106119029 , Algos Lab 10

Dipesh Kafle

1) Code for Bellman Ford and Dijkstra Single Source shortest path

```
#include <algorithm>
#include <chrono>
#include <fstream>
#include <functional>
#include <iomanip>
#include <iostream>
#include <iterator>
#include <numeric>
#include <queue>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <utility>
#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();
}

using namespace std;

/*
 *
 * DIJKSTRA
 *
 *
```

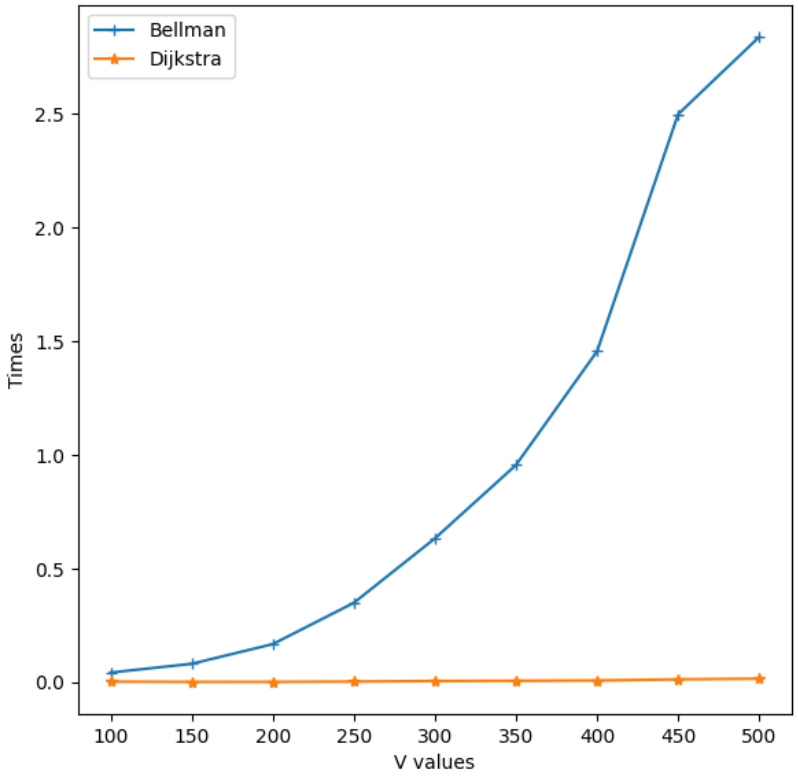
```
 */
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;
void dijkstra(int s, vector<int> &distance, vector<int> &p) {
  int n = adj.size();
  distance.assign(n, INF);
  p.assign(n, -1);
  vector<bool> u(n, false);
  distance[s] = 0;
  for (int i = 0; i < n; i++) {
    int v = -1;
    for (int j = 0; j < n; j++) {
      if (!u[j] && (v == -1 || distance[j] < distance[v]))
        v = j;
    }
    if (distance[v] == INF)
      break;
    u[v] = true;
    for (auto edge : adj[v]) {
      int to = edge.first;
      int len = edge.second;

      if (distance[v] + len < distance[to]) {
        distance[to] = distance[v] + len;
        p[to] = v;
      }
    }
  }
  cout << "Dijkstra : ";
  copy(distance.begin(), distance.end(), ostream_iterator<int>(std::cout, " "));
  cout << '\n';
}

/*
 *
 * BELLMAN FORD
 *
 *
 */
struct edge {
  int a, b, cost;
};

// n is number of vertices
// m is number of edges
int n, m;
```

```
vector<edge> edgelist;
void bellman_ford(int start) {
  vector<int> distance(n, INF);
  distance[start] = 0;
  for (int i = 0; i < n - 1; ++i) {
    for (int j = 0; j < m; ++j) {
      if (distance[edgelist[j].a] < INF) {
        distance[edgelist[j].b] =
            min(distance[edgelist[j].b],
                distance[edgelist[j].a] + edgelist[j].cost);
      }
    }
  }
  cout << "Bellman  : ";
  copy(distance.begin(), distance.end(), ostream_iterator<int>(std::cout, " "));
  cout << '\n';
}

void random_complete_graph(int size) {
  srand(time(NULL));
  adj.clear();
  edgelist.clear();
  n = size;
  adj.assign(n, vector<pair<int, int>>());
  for (int i = 0; i < size; i++) {
    for (int j = i + 1; j < size; j++) {
      int wt = rand() % 100;
      adj[i].push_back({j, wt});
      edgelist.push_back({i, j, wt});
    }
  }
  m = edgelist.size();
}

int main() {
  ofstream dijkstra_file("Dijkstra.txt");
  ofstream bellman_file("Bellman.txt");
  for (int i = 100; i <= 500; i += 50) {
    random_complete_graph(i);
    vector<int> parent, distance;
    cout << i << '\n';
    auto timeElapsed =
        timeMyFunction(dijkstra, 0, std::ref(distance), std::ref(parent));
    dijkstra_file << fixed << setprecision(30) << n << ':' << m << ':'
                  << timeElapsed << '\n';
    timeElapsed = timeMyFunction(bellman_ford, 0);
```

```
    bellman_file << fixed << setprecision(30) << n << ':' << m << ':'
                 << timeElapsed << '\n';
    cout << "\n\n";
  }
}
```

