LAB2

Wu Jiayao 517370910257

1. C programming

1.1 Kruskal implementation

lab2.h

```c
#ifndef LAB2_H
#define LAB2_H
#include <stdio.h>
#include <stdlib.h>
#define MAX 1000
typedef struct _edge
{
    int u, v, w;
} Edge;
typedef struct _result
{
    int u, v;
} Result;
int *parent;
Result *res;
Edge *graph;
int Find(int);
void Union(int, int);
void UFset(int);
int kruskal(int);
int Ecmp(const void *, const void *);
int Rcmp(const void *, const void *);
int prim(int, int);
#endif
```

UFset.c

```c
#include "lab2.h"
void UFset(int e)
{
    for (int i = 0; i < e; i++)
    {
        parent[i] = -1;
    }
}

int Find(int x)
{
```

```c
    if (parent[x] < 0)
    {
        return x;
    }
    return parent[x] = Find(parent[x]);
}

void Union(int u, int v)
{
    int r1 = Find(u);
    int r2 = Find(v);
    if (r1 > r2)
    {
        parent[r2] += parent[r1];
        parent[r1] = r2;
    }
    else
    {
        parent[r1] += parent[r2];
        parent[r2] = r1;
    }
}

```

kruskal.c

```c
#include "lab2.h"
int kruskal(int e)
{
    int num = 0;
    int u, v;
    UFset(e);
    for (int i = 0; i < e; i++)
    {
        u = graph[i].u;
        v = graph[i].v;
        if (Find(u) != Find(v))
        {

            num++;
            Union(u, v);
        }
    }
    return num;
}

```

cmp.c

```c
#include "lab2.h"
int Ecmp(const void *a, const void *b)
{
    Edge *e1 = (Edge *)a;
    Edge *e2 = (Edge *)b;
    return e1->w - e2->w;
}
```

```c

int Rcmp(const void *a, const void *b)
{
    Result *r1 = (Result *)a;
    Result *r2 = (Result *)b;
    return (r1->u == r2->u) ? (r1->v - r2->v) : (r1->u - r2->u);
}

```

main.c

```c
#include "lab2.h"
int main()
{
    int v = 0, e = 0;
    scanf("%d", &e);
    scanf("%d", &v);
    parent = malloc(sizeof(int) * MAX);
    graph = malloc(sizeof(Edge) * MAX);
    res = malloc(sizeof(Result) * MAX);
    for (int i = 0; i < e; i++)
    {
        int tmpU = 0, tmpV = 0;
        scanf("%d %d %d", &tmpU, &tmpV, &graph[i].w);
        int min = tmpU < tmpV ? tmpU : tmpV;
        int max = tmpU > tmpV ? tmpU : tmpV;
        graph[i].u = min;
        graph[i].v = max;
    }
    qsort(graph, e, sizeof(Edge), Ecmp);
    //switch between kruskal and prim

    int MSTNum = kruskal(e);
    //int MSTNum = prim(e,v);


    qsort(res, MSTNum, sizeof(Result), Rcmp);
    for (int i = 0; i < MSTNum; i++)
    {
        printf("%d--%d\n", res[i].u, res[i].v);
    }
    free(graph);
    free(res);
}

```

1.2 Complexity

Since disjoint sets takes $O(n)$ for find and union, sorting edges by weight takes $O(E \log E)$, Kruskal's algorithm has a time complexity of $O(E \log E)$, where E is the number of edges in the graph.

If to be optimized by Fibonacci Heap, Prim's algorithm can run in $O(E + V \log V)$ times, where E is the number of edges in the graph, V the number of vertices.

Prim's algorithm perform better in dense graph with lots of edges, while Kruskal's performs better in sparse graph with less edges.

2. The *with* statement

The "with" statement is used to wrap the execution of a block with methods defined by a context manager.

```
with open('in.txt','r') as file:
    for line in file:
        print(line.strip())
```

This is equal to

```
try:
    file = open('in.txt','r')
    for line in file:
        print(line.strip())
except:
    print('Fail to open file')
finally:
    file.close()
```

3. Decorator

A decorator is any callable Python object that is used to modify a function, method or class definition. the original object being defined is passed into a decorator. The decorator returns a modified object,

```
# To print the name of a function before execution
def cheer(func):
    def wrapper(*args,**kw):
        print('Cheer for %s():' % func.__name__)
        return func(*args,**kw)
    return wrapper
# Define function Using the decorator
@cheer
def zzNumberOne():
    print('ZZ, world Number One.')
#Or modify defined function
def zzExcellent():
    print('ZZZ, a student that is excellent in algorithm design.')
zzExcellent = cheer(zzExcellent)
```

4. Iterators

An iterator is an object that can be iterated upon, meaning that you can traverse through all the values. All iterable objects can be transformed into iterator.

```python
strong = ("zz","gg","fs")
it = iter(strong)
print(it)
print(next(myit))
print(next(myit))
print(next(myit))
# Output:
# <tuple_iterator object at XXXXXXX>
# zz
# gg
# fs
```

5. Generators

Generator functions can declare a function that behaves like an iterator,

```python
def jfmm(zz):
    n,a,b = 0,0,1
    while n < zz:
        yield b
        a,b = b,a+b
        n = n+1
for n in fab(5):
    print(n)

```