106119029 , Algos Lab 9

Dipesh Kafle

Code

```cpp
#include <algorithm>
#include <chrono>
#include <fstream>
#include <iostream>
#include <numeric>
#include <queue>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>

template <typename Func, typename... Args>
double timeMyFunction(Func func, Args &&...args) {
  auto start_time = std::chrono::steady_clock::now();
  func(std::forward<Args>(args)...);
  auto end_time = std::chrono::steady_clock::now();
  std::chrono::duration<double> elapsed_time =
      std::chrono::duration_cast<std::chrono::duration<double>>(end_time -
                                                                start_time);
  return elapsed_time.count();
}
```

```cpp

using namespace std;

struct node {
  int v;
  int weight;
  node(int v, int w) : v(v), weight(w) {}
  node(const node &other) : v(other.v), weight(other.weight) {}
  node &operator=(const node &other) {
    v = other.v;
    weight = other.weight;
    return *this;
  }
  bool operator<(const node &other) const { return weight < other.weight; }
  bool operator>(const node &other) const { return weight > other.weight; }
  bool operator==(const node &other) const {
    return weight == other.weight && v == other.v;
  }
  bool operator<=(const node &other) const {
    return *this < other || *this == other;
  }
  bool operator>=(const node &other) const {
    return *this > other || *this == other;
  }
};
```

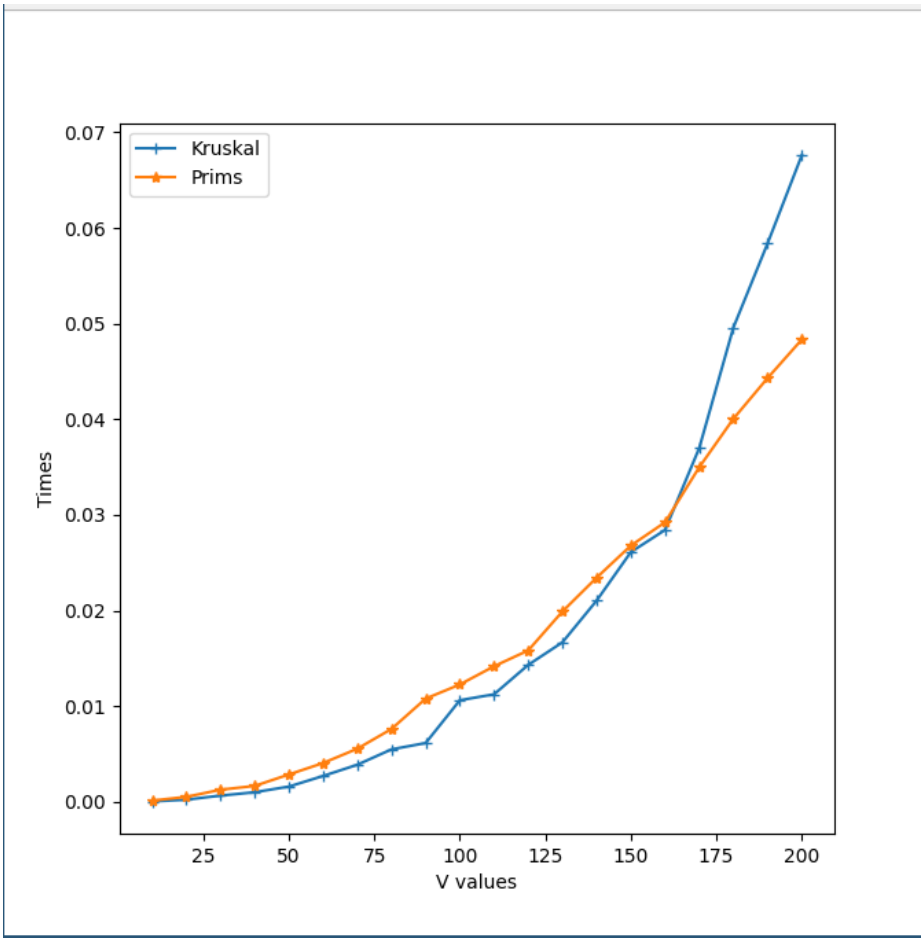
```cpp
class Graph {
public:
  int V;
  vector<vector<node>> adj;
  Graph(int v) : V(v) { adj.assign(v, vector<node>()); }
  // function to add an edge to graph
  void addEdge(int u, int v, int w) {
    adj[u].push_back(node(v, w));
    adj[v].push_back(node(u, w));
  }

  auto primMST() {
    priority_queue<node, vector<node>, std::greater<node>> pq;
    int src = 0;
    unordered_set<int> inMST;
    vector<int> key(V, numeric_limits<int>::max());
    vector<int> parent(V, -1);
    pq.push(node(src, 0)); // src is v and 0 is wt
    key[src] = 0;
    while (!pq.empty()) {
      int u = pq.top().v;
      pq.pop();
      inMST.insert(u);
      for (auto &nd : adj[u]) {
        if (inMST.find(nd.v) == inMST.end() && key[nd.v] > nd.weight) {
          key[nd.v] = nd.weight;
          pq.push(node(nd.v, key[nd.v]));
          parent[nd.v] = u;
        }
      }
    }
    // for (int i : inMST)
    //   printf("%d - %d\n", parent[i], i);
    return make_pair(std::move(inMST), std::move(parent));
  }
};

using edge = pair<int, int>;

vector<pair<int, edge>> convertToEdgeList(const Graph &G) {
  // creates an undirected edgelist
  vector<pair<int, edge>> lst;
  for (int i = 0; i < G.V; i++) {
    for (auto &y : G.adj[i]) {
      if (y.v > i) {
        lst.push_back({y.weight, {i, y.v}});
      }
    }
  }
  return lst;
}

int find_set(int i, const vector<int> &parent) {
  if (i == parent[i])
    return i;
  return find_set(parent[i], parent);
}

void union_set(int u, int v, vector<int> &parent) { parent[u] = parent[v]; }
```

