BÁO CÁO BÀI TẬP
LÝ THUYẾT ĐỒ THỊ

I. Cấu trúc chương trình giải quyết các bài toán:

Bài 1: [THANHPHO] Tìm cây khung ngắn nhất
Hướng giải quyết: Sử dụng ma trận kề để lưu khoảng cách giữa các cạnh, sau đó dùng thuật toán Prim để tìm cây khung ngắn nhất.

- Cấu trúc:

```
struct Vertice                          // Cấu trúc của một đỉnh trong đồ thị
{
        float _x;                       x, y là tọa độ của đỉnh
        float _y;
        int _parent;                    lưu dấu phần tử trước nối với đỉnh
        float _value;                   lưu khoảng cách từ đỉnh parent đến đỉnh
        bool _visited;                  đánh dấu đỉnh đã thuộc cây khung hay chưa
};

class Graph                             // Cấu trúc của một đồ thị
{
private:
        Vertice *list;                  lưu danh sách các đỉnh
        int n;                          số lượng đỉnh trong đồ thị
        float **map;                    ma trận kề lưu khoảng cách giữa các đỉnh
};
```

- Các phương thức:

```
float Distance(Vertice u, Vertice v)            // Hàm tính khoảng cách giữa đỉnh u và đỉnh v
{
        return sqrt((u._x - v._x) * (u._x - v._x) + (u._y - v._y) * (u._y - v._y));
}

void Graph::CreateDistance()                    // Hàm tạo giá trị cho ma trận kề
{
        map = new float*[n];                    // Khởi tạo ma trận kề n * n (n: số đỉnh dồ thị)
        for (int i = 0; i < n; i++)
                map[i] = new float[n];

        for (int i = 0; i < n; i++)
        {
                map[i][i] = 0;                  // K/c từ một đỉnh tới chính nó bằng 0
                for (int j = i + 1; j < n; j++) // Do đồ thị vô hướng nên có tính đối xứng
                {
                        float tmp = Distance(list[i], list[j]);
                        map[i][j] = tmp;
                        map[j][i] = tmp;
                }
```

```
	}
}

Graph::Graph()                                  // Hàm khởi tạo đồ thị từ file dữ liệu
{
	fin >> n;
	list = new Vertice[n];

	for (int i = 0; i < n; i++)             // Mặc định ban đầu
	{
		fin >> list[i]._x >> list[i]._y;
		list[i]._parent = -1;                   parent của mỗi đỉnh là -1
		list[i]._value = MAX_INT;               value = +∞
		list[i]._visited = false;               chưa có đỉnh nào thuộc cây khung
	}
}

void Graph::Prim()                              // Tìm cây khung ngắn nhất với Prim
{

	list[0]._value = 0;                     // Bắt đầu duyệt từ đỉnh 0 nên value[0] = 0

	for (EdgeCount = 0; EdgeCount < n; EdgeCount++)// Do có n đỉnh nên sẽ có (n – 1) cạnh
	{
		min = MAX_INT;
		for (int i = 0; i < n; i++)     // Chọn đỉnh u chưa có trong cây khung có value nhỏ nhất
			if (!list[i]._visited && list[i]._value < min)
			{
				min = list[i]._value;
				u = i;
			}

		list[u]._visited = true;        // Đánh dấu điểm tìm được thuộc cây khung

		for (int v = 0; v < n; v++)     // Tính lại value[v] với v chưa thuộc cây khung
			if (map[u][v] < list[v]._value && u != v & !list[v]._visited)
			{
				list[v]._parent = u; //Lưu vết, đỉnh nối với v cho k/c ngắn nhất là đỉnh u
				list[v]._value = map[u][v]; // Tối ưu value[v]
			}
	}
}


void Graph::PrintResult()
{
	float sum = 0; // Tổng chi phí để xây dựng cây khung = tổng các value
	int u;
	for (int i = 0; i < n; i++)
		sum += list[i]._value;
```

```
        for (int v = 1; v < n; v++) // Các cạnh của cây khung là parent[i] - i
        {
                u = list[v]._parent;
                fout << list[u]._x << " " << list[u]._y << " " << list[v]._x << " " << list[v]._y << endl;
        }
}
```

- Nhận xét:

Một cách lập trình đơn giản sử dụng ma trận kề và tìm kiếm toàn bộ mảng để tìm cạnh có trọng số nhỏ nhất có thời gian chạy $O(V^2)$. Để cải tiến chương trình có thể sử dụng cấu trúc dữ liệu đống nhị phân và danh sách kề, có thể giảm thời gian chạy xuống O(E log V) và cấu trúc dữ liệu đống Fibonacci phức tạp hơn, có thể giảm thời gian chạy xuống O(E + V log V), nhanh hơn thuật toán trước khi đồ thị có số cạnh E=ω(V).

