Yu xinmiao 于欣淼 518021910792

VE477 Lab2

C programming

1. The `Union-Find` data structure && Kruskal's algorithm

   `Union-Find` is used when implement the Kruskal's algorithm.

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

typedef struct graph_t Graph;
typedef struct edge_t Edge;
typedef struct node_t Node;
struct edge_t{
    int v1, v2, wt;
};
struct graph_t{
    int Vnum, Enum;
    Edge* edge;
};
struct node_t{
    int parent;
    int rank;
};
Graph* createGraph(int V, int E){
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->Vnum = V;
    graph->Enum = E;
    graph->edge = (Edge*)malloc(sizeof(Edge)*E);
    return graph;
};
void addEdge(Graph* graph, int E) {
    for (int i = 0; i < E; i++) {
        int v1, v2, wt;
        scanf("%d %d %d", &v1, &v2, &wt);
```

```
        if (v1 > v2) {
            int temp = v2;
            v2 = v1;
            v1 = temp;
        }
        graph->edge[i].v1 = v1;
        graph->edge[i].v2 = v2;
        graph->edge[i].wt = wt;
    }
}

int compare(const void * a, const void * b){
    // compare function to sort the edge according to weight
    Edge * e1 = (Edge*) a;
    Edge * e2 = (Edge*) b;
    if (e1->wt > e2->wt) return 1;
    else if (e1->wt < e2->wt) return -1;
    else return 0;
}
int vertexCom(const void*a, const void*b) {
    // compare function to sort the vertex
    Edge* e1 = (Edge*)a;
    Edge* e2 = (Edge*)b;
    if (e1->v1 < e2->v1) return -1;
    else if (e1->v1 > e2->v1) return 1;
    else return e1->v2 > e2->v2;
}

// Union-Find data structure
int Find(Node* nodes, int i) {
    if (nodes[i].parent != i)
        nodes[i].parent = Find(nodes, nodes[i].parent); //
update parent among the path
    return nodes[i].parent;
}
void Union(Node* nodes, int x, int y){
    int xRoot = Find(nodes, x);
    int yRoot = Find(nodes, y);
    // make the node with lower rank as the parent
    if (nodes[xRoot].rank < nodes[yRoot].rank)
        nodes[xRoot].parent = yRoot;
    else if (nodes[xRoot].rank > nodes[yRoot].rank)
```

```
        nodes[yRoot].parent = xRoot;
    else {
        nodes[xRoot].parent = yRoot;
        nodes[xRoot].rank++;
    }
}
void kruskalMST(Graph* graph){
    // sort based on the weight
    qsort(graph->edge, graph->Enum, sizeof(graph->edge[0]),
compare);
    Edge result[graph->Vnum - 1];
    Node* nodes = (Node*)malloc(sizeof(Node) * graph->Vnum);
    for (int i = 0; i < graph->Vnum; i++) {
        nodes[i].parent = i;
        nodes[i].rank = 0;
    }

    int eTaken = 0;
    int e = 0;
    while (eTaken < graph->Vnum - 1 && e < graph->Enum) {
        // for the edge in non-decreasing order
        Edge next = graph->edge[e++];
        int v1Root = Find(nodes, next.v1);
        int v2Root = Find(nodes, next.v2);
        if (v1Root != v2Root) { // no cycle created
            result[eTaken++] = next;
            Union(nodes, v1Root, v2Root); // union
        }
    }
    free(nodes);
    qsort(result, graph->Vnum - 1, sizeof(result[0]),
vertexCom);
    for (int i = 0; i < graph->Vnum - 1; i++) {
        printf("%d--%d\n", result[i].v1, result[i].v2);
    }
}

int main() {
    int Vnum, Enum;
    scanf("%d", &Enum);
    scanf("%d", &Vnum);
    Graph * graph = createGraph(Vnum, Enum);
```

