Assignment 1

Name: Shiyin Lin
UFID: 49619012

1 Cycle Finding

1.1 Pseudo-code

algorithm 1 Disjoint-set data structure

Input: *Array* Edges
Output: bool

```
function DSU(Array)
    if x is not already in the forest then
        x.patent := x
    end if
    for each edge in Edges do
        pf :=Find(edge.fr)
        pt :=Find(edge.to)
        if pf = pt then
            Find cycle
            return true
        end if
        y.parent := x
    end for
    No cycle
    return false
end function

function Find(x) is
    while x.parent ≠ x do
        (x, x.parent) := (x.parent, x.parent.parent)
    end while
    return x
end function
```

1.2 Proof of Correctness

Assume the edge's node x and node y are in the set but cannot form a cycle.
node x and node y are in the set
$\Rightarrow$node x is connected to node y
node x and node y cannot form a cycle
$\Rightarrow$No edge contains node x and node y
$\Rightarrow$contradiction
$\Rightarrow$node x and node y in the set can form a cycle

1.3 Algorithm's running time

When initializing the parent array, the loop is n times, and the complexity is $\Theta(n)$. If the lookup path contains s nodes, it is obvious that the time complexity of the lookup is $\Theta(s)$. The time complexity of the find(a) and union(a) operation can be proven to be $\Theta(\alpha(n))$ if no node's potential energy increases due to optimization using path compression and at least $s - \alpha(n)$ nodes have their potential energy reduced by at least 1. So the total cost is $\Theta(n) + \Theta(\alpha(n))$

1.4 Implement

1.4.1 Graph Generator

```
func generateUndirected(nnum int, enum int) graph.Undirected {
    r := rand.New(rand.NewSource(time.Now().Unix()))
    return graph.GnmUnidirected(nnum, enum, r)
}

func generateEdges(g graph.Undirected) [][]int {
    edgeCollect := [][]int
    for fr, to := range g.AdjacencyList {
        for _, to := range to {
            if graph.NI(fr) < to {
                tmp := []int{}
                tmp = append(tmp, fr)
                tmp = append(tmp, int(to))
                edgeCollect = append(edgeCollect, tmp)
            }
        }
    }
    return edgeCollect
}
```

1.4.2 Test Code

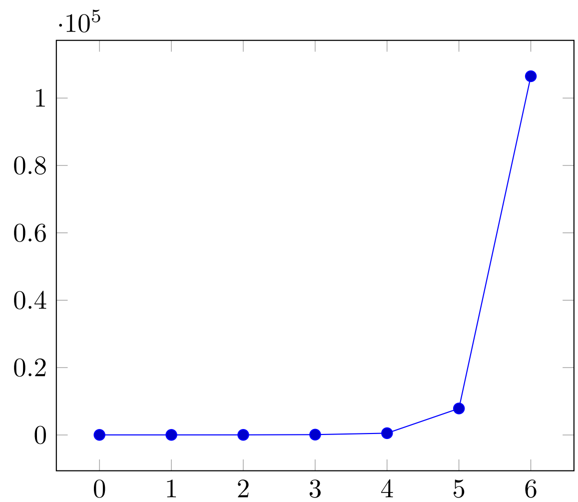
```
func Test() {
    nodes := 10000000
    edges := 20000000
    g := generateUndirected(nodes, edges)
    edgeCollect := generateEdges(g)
```

```
    start := time.Now()
    FindCycleDSU(edgeCollect)
    elapsed := time.Since(start)
    fmt.Println("total cost:", elapsed)
}
```

1.4.3 Test algorithm for increasing graph size

Node	Edge	t1	t2	t3	ave
10	20	25.025μs	25.826μs	26.6μs	25.817μs
100	200	24.165μs	26.877μs	27.149μs	26.064μs
1000	2000	34.612μs	32.773μs	33.821μs	33.735μs
10000	20000	97.649μs	111.776μs	98.966 μs	102.797μs
100000	200000	508.58μs	452.34μs	629.028μs	529.983μs
1000000	2000000	7.098ms	8.127ms	8.371ms	7.865ms
10000000	20000000	114.545ms	109.498ms	95.398ms	106.48ms

1.4.4 Plot running time



2 Minimum Spanning Tree for sparse graphs

2.1 Pseudo-code

algorithm 2 Kruskal

Input: *Int* numOfNodes, *Int* numOfEdges, *Array* EdgesWithWright
Output: *Null*

```
function Kruskal(Int, Int, Array)
    if node is not already in the forest then
        node.patent := node
    end if
    QuicksSore(Edges)
    while i < numOfEdges do
        pf :=Find(edge[i].fr)
        pt :=Find(edge[i].to)
        if pf ≠ pt then
            pt.parent = pf
            resault+ = edge.weight
        end if
    end while
    if not result then
        Can't find MST
    else
        MST is : result
    end if
end function
```

2.2 Proof of Correct

Suppose that the graph G has n vertices, then the spanning tree must have $n-1$ edges. Since the number of spanning trees of a graph is finite, there is at least one tree with minimum cost, which is assumed to be U. First make the following assumptions.