Bài 2: [CITIES] Tìm cây khung ngắn nhất

Hướng giải quyết: Sử dụng danh sách kề để lưu các cạnh, sau đó dùng thuật toán Kruskal để tìm cây khung ngắn nhất.

- Cấu trúc:

```
#define edge pair<int,int>              // Cấu trúc của một cạnh, với mỗi đầu là chỉ mục của đỉnh

class Graph
{
private:
        vector<pair<float, edge>> G;    // Đồ thị ban đầu, có n đỉnh, n * (n – 1) cạnh
        vector<pair<float, edge>> T;    // Đồ thị cây khung cón đỉnh, (n – 1) cạnh
        int *parent;                    // Lưu dấu gốc
        int V;                          // Số đỉnh của đồ thị
public:
};
```

- Các phương thức:

```
// Thêm cạnh vào danh sách liên kết, w là k/c giữa đỉnh u và đỉnh v
void Graph::AddWeightedEdge(int u, int v, float w)
{
        G.push_back(make_pair(w, edge(u, v)));
}

int Graph::find_set(int i) // Trả id của tập hợp chứa i
{
        // Nếu đỉnh i là gốc
        if (i == parent[i])
                return i;
        else
                // Ngược lại thì đệ quy tìm gốc của parent[i]
                return find_set(parent[i]);
}

void Graph::union_set(int u, int v) // Nối 2 đỉnh u và v
```

```
{
	parent[u] = parent[v];
}

Graph::Graph() // Khởi tạo đồ thị
{
	// Đọc dữ liệu từ file
	fin >> V;
	Point *list = new Point[V];
	for (int i = 0; i < V; i++)
		fin >> list[i]._id >> list[i]._x >> list[i]._y;
	fin.close();

	// Khởi tạo đồ thị ban đầu
	parent = new int[V];

	for (int i = 0; i < V; i++)
		parent[i] = i;

	G.clear();
	T.clear();

	// Tạo danh sách liên kết
	Point u, v;
	for (int i = 0; i < V; i++)
	{
		for (int j = i + 1; j < V; j++)
		{
			u = list[i];
			v = list[j];
			float tmp = Distance(u, v);
			AddWeightedEdge(u._id - 1, v._id - 1, tmp);
			AddWeightedEdge(v._id - 1, u._id - 1, tmp);
		}
	}
	delete[] list;
}

void Graph::kruskal() // Thuật toán tìm cây khung ngắn nhất với Kruskal
{
	int i, uRep, vRep;
	sort(G.begin(), G.end()); // Sắp xếp danh sách các cạnh theo trọng số w

	for (i = 0; i < G.size(); i++)
	{
		// Tìm đỉnh u và v phù hợp
		uRep = find_set(G[i].second.first);
		vRep = find_set(G[i].second.second);
		if (uRep != vRep)
		{
```

```
                T.push_back(G[i]); // Thêm cạnh đó vào đồ thị cây khung
                union_set(uRep, vRep);
            }
        }
}
```

- Nhận xét:
- Đồ thị G(V, E), sắp xếp các cạnh của đồ thị theo trọng số mất thời gian O(ElogE) = O(ElogV) (vd sử dụng quicksort), mỗi thao tác của cấu trúc Union-Find (kết hợp nén đường đi) mất thời gian α(V) trong đó α(.) là hàm Ackerman ngược. Hàm này có giá trị ≤5 với mọi giá trị thực tế của V. Do đó, vòng lặp for sẽ mất thời gian Eα(V). Do đó, tổng thời gian của thuật toán là O(ElogV+Eα(V))= O(ElogV). (nguồn: www.giaithuatlaptrinh.com)
- Thuật toán Kruskal không đạt hiệu quả cao với đồ thị dày đặc (E = (n-1)n/2)

