VE477 Lab2 Report

1 C programming

The time complexity for Kruskal's algorithm is `O(V+ElogE)`, while the time complexity for Prim's algorithm is `O(E+VlogV)` In practice, the Kruskal's algorithm is far more efficient and faster than Prim's algorithm, since we often encounter graphs with sparse graph. But for the graphs which are very dense, there will be far more edges than vertices. In this case the Prim's algorithm may have a better efficiency than the Kruskal's algorithm.

The codes will be attached at the end of this report.

2.1 More documentation

How to define an anonymous function in OCaml? When to define and use anonymous functions?

Since the function is the "first class citizen" in `OCaml`, it should accept some literals, just like `22` in `int` or `"Hello world"` in `string`. To define an anonymous function in `OCaml`, we just write

```
let (fun x -> x + 1);; (* increament the value by one
                        you can difine a none anonymous function like
                        let plusone = fun x - > x + 1;; or
                        let plusone x -> x + 1;;
```

To use it, we can directly put the parameter at the back

```
let (fun x -> x + 1)7;; (* which will give 8 as result
```

As most of the literals, the anonymous functions are usually passed as function inputs. Actually, the anonymous function can be in the place wherever a function is required.

How should variables and functions be named (e.g. capital letters, underscore)?

The function and variables should be named in lower case. And the name of the variable should be meaningful.

What is a module? How does it differ from an class in OOP?

The module is like a package in `Java` or a `.h` file in `C/C++` which contains a piece of code for other modules to use. It may contain some predefined functions or variables. A class in OOP is a basic object that may contains some data and methods. The main difference is that a class can be inherited in order to add some data and methods to it. While a module should be used when the data structure are fixed and it is enough to add some extra function in the program which use the module.

Life without arrays is not simple. How can Lists help?

The `List` in `OCaml` can contain elements in the same type, which is very similar to the array in other languages. It also provides some useful methods like `::` to add elements at the beginning and `@` to combine to lists. We can also have `List` of `List`, which can perform the same operation as two dimensional array does.

What are maps and iterators? How to best use them?

`Maps` are some functions that map the input to output. Specifically, a map can map a list or multiple lists to target list or lists using the corresponding function. `iterators` are functions that will be applied to each of the element in the list one by one. To best use them, I think `maps` should be used to deal with some uniform functions that deal with the list as a whole. While the `iterators` should be used to deal with some individual functions like `print`.

Foldings are very powerful features. Explain why and provide a simple example different from the ones in the documentation.

The folding can perform like an iterator and operate on the whole list based on a given function. Also, we can give an initial input as accumulator value. The accumulator value can be carefully chosen in order to implement different target. For example

```
let div_2 l =
    List.fold_left (fun e a -> e :: a/2) [] l ;;
```

What is tail recursion? Why is it an important point in functional programming?

Sometimes the recursion will result in a huge amount of arithmetic calculations like a long adding queue or a long multiplication queue. These command may be too long for the stack to store them. So we may introduce the accumulator to save space. Namely, we calculate the arithmetic part of the recursion and store them in the accumulator and then do the next recursion step.

What does `ref` mean, and when to use this keyword?

`ref` in `OCaml` means that we take the reference of a certain variable. For example, we declare that `ref x` to mean that we get the reference of `x`. This `ref` is usually used when we want to define a pointer.

What are `functors`, how do they relate and differ from templates in C++?

The `functors` are modules that are parametrized by other modules. It will take one module and return a new module according to some additional informations. It in a sense gives some constrains on the module and make it smaller. It is kind of like a template in `C++`, but note that a template in `C++` can only have different variable types. But `functors` and accept other constrains like the size of an input array.

How to define new types?

We can define type by defining the type name with lower case letters and constructors with upper class letters. For example,

```
type colour =
    | Red
    | Blue
    | Green
    | RGB of float * float * float (* We can define with different forms*)
    | Mix of float * colour * colour (*we can even use recursive definition*)
```

What are sum and product types? How do they help improving coding quality?

