа.

**Вариант №9**

Найти максимальное множество ребер, исключение которых из связного графа разбивает его на две связные компоненты.

**Выполнение**

#include <stdio.h>  
#include <stdlib.h>  
#include <malloc.h>  
#include <ctype.h>  
  
//вывод массива a размера n  
void display\_array(int \*a, int n) {  
 for (int i = 0; i < n; i++)  
 printf("%d ", a[i]+1);  
 printf("\n");  
}  
  
void display\_matrix(int \*\*a, int n) {  
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < n; j++)  
 printf("%d ", a[i][j]);  
 printf("\n");  
 }  
 printf("\n");  
}  
  
int \*\*init\_matrix(int n){  
 int \*\*matrix = (int\*\*)calloc(n, sizeof(int\*));//создаем тестовую матрицу  
 for (int i = 0; i < n; i++)  
 matrix[i] = (int\*)calloc(n, sizeof(int));  
 return matrix;  
}  
  
//алгоритм Краскала  
int\*\* kraskal\_algorithm(int \*\*graph, int n) {  
 int \*flower = (int\*)malloc(sizeof(int)\*n);  
 int \*\*forest = init\_matrix(n);  
 for (int j = 0; j < n; j++)  
 flower[j] = j;  
 for (int i = 0; i < n; i++)  
 for (int j = i; j < n; j++)  
 if (graph[i][j] == 1 )  
 if (flower[i] != flower[j]) {  
 forest[i][j] = 1;  
 forest[j][i] = 1;  
 for (int k = 0; k < n; k++)  
 if(flower[k] == flower[j])  
 flower[k] = flower[i];  
 }  
 printf("Flowers: ");  
 display\_array(flower, n);  
 printf("Forest:\n");  
 display\_matrix(forest, n);  
 free(flower);  
 return forest;  
}  
  
//обнуление массива a размера n  
void zeros\_array(int \*a, int n) {  
 for (int i = 0; i < n; i++)  
 a[i] = 0;  
}  
  
//освобождение памяти  
void free\_matrix(int\*\* matrix, int n){  
 for (int i = 0; i < n; i++)  
 free(matrix[i]);  
 free(matrix);  
}  
  
int min(int a, int b){  
 if (a < b) return a;  
 else return b;  
}  
  
// Массив visited используется для отмечания уже пройденных вершин.  
// Для решения задачи будем также использовать два дополнительных массива timer\_in и timer\_up, храняшие глубину спуска.  
// Чтобы определить, является ли прямое ребро v→to мостом, мы можем воспользоваться следующим критерием:  
// глубина timer\_in[v] вершины v меньше, чем минимальная глубина всех вершин,  
// соединенных обратным ребром с какой-либо вершиной из поддерева to.  
// Если условие (timer\_up[v] < timer\_in[to]) не выполняется, то существует какой-то путь из to в какого-то предка v или саму v,  
// не использующий ребро (v,to), а в противном случае — наоборот.  
void recursion\_part\_for\_finding\_bridges(int \*\*graph, int \*\*bridges, int n, int \*visited, int \*timer\_in, int \*timer\_up, int v, int p){  
 static int timer;  
 visited[v] = 1;  
 timer\_in[v] = timer\_up[v] = timer++;  
 for (int to = 0; to < n; to++){  
 if(graph[v][to] == 1){  
 if (to == p)  
 continue;  
 if (visited[to])  
 timer\_up[v] = min(timer\_up[v], timer\_in[to]);  
 else{  
 recursion\_part\_for\_finding\_bridges(graph, bridges, n, visited, timer\_in, timer\_up, to, v);  
 timer\_up[v] = min(timer\_up[v], timer\_in[to]);  
 if (timer\_up[to] > timer\_in[v]) {  
 bridges[v][to] = 1;  
 bridges[to][v] = 1;  
 printf("Bridge: (%d, %d)\n", v + 1, to + 1);  
 }  
 }  
 }  
 }  
}  
  
int\*\* taryans\_algorithm\_for\_finding\_bridges(int \*\*graph, int n){  
 int timer\_in[n], timer\_up[n];  
 int visited[n];  
 int \*\*bridges = init\_matrix(n);  
 for(int i = 0; i < n; ++i)  
 visited[i] = 0;  
 for(int i = 0; i < n; ++i)  
 if(!visited[i])  
 recursion\_part\_for\_finding\_bridges(graph, bridges, n, visited, timer\_in, timer\_up, i, -1);  
 return bridges;  
}  
  
