VE477 Introduction to Algorithms

Lab 1

Taoyue Xia, 518370910087

2021/09/28

1. C programming

- The Union-Find data structure

```
int Find(Vertex *v, int i) {
    if(v[i].parent != i) {    //v is the array of all the vertexes
        return Find(v, v[i].parent);
    }
    return v[i].parent;
}

// The Union method is implemented according to the pseudocode on slide
void Union(Vertex *v, int v1, int v2) {
    int v1Root = Find(v, v1);
    int v2Root = Find(v, v2);

    if(v[v1Root].rank > v[v2Root].rank) v[v2Root].parent = v1Root;
    else v[v1Root].parent = v2Root;

    if(v[v1Root].rank == v[v2Root].rank) v[v2Root].rank ++;
}
```

- Kruskal's Algorithm

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

typedef struct vertex_t {
    int parent;
    int rank;
} Vertex;

typedef struct edge_t {
    int v1;
    int v2;
    int weight;
} Edge;

typedef struct graph_t {
    int EdgeNum;
    int vertexNum;
    Edge *edges;
} Graph;

void insertEdge(Graph *g, int eSize) {
    for(int i = 0; i < eSize; i++) {
        int v1, v2, weight;
        scanf("%d %d %d", &v1, &v2, &weight);
        if(v1 > v2) {
            int temp = v1;
            v1 = v2;
            v2 = temp;
```

```c
        }
        g->edges[i].v1 = v1;
        g->edges[i].v2 = v2;
        g->edges[i].weight = weight;
    }
}

Graph *initGraph(int eSize, int vSize) {
    Graph *g = (Graph *) malloc(sizeof(Graph));
    g->vertexNum = vSize;
    g->EdgeNum = eSize;
    g->edges = (Edge *) malloc(sizeof(Edge) * eSize);

    insertEdge(g, eSize);
    return g;
}

Vertex *initVertexes(int vSize) {
    Vertex *v = (Vertex *) malloc(sizeof(Vertex) * vSize);
    for(int i = 0; i < vSize; i++) {
        v[i].parent = i;
        v[i].rank = 0;
    }
    return v;
}

// According to the pseudocode in lecture c1
int Find(Vertex *v, int i) {
    if(v[i].parent != i) {
        return Find(v, v[i].parent);
    }
    return v[i].parent;
}

void Union(Vertex *v, int v1, int v2) {
    int v1Root = Find(v, v1);
    int v2Root = Find(v, v2);

    if(v[v1Root].rank > v[v2Root].rank) v[v2Root].parent = v1Root;
    else v[v1Root].parent = v2Root;

    if(v[v1Root].rank == v[v2Root].rank) v[v2Root].rank ++;
}

int EdgeCompare(const void *x, const void *y) {
    Edge *e1 = (Edge *) x;
    Edge *e2 = (Edge *) y;

    if(e1->weight < e2->weight) return -1;
    else if(e1->weight == e2->weight) return 0;
    else return 1;
}

int VertexCompare(const void *x, const void *y) {
    Edge *e1 = (Edge *) x;
    Edge *e2 = (Edge *) y;

    if(e1->v1 < e2->v1) return -1;
```

```c
    else if(e1->v1 > e2->v1) return 1;
    else {
        if(e1->v2 < e2->v2) return -1;
        else return 1;
    }
}

void kruskal(Graph *g) {
    qsort(g->edges, g->EdgeNum, sizeof(Edge), EdgeCompare);
    Edge *MST = (Edge *) malloc(sizeof(Edge) * (g->vertexNum - 1));
    Vertex *v = initVertexes(g->vertexNum);

    int i = 0;
    int j = 0;

    while(i < g->EdgeNum && j < g->vertexNum - 1) {
        Edge temp = g->edges[i];
        i++;

        int v1Root = Find(v, temp.v1);
        int v2Root = Find(v, temp.v2);
        if(v1Root != v2Root) {
            MST[j] = temp;
            j++;
            Union(v, v1Root, v2Root);
        }
    }

    free(v);
    qsort(MST, j, sizeof(Edge), VertexCompare);

    for(int k = 0; k < j; k++) {
        printf("%d--%d\n", MST[k].v1, MST[k].v2);
    }
    free(MST);
}

int main() {
    int eSize, vSize;
    scanf("%d", &eSize);
    scanf("%d", &vSize);