A `sum type` is a combination of two types. Since the possible value of the result type is the sum of the input types, it is called `sum type`. `Product type` is a combination of two types with the possible value of the result type being the product of the input types. For example, we have a `sum type` like:

```
// imagine that rather than sets here we have types that can only have these
values
const bools = new Set([true, false])
const halfTrue = new Set(['half-true'])

// The weakLogic type contains the sum of the values from bools and halfTrue
const weakLogicValues = new Set([...bools, ...halfTrue])
```

we have a `product type` like:

```
// point :: (Number, Number) -> {x: Number, y: Number}
const point = (x, y) => ({ x, y })
```

These types can combine two values and make them as one value, which reduces the numbers of further manipulations.

2.2 Coding

```
let a = read_line();; (*read in a line*)
let a =  String.split_on_char ',' a;; (*split the line into string lists
according to the position of comma*)
let b = List.map (String.trim) a;; (*now get rid of the spaces at the end of each
string list*)
let numbers = List.map (int_of_string) b;; (*convert the string to integers*)

(*following is the sort function, just follows the recurivse definition of quick
sort*)
let rec quick_sort = function
  | [] -> []
  | x::xs -> let left, right = List.partition (fun y -> y < x) xs (*choose the
first element as pivot*)
    in ( quick_sort left ) @ ( x::quick_sort right ) ;;

let result = quick_sort numbers;;
List.iter (fun x -> Printf.printf "%d " x) result ;;
```

The average time complexity is `O(nlogn)`, since the mapping part deal with `n` elements and dealing with each element takes `O(1)` time and the sort part has an average time complexity `O(nlogn)`. In the worst case, the elements are totally reversed, then it will take `O(n²)` time.