// Обход в глубину  
void dfs(int\*\* graph, int\* visited, int v, int n){  
 if(visited[v])  
 return;  
 visited[v] = 1;  
 for(int i = 0; i < n; i++)  
 if(graph[v][i])  
 dfs(graph, visited, i, n);  
}  
  
// Является ли граф связным  
int is\_graph\_connect(int \*\* graph, int n) {  
 int \*visited = (int\*)malloc(sizeof(int)\*n);  
 zeros\_array(visited, n);  
 dfs(graph, visited, 0, n);  
 for(int i = 0; i < n; i++)  
 if(!visited[i]) {  
 free(visited);  
 return 0;  
 }  
 free(visited);  
 return 1;  
}  
  
int \*\*copy\_matrix(int\*\* matrix, int n){  
 int \*\*new\_matrix = init\_matrix(n);  
 for (int i = 0; i < n; i++)  
 for (int j = 0; j < n; j++)  
 new\_matrix[i][j] = matrix[i][j];  
 return new\_matrix;  
}  
  
// По одному удаляет ребра deleted\_edges из графа, до тех пор, пока тот не перестанет быть связным  
// возвращает граф с двумя связными компонентами  
int\*\* split\_graph\_to\_two\_connected\_components(int \*\*graph, int \*\*deleted\_edges, int n){  
 int \*\*forest\_with\_two\_components = copy\_matrix(graph, n);  
 for (int i = 0; i < n; i++)  
 for (int j = 0; j < n; j++)  
 if(graph[i][j] == 1 && deleted\_edges[i][j] == 1){  
 forest\_with\_two\_components[i][j] = 0;  
 forest\_with\_two\_components[j][i] = 0;  
 if(!is\_graph\_connect(forest\_with\_two\_components, n))  
 return forest\_with\_two\_components;  
 }  
 return forest\_with\_two\_components;  
}  
  
void show\_graph\_difference(int\*\* graph\_1, int\*\* graph\_2, int n){  
 for (int i = 0; i < n; i++)  
 for (int j = i; j < n; j++)  
 if(graph\_1[i][j] != graph\_2[i][j])  
 printf("Pair (%d, %d)\n", i+1, j+1);  
}  
  
int main() {  
 int n = 6;  
 int \*\*graph = init\_matrix(n);  
// input\_matrix(graph, n);  
 graph[0][0] = 0; graph[0][1] = 1; graph[0][2] = 0; graph[0][3] = 0; graph[0][4] = 1; graph[0][5] = 0;  
 graph[1][0] = 1; graph[1][1] = 0; graph[1][2] = 1; graph[1][3] = 1; graph[1][4] = 0; graph[1][5] = 1;  
 graph[2][0] = 0; graph[2][1] = 1; graph[2][2] = 0; graph[2][3] = 0; graph[2][4] = 1; graph[2][5] = 0;  
 graph[3][0] = 0; graph[3][1] = 1; graph[3][2] = 0; graph[3][3] = 0; graph[3][4] = 1; graph[3][5] = 0;  
 graph[4][0] = 1; graph[4][1] = 0; graph[4][2] = 1; graph[4][3] = 1; graph[4][4] = 0; graph[4][5] = 1;  
 graph[5][0] = 0; graph[5][1] = 1; graph[5][2] = 0; graph[5][3] = 0; graph[5][4] = 1; graph[5][5] = 0;  
  
 printf("Started matrix:\n");  
 display\_matrix(graph, n);  
  
 if(is\_graph\_connect(graph, n)) {  
 int \*\*forest = kraskal\_algorithm(graph, n);  
 int \*\*bridges = taryans\_algorithm\_for\_finding\_bridges(graph, n);  
 bridges = split\_graph\_to\_two\_connected\_components(forest, bridges, n);  
 printf("\nMaximum set for split into two connected components:\n");  
 show\_graph\_difference(graph, bridges, n);  
 free\_matrix(forest, n);  
 free\_matrix(bridges, n);  
 }else  
 printf("Graph is not connected!");  
 free\_matrix(graph, n);  
 return 0;  
}

Изображение выглядит как снимок экрана, текст, черный, дизайн

Автоматически созданное описание Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

**Вывод:** на этой лабораторной работе я изучил алгоритм Краскала построения покрывающего леса, научился использовать его при решении различных задач.