Bài 3: [BITMAP] Đếm số thành phần liên thông theo liên thông 4

Hướng giải quyết: Duyệt theo chiều rộng

```
// Các hướng duyệt trên, dưới, trái, phải
const int ax[] = { -1, 1, 0, 0 };
const int ay[] = { 0, 0, -1, 1 };

// Dùng ma trận kề để lưu đồ thị
int g[255][255];

void bfs(int x, int y) // Thuật toán duyệt theo chiều rộng
{
    if (g[x][y] != 1) // Nếu (x, y) không phải điểm đen thì không xử lí
        return;

    queue<int> qx, qy; // (x, y) là điểm đen thì đẩy vào queue
    qx.push(x);
    qy.push(y);
    g[x][y] = -1; // đánh dấu (x, y) đã được xử lí

    while (!qx.empty()) // Duyệt cho đến khi queue rỗng
    {
        // Lấy phần tử (x, y) đầu của queue để xét
        int nx = qx.front();
        int ny = qy.front();
        qx.pop();
        qy.pop();

        for (int i = 0; i < 4; i++) // Duyệt 4 điểm lân cận của (x, y)
        {
            int tx = nx + ax[i];
            int ty = ny + ay[i];

            if (g[tx][ty] == 1) // Nếu đó là điểm đen thi xử lí
            {
                g[tx][ty] = -1; // Đánh dấu đã duyệt
```

```
                        qx.push(tx); // Đẩy vào queue
                        qy.push(ty);
                    }
                }
            }
}


void main()
{
        // Duyệt bảng để đếm số lượng đối tượng (Do có nhiều thành phần liên thông nên phải duyệt từng điểm để đếm đủ số lượng thành phần liên thông)
        int count = 0;
        for (int i = 0; i <= n; i++)
                for (int j = 0; j <= m; j++)
                        if (g[i][j] == 1)
                        {
                                count++;
                                bfs(i, j);
                        }
}
```

- Nhận xét:
- Không gian: V là số đỉnh thì không gian cần dùng của thuật toán là O(V)
- Thời gian: Nếu V và E là số đỉnh và số cạnh của đồ thị, thì thời gian thực thi của thuật toán là O(E + V) vì trong trường hợp xấu nhất, mỗi đỉnh và cạnh của đồ thị được thăm đúng một lần. với O(V) <= O(E + V) <= O(V^2)

Bài 4: [SARS] Tìm thành phần liên thông
Hướng giải quyết: Duyệt theo chiều sâu

- Cấu trúc:

```
class Node                                      // Cấu trúc của một nút trong danh sách kề
{
        int _value;     // chỉ mục của nút
        Node * _next; // con trỏ trỏ đến nút tiếp theo
};

class List                                      // Cấu trúc của đồ thị
{
private:
        bool *_visited;         // đánh dấu các đỉnh đã được xử lí hay chưa
        int _itemCount;         // số lượng đỉnh trong đồ thị
        int _beginner;          // người bị bệnh đầu tiên
        Node ** _map;           // danh sách kề của các đỉnh
};
```

- Các phương thức:

```
List::List() // Khởi tạo dữ liệu từ file
{
```

```
	fin >> _itemCount >> _beginner;

	for (int i = 1; i <= _itemCount; i++)
	{
		_map[i] = NULL;
		_visited[i] = false; //  đánh dấu đỉnh i chưa được xử lí

		if (k != 0)
			for (int j = 1; j <= k; j++) // Tạo liên kết giữa người i và người x
				Insert(i, x);
	}
}

void List::Insert(int id, int value)
{
	Node * k = new Node(value);

	if (_map[id] == NULL)
	{
		_map[id] = k;
		return;
	}

	Node * tmp = _map[id];
	Node * prev = NULL;
	while (tmp != NULL)
	{
		if (tmp->getValue() == value)
			return;
		prev = tmp;
		tmp = tmp->getNext();
	}

	if (tmp == NULL)
		prev->setNext(k);
}

void List::DFS(int u)
{
	_visited[u] = true; // đánh dấu đỉnh u đã đi qua
	for (Node * v = _map[u]; v; v = v->getNext()) // với mỗi đỉnh v kề với u
		if (!_visited[v->getValue()]) // nếu v chưa đánh dấu, tới thăm đỉnh v
			DFS(v->getValue());
}

void List::FindConnect()
{
	DFS(_beginner); // Tìm những người đã tiếp xúc với người mắc bệnh đầu tiên và người bị ảnh hưởng
```

```
    int count = 0;
    for (int i = 1; i <= _itemCount; i++)
        if (_visited[i]) // Nếu đỉnh i đã được thăm thì i đã tiếp xúc với người bệnh
            count++;

    for (int i = 1; i <= _itemCount; i++)
        if (_visited[i])
            fout << i << " ";
}
```

- Nhận xét:

- Không gian: Độ phức tạp không gia thấp hơn BFS vì không cần sử dụng queue để lưu dấu
- Thời gian: Nếu V và E là số đỉnh và số cạnh của đồ thị, thì thời gian thực thi của thuật toán là O(E + V) vì trong trường hợp xấu nhất, mỗi đỉnh và cạnh của đồ thị được thăm đúng một lần. với O(V) <= O(E + V) <= O(V^2), tương đương BFS

II. Đánh giá:

Mức độ hoàn thành: 100%