Source Code

1.Prim

```
#include<stdio.h>
#include"union_find.h"

int min(int a, int b) {
    return (a >= b) ? b : a;
}

int alles_zero(edge ** arr, int size) {
    int flag = 1;
    for(int i = 0; i < size; i++) {
        if(arr[i]->weight != 0) {
            flag = 0;
            break;
        }
    }
    return flag;
}

int main() {
    int eSize = 0;
    int vSize = 0;
    char * buffer = NULL;
    size_t input_size = 1;

    //read in the first line and convert it to int
    getline(&buffer,&input_size, stdin);
    eSize = atoi(buffer);

    //read in the second line and convert it to int
    getline(&buffer,&input_size,stdin);
    vSize = atoi(buffer);

    //test
    //printf("eSize is = %d",eSize);
    //printf("vSize is = %d", vSize);

    //we first initialize all the nodes
    node ** nodes = (node **)malloc(vSize * sizeof(node *));
    for (int i = 0; i < vSize; i++) {
        nodes[i] = GenSet(nodes[i]);
    }

    //then read in edges
    edge ** edges = (edge **)malloc(eSize * sizeof(edge *));
    for(int i = 0 ; i < eSize; i++) {
        edges[i] = GenEdge(edges[i]);
    }
    for(int i = 0; i < eSize ; i++) {
        char string_1 [10];
        char string_2 [10];
        char string_3 [10];
        scanf("%s", string_1);
        scanf("%s", string_2);
```

```
        scanf("%s", string_3);

        //we ensure the first node is smaller
        if(atoi(string_1)<atoi(string_2)) {
            (edges[i]->node_1)->tag = atoi(string_1);
            (edges[i]->node_2)->tag = atoi(string_2);
        }
        else {
            (edges[i]->node_1)->tag = atoi(string_2);
            (edges[i]->node_2)->tag = atoi(string_1);
        }
        edges[i]-> weight = atoi(string_3);
    }

//sort the edges first by node_2 and then by node_1, this ensure the order
edges = sort_edge_by_node2(edges,0,eSize-1);
edges = sort_edge_by_node1(edges,0,eSize-1);

//now we are able to perform the Prim's algorithm

//A fact: every node should have exactly one edge from union to it
//declare a distance array to store the edges with smallest distance between the
Union and the other points. Initialize them to be max
edge ** distance = ( edge **)malloc(vSize * sizeof(int));
for(int i = 0; i < vSize; i++) {
    distance[i] = GenEdge(distance[i]);
}

distance[0]->node_1 = nodes[0];
distance[0]->node_2 = nodes[0];
distance[0]->weight = 0;


for (int i = 1; i < vSize; i++) {
    distance[i]->node_1 = NULL;
    distance[i]->node_2 = nodes[i];
    distance[i]->weight = __INT_MAX__;
}

//T is used for store the edges
edge ** T  = (edge **)malloc(eSize * sizeof(edge *));
int T_top = 0;

//the loop is terminated if the distance from union to every point is 0
while(alles_zero(distance,vSize) != 1) {
    //for every edge that has one end in the union and the other end no in the
union, update the corresponding distance
    for(int i = 0 ; i < eSize; i++) {
        if(Find(nodes[edges[i]->node_1->tag]) == Find(nodes[0]) &&
Find(nodes[edges[i]->node_2->tag]) != Find(nodes[0])) {
            distance[edges[i]->node_2->tag]->weight =  min(edges[i]->weight,
distance[edges[i]->node_2->tag]->weight);
            distance[edges[i]->node_2->tag]->node_1 = edges[i]->node_1;
        }
    }

    // we go through the distance array and choose the node with smallest
distance, union it with root and update the distance
```

```
    int min_distance = __INT_MAX__;
    int min_tag = 0;
    for (int i = 1; i < vSize ; i++) {
        if(distance[i]->weight > 0 && distance[i]->weight < min_distance) {
            min_distance = distance[i]->weight;
            min_tag = i;
        }
    }
    Union(nodes[0],nodes[min_tag]);
    distance[min_tag]->weight = 0;
    T[T_top] = distance[min_tag];
    T_top++;
}

    T = sort_edge_by_node2(T,0,T_top-1);
    T = sort_edge_by_node1(T,0,T_top-1);

    for(int i = 0 ; i < T_top-1; i++) {
         if(T[i]->node_1->tag == T[i+1]->node_1->tag && T[i]->node_2->tag >
T[i+1]->node_2->tag) {
            swap(T[i],T[i+1]);
         }
    }

    for(int i = 0; i < T_top; i++) {
        printf("%d--%d\n", T[i]->node_1->tag, T[i]->node_2->tag);
    }


    for(int i = 0; i < vSize ; i++) {
        free(nodes[i]);
    }
    for(int i = 0; i < eSize ; i++) {
        DesEdge(edges[i]);
    }
    free(nodes);
    free(edges);
    free(distance);
return 0;
}
```