```
    addEdge(graph, Enum);
    kruskalMST(graph);
    free(graph->edge);
    free(graph);
    return 0;
}
```

Time complexity for $G = (V, E)$,

- sort the edges according to the weight: using `qsort`, $\mathcal{O}(ElogE)$
- For sorted edge, add the edge if no cycle created. Create `parent` to store `parent` and `rank`: $|V|$ operations. In for loop, `Find` and `Union` operations are performed in $\mathcal{O}(E)$ times, as we use union-by-rank and path-compression mehods, the operations should grow in a very slow rate when V increases. Such that we could state that totally $\mathcal{O}((E+V)\alpha(V))$ time, where α is inverse Ackerman's function.

As $|E| \geq |V| - 1$, so takes totally $\mathcal{O}(E\alpha(V))$ times of operation. So for the whole algorithm, the time complexity is $\mathcal{O}(ElogE)$.

(ignore the part that after find MST, sort to print)

2. Prim's algorithm

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
typedef struct node_t node;
struct node_t {
    int v1; int v2;
};

int cmp (const void*a, const void*b) {
  // compare function for qsort, to output in correct order
    node* n1 = (node*) a;
    node* n2 = (node*) b;
    if (n1->v1 < n2->v1) return -1;
    else if (n1->v1 > n2->v1) return 1;
    else return n1->v2 > n2->v2;
 }
void printGraph(int vNum, const int parent[]) {
```

```
    node* nodes = (node*)malloc(sizeof(node) * vNum);
    nodes[0].v1 = 0; nodes[0].v2 = 0;
    for (int i = 1; i < vNum; ++i) {
        if (i > parent[i]) {nodes[i].v1 = parent[i]; nodes[i].v2
= i;}
        else {nodes[i].v1 = i; nodes[i].v2 = parent[i];}
    }
    qsort(nodes, vNum, sizeof(nodes[0]), cmp);
    for (int i = 1; i < vNum; i++) {
        printf("%d--%d\n", nodes[i].v1, nodes[i].v2);
    }
    free(nodes);
}

int minDis(const int distance[], const bool added[], int vNum) {
  // find the minimum distance among not added node
    int minimum = INT_MAX;
    int index = 0;
    for (int i = 0; i < vNum; i++) {
        if (distance[i] < minimum && !added[i]) {
            minimum = distance[i];
            index = i;
        }
    }
    return index;
}

void primMST(int** graph, int vNum) {
    int parent[vNum]; int distance[vNum]; bool added[vNum];
    for (int i = 0; i < vNum; i++) {
        distance[i] = INT_MAX;
        added[i] = false;
    }
    distance[0] = 0; added[0] = true; parent[0] = -1; // add the
first element
    for (int i = 0; i < vNum - 1; i++) {
        int v = minDis(distance, added, vNum);
        added[v] = true; // add the node with minimum distance
        for (int u = 0; u < vNum; ++u) {
            if (graph[v][u] && !added[u] && distance[u] >
graph[u][v]){
                parent[u] = v; // update
```

```
                distance[u] = graph[u][v];
            }
        }
    }
    printGraph(vNum, parent);
}

int main (){
    int vNum, eNum;
    scanf("%d", &eNum);
    scanf("%d", &vNum);
    int** graph = (int**)malloc(sizeof(int*) * vNum);
    for (int i = 0; i < vNum; i++) graph[i] =
(int*)malloc(sizeof(int) * vNum);
    for (int i = 0; i < eNum; i++) {
        int v1, v2, wt;
        scanf("%d %d %d", &v1, &v2, &wt);
        graph[v1][v2] = graph[v2][v1] = wt;
    }
    primMST(graph, vNum);
    for (int i = 0; i < vNum; i++) free(graph[i]);
    free(graph);
    return 0;
};
```

Time complexity for $G = (V, E)$,

If implement with an adjacency matrix, the time complexity is $\mathcal{O}(V^2)$, which is caused by the traversal. However, if optimazied through Fibnacci Heap, the running time will be $\mathcal{O}(E + VlogV)$.