2) Compare their performance for connected and disconnected graphs.

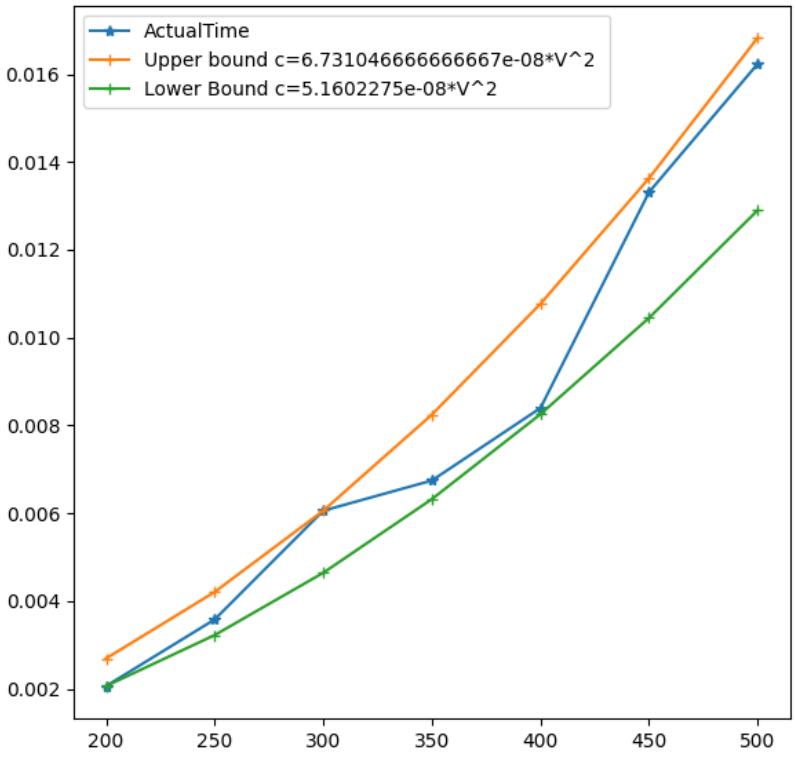
- We have implemented a O(V^2) Dijkstra and O(V^3) Bellman Ford algorithm.



3) Plot the time taken by them with same random graphs as input and tally with theoretical analysis.

- We have two loops , one inside another both running from 0 to n (n is V), with constant work inside. So trivially, the time complexity is O(V^2).
- We have outer loop going from 0 to n(V) and the inner loop going from 0 to m(E), with O(1) work going on inside. So trivially, its complexity is O(VE)

Dijkstra



Bellman Ford