2.Kruskal

```
#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include"union_find.h"

int main() {
    int eSize = 0;
    int vSize = 0;
    char * buffer = NULL;
    size_t input_size = 1;

    //read in the first line and convert it to int
    getline(&buffer,&input_size, stdin);
    eSize = atoi(buffer);
```

```
    //read in the second line and convert it to int
    getline(&buffer,&input_size,stdin);
    vSize = atoi(buffer);

    //test
    //printf("eSize is = %d",eSize);
    //printf("vSize is = %d", vSize);

    //we first initialize all the nodes
    node ** nodes = (node **)malloc(vSize * sizeof(node *));
    for (int i = 0; i < vSize; i++) {
        nodes[i] = GenSet(nodes[i]);
    }

    //then read in edges
    edge ** edges = (edge **)malloc(eSize * sizeof(edge *));
    for(int i = 0 ; i < eSize; i++) {
        edges[i] = GenEdge(edges[i]);
    }
    for(int i = 0; i < eSize ; i++) {
        char string_1 [10];
        char string_2 [10];
        char string_3 [10];
        scanf("%s", string_1);
        scanf("%s", string_2);
        scanf("%s", string_3);

        //we ensure the first node is smaller
        if(atoi(string_1)<atoi(string_2)) {
            (edges[i]->node_1)->tag = atoi(string_1);
            (edges[i]->node_2)->tag = atoi(string_2);
        }
        else {
            (edges[i]->node_1)->tag = atoi(string_2);
            (edges[i]->node_2)->tag = atoi(string_1);
        }
        edges[i]-> weight = atoi(string_3);
    }

    //test
    // for(int i = 0; i < eSize; i++) {
    //     printf("%d ",edges[i]->node_1->tag);
    //     printf("%d ", edges[i]->node_2->tag);
    //     printf("%d ", edges[i]->weight);
    //     printf("\n");
    // }

    //sort thr edges to be non-decreasing order
    sort_edge(edges, 0, eSize-1);

    //test
    // for(int i = 0; i < eSize; i++) {
    //     printf("%d ",edges[i]->node_1->tag);
    //     printf("%d ", edges[i]->node_2->tag);
    //     printf("%d ", edges[i]->weight);
    //     printf("\n");
    // }
```

```
    edge ** T  = (edge **)malloc(eSize * sizeof(edge *));
    int T_top = 0;
    for(int i = 0; i < eSize; i++) {

        //if the two points in the edge is not connected before
        if(Find(nodes[edges[i]->node_1->tag]) != Find(nodes[edges[i]->node_2-
>tag])) {
            //add the edge to the stack
            T[T_top] = edges[i];
            T_top++;

            //union two points
            Union(nodes[edges[i]->node_1->tag], nodes[edges[i]->node_2->tag]);
        }
    }



    T = sort_edge_by_node2(T,0,T_top-1);
    T = sort_edge_by_node1(T,0,T_top-1);

    for(int i = 0 ; i < T_top-1; i++) {
         if(T[i]->node_1->tag == T[i+1]->node_1->tag && T[i]->node_2->tag >
T[i+1]->node_2->tag) {
            swap(T[i],T[i+1]);
         }
    }

    for(int i = 0; i < T_top; i++) {
        printf("%d--%d\n", T[i]->node_1->tag, T[i]->node_2->tag);
    }


    for(int i = 0; i < vSize ; i++) {
        free(nodes[i]);
    }
    for(int i = 0; i < eSize ; i++) {
        DesEdge(edges[i]);
    }
    free(nodes);
    free(edges);
}
```

3. Union Find

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

#ifndef _UNION_FIND_H_
#define _UNION_FIND_H_

typedef struct node_t {
    struct node_t * parent;
    int tag;
```

```
    int rank;
} node;

//GenSet initialize one node
node * GenSet(node * in_node) {
    in_node = (node *)malloc(sizeof(node));
    in_node->parent = in_node;
    in_node->rank = 0;
    return in_node;
}

//Find return the root node of the node
node * Find(node * in_node) {
    if(in_node->parent != in_node) {
        in_node->parent = Find(in_node->parent);
    }
    return in_node->parent;
}


//Union glue the root node of x and y together
void Union(node * x, node * y) {
    node * x_root = Find(x);
    node * y_root = Find(y);
    if(x_root->rank > y_root->rank) {
        y_root->parent = x_root;
    }
    else {
        x_root->parent = y_root;
    }
    if(x_root->rank == y_root->rank) {
        y_root->rank++;
    }
}

typedef struct edge_t {
    node * node_1;
    node * node_2;
    int weight;
} edge;

//constructor
edge * GenEdge(edge * e) {
    e = (edge *)malloc(sizeof(edge));
    e->node_1 = GenSet(e->node_1);
    e->node_2 = GenSet(e->node_2);
    e->weight = 0;
    return e;
}