1) The output tree is T.

2) The number of different edges in U and T is k, and the other $n-1-k$ edges are the same, and these $n-1-k$ edges form the edge set E.

3) The edges in T but not in U are $a_1, a_2, ...$ in order of cost from smallest to largest. , a_k.

4) The edges in U but not in T are $x_1, x_2, ...$ in order of cost from smallest to largest. , x_k.

By converting U to T (moving the edges of T into U in order), to prove that U and T have the same cost.

First, we move a_1 into U. Since U itself is a tree, adding any edge at this point constitutes a cycle, so the addition of a_1 must produce a cycle, and this cycle must include the edges in $x_1, x_2, ..., x_k., x_k$. (Otherwise a_1 and the edges in E form a cycle, and E is also in T, which contradicts the absence of a cycle in T.)

In this cycle delete edges belonging to $x_1, x_2, ..., x_k$ and the most costly edge x_i forms a new spanning tree V.

Assuming that a_1 cost is less than x_i, the cost of V is less than U. This contradicts that U is a minimum cost tree, so a_1 cannot be less than x_i.
Assuming that a_1 is greater than x_i, according to Kruskal's algorithm, the edge with small cost is considered first, then when Kruskal's algorithm is executed, x_i should be considered before a_1, which in turn is considered before $a_2, ..., a_k$ before considering x_i, so before considering x_i, the edges in T can only be edges in E. And since x_i did not join the tree T, it means that x_i must constitute a cycle with some edges in E. But x_i and E are in U at the same time, which contradicts with U being a spanning tree, so a_1 cannot be larger than x_i either.

Therefore, the newly obtained tree V has the same cost as T.

By analogy, adding the edges of $a_1, a_2, ...,$ the edges of a_k are gradually added to U, and the final obtained tree T has the same cost as U.

2.3 Algorithm's running time

Since the algorithm is optimized using the DSU of path compression, the complexity is $\Theta(n) + \Theta(\alpha(n))$, and since it contains a quick sort operation, the consultation complexity is $\Theta(n) + \Theta(\alpha(n)) + \Theta(mlogm)$

2.4 Implement

2.4.1 Graph Generator

```
func generateConnectGraph(nnum int, enum int) graph.Undirected {
    n, _ := rand2.Int(rand2.Reader, big.NewInt(65536))
  r := rand.New(rand.NewSource(n.Int64()))
  g := graph.GnmUndirected(nnum, enum, r)
  if !g.IsConnected() {
    generateConnectGraph(nnum, enum)
  }
  return g
}

func generateWEdges(g graph.Undirected, enum int) [][]int {
```

```
  weight := make([]int, enum)

  for i := 0; i < enum; i++ {
    n, _ := rand2.Int(rand2.Reader, big.NewInt(int64(20)))
    if n.Int64() == 0 {
      weight[i] = int(n.Int64()) + 1
    } else {
      weight[i] = int(n.Int64())
    }
  }
  edgeCollect := [][]int{}
  for fr, to := range g.AdjacencyList {
    for _, to := range to {
      if graph.NI(fr) < to {
        tmp := []int{}
        tmp = append(tmp, fr)
        tmp = append(tmp, int(to))
        edgeCollect = append(edgeCollect, tmp)
      }
    }
  }
  for i:=0; i<enum; i++ {
    edgeCollect[i] = append(edgeCollect[i], weight[i])
  }
  return edgeCollect
}
```

2.4.2 Test Code

```
func Test2() {
  nodes := 10
  edges := 18
  if edges < nodes-1 {
    fmt.Println("too less edges")
    os.Exit(-1)
  }
  g := generateConnectGraph(nodes, edges)
  edgeCollect := generateWEdges(g, edges)
  fmt.Println(edgeCollect)
  sort.Slice(edgeCollect[:edges], func(i,j int) bool {
    return edgeCollect[i][2] < edgeCollect[j][2]
  })
  fmt.Println(edgeCollect)
  start := time.Now()
  kruskal(nodes, edges, edgeCollect)
  elapsed := time.Since(start)
  fmt.Println("total cost:", elapsed)
}
```

2.4.3 Test algorithm for increasing graph size

Node	Edge	t1	t2	t3	ave
10	18	4.048μs	3.732μs	5.301μs	4.36μs
20	28	5.059μs	5.263μs	5.503μs	5.275μs
30	38	5.912μs	6.125μs	5.426μs	5.821μs
40	48	5.534μs	6.11μs	6.466μs	6.037μs
50	58	9.591μs	9.011μs	9.313μs	9.305μs
60	68	9.369μs	9.515μs	9.561μs	9.482μs
70	78	10.076μs	11.741μs	12.044μs	11.287μs

2.4.4 Plot running time