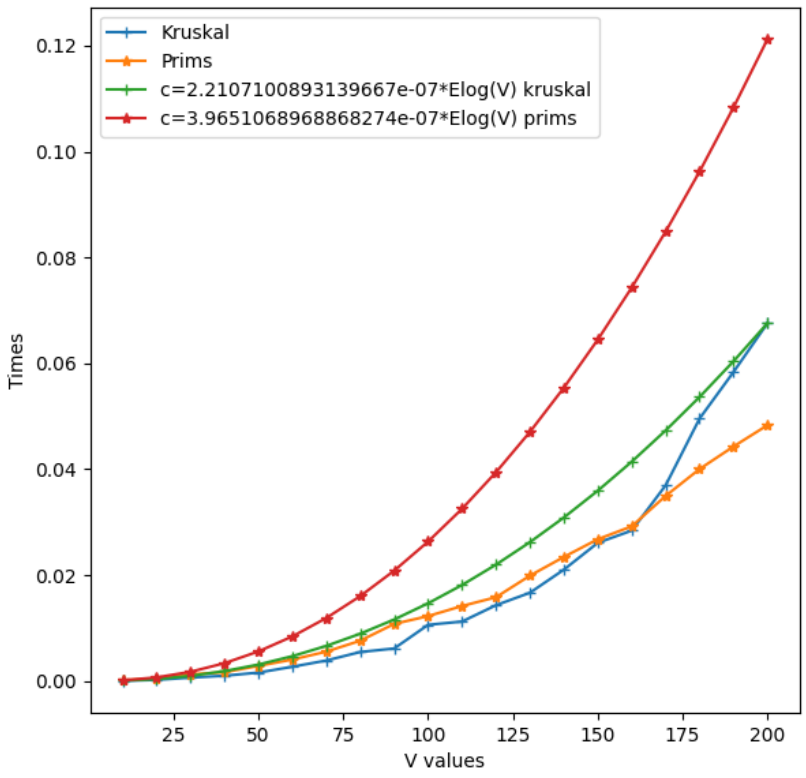
```cpp
auto kruskal(vector<pair<int, edge>> &edgelist, int V) {
  vector<pair<int, edge>> T;
  vector<int> parent(V);
  iota(parent.begin(), parent.end(), 0); // parent = 0,1,2,3,4..V
  int i, u_representative, v_representative;
  sort(edgelist.begin(), edgelist.end());
  for (i = 0; i < edgelist.size(); i++) {
    u_representative = find_set(edgelist[i].second.first, parent);
    v_representative = find_set(edgelist[i].second.second, parent);
    if (u_representative != v_representative) {
      T.push_back(edgelist[i]);
      union_set(u_representative, v_representative, parent);
    }
  }
  return T;

  //   prints
  //   for (auto &v : T) {
  //     cout << v.second.first << '-' << v.second.second << '\n';
  //   }
}
```

```cpp
int main() {
  int V = 10;
  ofstream krus("kruskal.txt");
  ofstream prim("prim.txt");
  for (int i = 0; V <= 200; i++) {
    Graph g(V);
    for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
        if (i != j) {
          g.addEdge(i, j, rand() % 100);
        }
      }
    }

    auto graph = convertToEdgeList(g);
    double elapsed_time = timeMyFunction(kruskal, graph, V);
    krus << V << ':' << elapsed_time << '\n';

    elapsed_time = timeMyFunction([&g] { return g.primMST(); });
    prim << V << ':' << elapsed_time << '\n';
    V += 10;
  }
}
```

Plots

- I have generated complete graphs of multiple sizes and ran prims and kruskal on them.
- They both have complexity of O((E+V)log(V)) when used with suitable data structures. Since E is O(V^2), We can say it is O($V^2 log(V)$)
- Since both the algorithm do not work for directed graph, we cannot compare their performance in that area.


Kruskal
Prims
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
Times
25
50
75
100
125
150
175
200
V values


Kruskal
Prims
c=2.2107100893139667e-07*Elog(V) kruskal
c=3.9651068968868274e-07*Elog(V) prims
Times
V values
0.00
0.02
0.04
0.06
0.08
0.10
0.12
25
50
75
100
125
150
175
200