//destructor
void DesEdge(edge * e) {
    free(e->node_1);
    free(e->node_2);
    free(e);
}

void swap(edge * x, edge * y) {
```

```
    edge temp = * x;
    * x = * y;
    * y = temp;
}

//input the edges array and the left point, right point, which initially are 0
and size - 1, sort the array in non-decreasing order
edge ** sort_edge (edge ** arr, int left, int right) {
    //choose the first element as pivot
    int pivotat = 0;

    //base case
    if(left >= right) {
        return arr;
    }

    // two counters from first and last
    int i = left;
    int j = right;
    while(1) {

        //iterate for i until find the first element with weight greater than
weight of arr[0],
        //or i > j, since all elements righter than j are bigger than pivot, we
end
        while (1) {
            if( i > j || arr[i]->weight > arr[left]->weight ) {
                break;
            }
            else {
                i++;
            }
        }

        while(1) {
            if( j < i || arr[j]->weight < arr[left]->weight ) {
                break;
            }
            else {
                j--;
            }
        }

        if(i < j) {
            swap (arr[i],arr[j]);
        }
        //else we finish the searching and quit the loop
        else{
            swap(arr[j],arr[left]);
            pivotat = j;
            break;
        }
    }
    arr = sort_edge(arr,left, pivotat-1);
    arr = sort_edge(arr,pivotat+1,right);
    return arr;
}
```

```
//input the edges array and the left point, right point, which initially are 0
and size - 1, sort the array in non-decreasing order
edge ** sort_edge_by_node2 (edge ** arr, int left, int right) {
    //choose the first element as pivot
    int pivotat = 0;

    //base case
    if(left >= right) {
        return arr;
    }

    // two counters from first and last
    int i = left;
    int j = right;
    while(1) {

        //iterate for i until find the first element with weight greater than
weight of arr[0],
        //or i > j, since all elements righter than j are bigger than pivot, we
end
        while (1) {
            if( i > j || arr[i]->node_2->tag > arr[left]->node_2->tag ) {
                break;
            }
            else {
                i++;
            }
        }

        while(1) {
            if( j < i || arr[j]->node_2->tag < arr[left]->node_2->tag ) {
                break;
            }
            else {
                j--;
            }
        }

        if(i < j) {
            swap (arr[i],arr[j]);
        }
        //else we finish the searching and quit the loop
        else{
            swap(arr[j],arr[left]);
            pivotat = j;
            break;
        }
    }
    arr = sort_edge_by_node2(arr,left, pivotat-1);
    arr = sort_edge_by_node2(arr,pivotat+1,right);
    return arr;
}

//input the edges array and the left point, right point, which initially are 0
and size - 1, sort the array in non-decreasing order
edge ** sort_edge_by_node1 (edge ** arr, int left, int right) {
    //choose the first element as pivot
    int pivotat = 0;
```

```
    //base case
    if(left >= right) {
        return arr;
    }

    // two counters from first and last
    int i = left;
    int j = right;
    while(1) {

        //iterate for i until find the first element with weight greater than
weight of arr[0],
        //or i > j, since all elements righter than j are bigger than pivot, we
end
        while (1) {
            if( i > j || arr[i]->node_1->tag > arr[left]->node_1->tag ) {
                break;
            }
            else {
                i++;
            }
        }

        while(1) {
            if( j < i || arr[j]->node_1->tag < arr[left]->node_1->tag ) {
                break;
            }
            else {
                j--;
            }
        }

        if(i < j) {
            swap (arr[i],arr[j]);
        }
        //else we finish the searching and quit the loop
        else{
            swap(arr[j],arr[left]);
            pivotat = j;
            break;
        }
    }
    arr = sort_edge_by_node1(arr,left, pivotat-1);
    arr = sort_edge_by_node1(arr,pivotat+1,right);
    return arr;
}

int * cut(char * str) {
    int * result = (int *)malloc(3 * sizeof(int));
    for (int i = 0; i < 3; i++) {
        char * numbers = strtok(str, " ");
        result [i] = atoi(numbers);
    }
    return result;
}
```

```
#endif
```