3. Compare

   It depends on the type of graphs. For **dense graph**, Prim algorithm will have a better performance. For **sparse graph**, Kruskal algorithm will have a better performance.

1 With Statement

Generally, `with` statement is used to wrap the execution of a block with With Statement Context Managers. It simplifies the management of common resources. `with` ensures proper acquisition and release of resources, so helps avoiding bugs and leaks. Previously, the similar usage might be achieved through `try...finally...` block, `with` make it more simple and readable.

[ ]:
```
# basic structure
with expression as [variable]:  # result in an object that supports the context␣
 ↪manager protocol
    # such that, has _enter_() / _exit_() methods
    # The object's _enter_ is called before block execution
    # also could return a value bound to [variable]
    block_execution
    # after finish block_execution, _exit_() is called

# file handling
with open('path', 'w') as file:
    file.write('hello world!')  # do not need to manually close file

# in user defined objects
class OwnWriter(object):
    def _init_(self, file_name):
        self.file_name = file_name

    def _enter_(self):
        self.file = open(self.file_name, 'w')
        return self.file

    def __exit__(self):
        self.file.close()

with OwnWriter("file_name") as file:
    file.write("hello world")
```

2 Decorator

Functions are the first class objects. Decorators make it possible to wrap a function to add more behaviors, but do not need to permanently modifying it.

[10]:
```
def this_decorator(func):  # define a decorator
    print("before execution")
    def inner(*args, **kw):
        value = func(*args, **kw)
        print("after execution")
```

```
        return value
    return inner
```

[11]:
```
@this_decorator
def now():  # equal to now = this_decorator(now); now points to a new funciton
 ↪inner
    print("2020") # before inner, before execution is printed
```

```
before execution
```

[12]:
```
now()
```

```
2020
after execution
```

[20]:
```
@this_decorator
def sum_number(a, b):
    print("execution")
    return a + b
```

```
before execution
```

[23]:
```
print("sum=", sum_number(5,6))  # value is returned after execution
```

```
execution
after execution
sum= 11
```

3 Iterators

an object that is used to iterate over iterable objects.

[25]:
```
iterable_list = [1, 2, 3, 4, 5]
iterable_obj = iter(iterable_list)  # iter(iterable) is called to initialze an
 ↪iterator, returns an iteratot object
while True:
    try:
        item = next(iterable_obj)  # next(iterator) returns the next value. When
 ↪end, raise StopIteration
        print(item)
    except StopIteration:
        break
```

```
1
2
3
```

```
4
5
```

[26]:
```python
# some built-in iterator

print("list")
l = ["one","two","three"]
for i in l:
    print(i)

print("tuple")
t = ("one","two","three")
for i in t:
    print(i)

print("string")
s = "one"
for i in s:
    print(i)
```

```
list
one
two
three
tuple
one
two
three
string
o
n
e
```

[28]:
```python
# to test whether an object is Iterable, so that we could use iterator
from collections.abc import Iterable
print(isinstance('123', Iterable))
print(isinstance(123, Iterable))
```

```
True
False
```

4 Generator

- `yield` keyword: suspends the function execution and sends value back to the caller, but it will remain the state so that the function could start from where left.
- **Generator Function**: defines as a normal function, but use `yield` instead of `return` when

need to generate the value. **Generator Object**: generator function will return a generator object. Generator object could be iterated by `next` or using iterators.

[29]:
```
def fib(max):  # with `yield`, becomes a generator function
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'
```

[47]:
```
fib(6)  # call the generator funciton, a generator object return
```

[47]:
```
<generator object fib at 0x7f81d18aaa50>
```

[46]:
```
f = fib(6)
while True:
    try:
        print(next(f))  # generator execute when call next, return when yield.␣
 ↪next exectuion will begin from yield
    except StopIteration as e:
        print(e.value)
        break
```

```
1
1
2
3
5
8
done
```

[48]:
```
for i in fib(6):  # using loop to iterate over the generator object
    print(i)
```

```
1
1
2
3
5
8
```

[ ]:

[ ]: