Notebook - Maratona de Programação

Cabo HDMI, VGA, USB

Contents

1 Graph

1.1 Dinic

Complexity: O(V*V*E), Bipartite is O(sqrt(V) E)
Dinic is a strongly polynomial maximum flow algorithm, doesnt depend on capacity values good for matching

```cpp
// TITLE: Dinic
// COMPLEXITY: O(V*V*E), Bipartite is O(sqrt(V) E)
// DESCRIPTION: Dinic is a strongly polynomial maximum flow algorithm, doesnt depend on capacity values good for matching

const int oo = 0x3f3f3f3f3f3f3f3f;
// Edge structure
struct Edge
{
    int from, to;
    int flow, capacity;

    Edge(int from_, int to_, int flow_, int capacity_)
        : from(from_), to(to_), flow(flow_), capacity(capacity_)
    {}
};

struct Dinic
{
    vector<vector<int>> graph;
    vector<Edge> edges;
    vector<int> level;
    int size;

    Dinic(int n)
    {
        graph.resize(n);
        level.resize(n);
        size = n;
        edges.clear();
    }

    void add_edge(int from, int to, int capacity)
    {
        edges.emplace_back(from, to, 0, capacity);
        graph[from].push_back(edges.size() - 1);

        edges.emplace_back(to, from, 0, 0);
        graph[to].push_back(edges.size() - 1);
    }

    int get_max_flow(int source, int sink)
    {
        int max_flow = 0;
        vector<int> next(size);
        while(bfs(source, sink)) {
            next.assign(size, 0);
            for (int f = dfs(source, sink, next, oo); f != 0; f = dfs(source, sink, next, oo)) {
                max_flow += f;
            }
        }
        return max_flow;
    }

    bool bfs(int source, int sink)
    {
        level.assign(size, -1);
        queue<int> q;
        q.push(source);
        level[source] = 0;

        while(!q.empty()) {
            int a = q.front();
            q.pop();

            for (int & b: graph[a]) {
                auto edge = edges[b];
                int cap = edge.capacity - edge.flow;
                if (cap > 0 && level[edge.to] == -1) {
                    level[edge.to] = level[a] + 1;
                    q.push(edge.to);
                }
            }
        }
        return level[sink] != -1;
    }

    int dfs(int curr, int sink, vector<int> & next, int flow)
    {
        if (curr == sink) return flow;
        int num_edges = graph[curr].size();

        for (; next[curr] < num_edges; next[curr]++) {
            int b = graph[curr][next[curr]];
            auto & edge = edges[b];
            auto & rev_edge = edges[b^1];

            int cap = edge.capacity - edge.flow;
            if (cap > 0 && (level[curr] + 1 == level[edge.to])) {
                int bottle_neck = dfs(edge.to, sink, next, min(flow, cap));
                if (bottle_neck > 0) {
                    edge.flow += bottle_neck;
                    rev_edge.flow -= bottle_neck;
                    return bottle_neck;
                }
            }
        }
        return 0;
    }
};

// Example on how to use
void solve()
{
    int n, m;
    cin >> n >> m;
    int N = n + m + 2;

    int source = N - 2;
    int sink = N - 1;

    Dinic flow(N);

    for (int i = 0; i < n; i++) {
        int q; cin >> q;
        while(q--) {
            int b; cin >> b;
            flow.add_edge(i, n + b - 1, 1);
        }
    }
    for (int i =0; i < n; i++) {
        flow.add_edge(source, i, 1);
    }
    for (int i =0; i < m; i++) {
        flow.add_edge(i + n, sink, 1);
    }
```

```
    cout << m - flow.get_max_flow(source, sink) <<
    endl;

    // Getting participant edges
    for (auto & edge: flow.edges) {
        if (edge.capacity == 0) continue; // This
    means is a reverse edge
        if (edge.from == source || edge.to == source)
     continue;
        if (edge.from == sink   || edge.to == sink)
    continue;
        if (edge.flow == 0) continue; // Is not
    participant

        cout << edge.from + 1 << " " << edge.to -n +
    1 << endl;
    }
}
```

1.2 Bellman Ford

Complexity: O(n * m) | n = |nodes|, m = |edges|
Finds shortest paths from a starting node to all nodes of the graph. The node can have negative cycle and belman-ford will detected

```
// TITLE: Bellman Ford
// COMPLEXITY: O(n * m) | n = |nodes|, m = |edges|
// DESCRIPTION: Finds shortest paths from a starting
    node to all nodes of the graph. The node can have
     negative cycle and belman-ford will detected

// a and b vertices, c cost
// [{a, b, c}, {a, b, c}]
vector<tuple<int, int, int>> edges;
int N;

void bellman_ford(int x){
    for (int i = 0; i < N; i++){
        dist[i] = oo;
    }
    dist[x] = 0;

    for (int i = 0; i < N - 1; i++){
        for (auto [a, b, c]: edges){
            if (dist[a] == oo) continue;
            dist[b]= min(dist[b], dist[a] + w);
        }
    }
}
// return true if has cycle
bool check_negative_cycle(int x){
    for (int i = 0; i < N; i++){
        dist[i] = oo;
    }
    dist[x] = 0;

    for (int i = 0; i < N - 1; i++){
        for (auto [a, b, c]: edges){
            if (dist[a] == oo) continue;
            dist[b]= min(dist[b], dist[a] + w);
        }
    }

    for (auto [a, b, c]: edges){
        if (dist[a] == oo) continue;
        if (dist[a] + w < dist[b]){
            return true;
        };
    }
    return false;
}
```
```

## 1.3 Topological Sort

Complexity: O(N + M), N: Vertices, M: Arestas
Retorna no do grapho em ordem topologica, se a quantidade de nos retornada nao for igual a quantidade de nos e impossivel

```
// TITLE: Topological Sort
// COMPLEXITY: O(N + M), N: Vertices, M: Arestas
// DESCRIPTION: Retorna no do grapho em ordem
    topologica, se a quantidade de nos retornada nao
    for igual a quantidade de nos e impossivel

typedef vector<vector<int>> Adj_List;
typedef vector<int> Indegree_List; // How many nodes
    depend on him
typedef vector<int> Order_List;    // The order in
    which the nodes appears

Order_List kahn(Adj_List adj, Indegree_List indegree)
{
    queue<int> q;
    // priority_queue<int> q; // If you want in
    lexicografic order
    for (int i = 0; i < indegree.size(); i++) {
        if (indegree[i] == 0)
            q.push(i);
    }
    vector<int> order;

    while (not q.empty()) {
        auto a = q.front();
        q.pop();

        order.push_back(a);
        for (auto b: adj[a]) {
            indegree[b]--;
            if (indegree[b] == 0)
                q.push(b);
        }
    }
    return order;
}

int32_t main()
{

    Order_List = kahn(adj, indegree);
    if (Order_List.size() != N) {
        cout << "IMPOSSIBLE" << endl;
    }
    return 0;
}
```

1.4 Kth Ancestor

Complexity: O(n * log(n))
Preprocess, then find in log n

```
// TITLE: Kth Ancestor
// COMPLEXITY: O(n * log(n))
// DESCRIPTION: Preprocess, then find in log n

const int LOG_N = 30;
int get_kth_ancestor(vector<vector<int>> & up, int v,
     int k)
{
    for (int j = 0; j < LOG_N; j++) {
```
```

```
        if (k & ((int)1 << j)) {
            v = up[v][j];
        }
    }
    return v;
}

void solve()
{
    vector<vector<int>> up(n, vector<int>(LOG_N));

    for (int i = 0; i < n; i++) {
        up[i][0] = parents[i];
        for (int j = 1; j < LOG_N; j++) {
            up[i][j] = up[up[i][j-1]][j-1];
        }
    }
    cout << get_kth_ancestor(up, x, k) << endl;

}
```

2 String

2.1 Z function

Complexity: Z function complexity
z function

```
// TITLE: Z function
// COMPLEXITY: Z function complexity
// DESCRIPTION: z function

void z_function(string& s)
{
    return;
}
```

3 Set

3.1 Ordered Set

Complexity: O(log(n))

```
// TITLE: Ordered Set
// COMPLEXITY: O(log(n))
// DESCRIPION: Set but you can look witch elements is in position (k)

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

#define ordered_set tree<int, null_type,less<int>, rb_tree_tag,tree_order_statistics_node_update>

int32_t main() {
    ordered_set o_set;

    o_set.insert(5);
    o_set.insert(1);
    o_set.insert(2);
    // o_set = {1, 2, 5}
    5 == *(o_set.find_by_order(2));
    2 == o_set.order_of_key(4); // {1, 2}
}
```

3.2 Set

Complexity: Insertion Log(n)
Keeps elements sorted, remove duplicates, upper_bound, lower_bound, find, count

```
// TITLE: Set
// COMPLEXITY: Insertion Log(n)
// Description: Keeps elements sorted, remove duplicates, upper_bound, lower_bound, find, count

int main() {
  set<int> set1;

  set1.insert(1);       // O(log(n))
  set1.erase(1);        // O(log(n))

  set1.upper_bound(1);  // O(log(n))
  set1.lower_bound(1);  // O(log(n))
  set1.find(1);         // O(log(n))
  set1.count(1);        // O(log(n))

  set1.size();          // O(1)
  set1.empty();         // O(1)

  set1.clear()          // O(1)
  return 0;
}
```

3.3 Multiset

Complexity: O(log(n))
Same as set but you can have multiple elements with same values

```
// TITLE: Multiset
// COMPLEXITY: O(log(n))
// DESCRIPTION: Same as set but you can have multiple elements with same values

int main() {
  multiset<int> set1;
}
```