    Graph *graph = initGraph(eSize, vSize);
    kruskal(graph);
    free(graph->edges);
    free(graph);

    return 0;
}
```

The time complexity of creating a new graph is $\mathcal{O}(1)$.

The time complexity of initializing vertices is $\mathcal{O}(V)$, where V is the number of vertices.

The time complexity of sorting the edges by weight is $\mathcal{O}(E \log E)$, where E is the number of edges.

Finally, the time complexity of adding edges to MST with Union-Find is $\mathcal{O}(V + E)\alpha(V)$.

In conclusion, the time complexity is $\mathcal{O}(E \log E)$.

- Prim's Algorithm for MST

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

typedef struct edge_t {
    int v1;
    int v2;
} Edge;

typedef struct graph_t {
    int vSize;
    int eSize;
    int **adj;
} Graph;

void insertEdges(Graph *g, int eSize) {
    for(int i = 0; i < eSize; i++) {
        int v1, v2, weight;
        scanf("%d %d %d", &v1, &v2, &weight);

        g->adj[v1][v2] = weight;
        g->adj[v2][v1] = weight;
    }
}

Graph *initGraph(int vSize, int eSize) {
    Graph *g = (Graph *) malloc(sizeof(Graph));
    g->vSize = vSize;
    g->eSize = eSize;
    g->adj = (int **) malloc(sizeof(int *) * vSize);
    for(int i = 0; i < vSize; i++) {
        g->adj[i] = (int *) malloc(sizeof(int) * vSize);
    }
    for(int i = 0; i < vSize; i++) {
        for (int j = 0; j < vSize; j++) {
            if(i == j) g->adj[i][j] = 0;
            else {
                g->adj[i][j] = INT_MAX;
            }
        }
    }

    insertEdges(g, eSize);
    return g;
}

int EdgeCompare(const void *x, const void *y) {
    Edge *former = (Edge *) x;
    Edge *latter = (Edge *) y;

    if(former->v1 < latter->v1) return -1;
    else if(former->v1 > latter->v1) return 1;
```

```
    else {
        if(former->v2 < latter->v2) return -1;
        else return 1;
    }
}

void prim(Graph *g) {
    int lowCost[g->vSize];
    int mst[g->vSize];
    Edge *edges = (Edge *) malloc(sizeof(Edge) * g->vSize);
    mst[0] = -1;

    for(int i = 1; i < g->vSize; i++) {
        lowCost[i] = INT_MAX;
        mst[i] = 0;
    }
    lowCost[0] = INT_MAX;

    for(int i = 0; i < g->vSize; i++) {
        int min = INT_MAX;
        int minVertex = 0;

        for(int j = 0; j < g->vSize; j++) {
            if(lowCost[j] < min && lowCost[j] != 0) {
                min = lowCost[j];
                minVertex = j;
            }
        }
        edges[i].v1 = minVertex;
        edges[i].v2 = mst[minVertex];

        lowCost[minVertex] = 0;
        for(int j = 0; j < g->vSize; j++) {
            if(g->adj[minVertex][j] < lowCost[j]) {
                lowCost[j] = g->adj[minVertex][j];
                mst[j] = minVertex;
            }
        }
    }

    for(int i = 0; i < g->vSize; i++) {
        if(edges[i].v1 > edges[i].v2) {
            int temp = edges[i].v1;
            edges[i].v1 = edges[i].v2;
            edges[i].v2 = temp;
        }
    }

    qsort(edges, g->vSize, sizeof(Edge), EdgeCompare);
    for(int i = 1; i < g->vSize; i++) {
        printf("%d--%d\n", edges[i].v1, edges[i].v2);
    }
    free(edges);
}

int main() {
    int vSize, eSize;
    scanf("%d", &eSize);
```

```
    scanf("%d", &vSize);
    Graph *graph = initGraph(vSize, eSize);
    prim(graph);
    for(int i = 0; i < vSize; i++) {
        free(graph->adj[i]);
    }
    free(graph->adj);
    free(graph);
}
```

The time complexity of creating a new graph is $\mathcal{O}(1)$.

The time complexity of generating vertices is $\mathcal{O}(V)$.

The time complexity of choosing the proper MST is $\mathcal{O}(V^2)$.

The sort time complexity is $\mathcal{O}(V \log V)$.

Therefore, the whole time complexity is $\mathcal{O}(V^2)$.

- The performance of **Kruskal's Algorithm** and **Prim's Algorithm** depends on the type of graphs. For sparse graphs, Kruskal's Algorithm will have a better performance due to the low time complexity. For dense graphs, Prim's Algorithm will be better since its trunning time is independent of the total edge number.

2 Getting started with OCaml

2.1 More documentation

- Anonymous functions can be defined using the **fun** or **function** construct, for example, (fun x -> x * x).

  Anonymous functions are values and are used to be passed to other functions as inputs.

- Capital letters cannot be used for variables and functions, underscore is used instead.

- In OCaml, every piece of code is wrapped into a module, and a module can be submodule of another module. When you create a file, for example, named "amodule.ml", it will automatically define a module named "Amodule".

  Different from OOP, we don't need to create a new instance to use the methods and variables.

- **List** is the array type defined in OCaml. In package **Lists**, many operation methods on arrays are defined, such as **"List.partition"** and **"List.map"**.

- **List.map** and **List.iter** both can iterate through a list. However, **List.map** needs anonymous function to pass into it, and it will return a new list, while **List.iter** is like the usage of a for loop.

  List.map can map the input to an output according to the anonymous function, and **List.iter** is a more efficient way to iterate comparing with for and while loops.

- Folding can apply the defined anonymous function to elements in the list, and return the accumulated result.

  For example, **List.fold_left (+) 0 [1; 3; 5; 7]**;;, which will add 1 to 0, and then add 3 to the previous result, until 7 is added, which will give a result of $1 + 3 + 5 + 7 = 16$.

- Tail recursive functions can be optimized by compilers from recursive functions to while loops, if the recursion call is at the end of the function definition, and they run in constant stack space, avoiding stack overflow, which is important.

- **ref** means creating a reference and turn it into a pointer. **ref** is used when you want to make data mutable.

- A functor is a module that is parametrized by another module. It can define the data type of the new created module, like the template in C++ does. However, functors can refine signatures of modules, but templates are limited due to private and public attributes.

- **type t = type1 | type2 | type3 ... | type n**, or

  type t = {**type1** : **t1**; **type2** : **t2**; ...; **typen** : **tn**}

- Sum types generalize what is often known as enum, union, or variant record. Product types are like tuples, products of types.

  Sum types can be used to model finite sets, which is like enum in C. Product types are used when multiple properties exist simultaneously.

2.2 Coding

- The Ocaml code is shown below:

  ```
  let s = read_line();;

  let list = List.map int_of_string (Str.split (Str.regexp ", ") s);;

  let rec quicksort = function
    | [] -> []
    | head::tail -> let smaller, larger = List.partition ((>) head) tail in
  quicksort smaller @ (head::quicksort larger)
  ;;

  let ll = quicksort list;;

  List.iter (Printf.printf "%d ") ll;;
  ```

  Firstly, take the standard input as a string and save it to **s**. Then use `Str.split` and `Str.regexp` to split `s` into multiple strings of integers, and use `List.map` with `int_of_string` to transform into the list of integers.

  After that, the function `quicksort` is defined as a recursive function. If the input is empty list, it returns an empty list. If not, the list will be parsed into two lists using `List.partition`, one contains all the integers smaller or equal to the pivot `head`, the other with all the larger integers. Then, quicksort the two split lists until empty.

  Finally, assign the returned list of `quicksort` to `ll` and use `List.iter` to print all the integers in the sorted list.

- The average time complexity is $\mathcal{O}(n \log n)$.