List of Figures

Figure 1: ./strings/trie/trie.cpp

```cpp
#include "../../headers/headers/headers.h"

class Trie
{
  private:
    vector<unordered_map<char, int>> nodes;
    int next;

  public:
    Trie()
    {
        nodes.eb();
        next = 1;
    }

    bool build(string s)
    {
        int i = 0;
        int v = 0;
        while (i < s.size())
        {
            if (nodes[v].find(s[i]) == nodes[v].end())
            {
                nodes.eb();
                v = nodes[v][s[i]] = next;
                i++;
                next++;
            }
            else
            {
                v = nodes[v][s[i]];
                i++;
            }
        }
    }
};
```

Figure 2: ./graphs/dinic/dinic.cpp

```cpp
#include "../../headers/headers/headers.h"
class Dinic
{
    struct edge
    {
        int to, rev;
        ll f, cap;
    };

    vector<vector<edge>> g;
    vector<ll> dist;
    vector<int> q, work;
    int n, sink;

    bool bfs(int start, int finish)
    {
        dist.assign(n, -1);
        dist[start] = 0;
        int head = 0, tail = 0;
        q[tail++] = start;
        while (head < tail)
        {
            int u = q[head++];
            for (const edge &e : g[u])
            {
                int v = e.to;
                if (dist[v] == -1 and e.f < e.cap)
                {
                    dist[v] = dist[u] + 1;
                    q[tail++] = v;
                }
            }
        }
        return dist[finish] != -1;
    }

    ll dfs(int u, ll f)
    {
        if (u == sink)
            return f;
        for (int &i = work[u]; i < (int)g[u].size(); ++i)
        {
            edge &e = g[u][i];
            int v = e.to;
            if (e.cap <= e.f or dist[v] != dist[u] + 1)
                continue;
            ll df = dfs(v, min(f, e.cap - e.f));
            if (df > 0)
            {
                e.f += df;
                g[v][e.rev].f -= df;
                return df;
            }
```

Figure 3: ./graphs/dijsktra/dijsktra.cpp

```cpp
#include "../../headers/headers/headers.h"
//g has vectors of pairs of the form (w, index)
int dijsktra(wgraph g, int start, int end)
{
    int n = g.size();
    vi cost(n, 1e9); //~INT_MAX/2
    priority_queue<ii, greater<ii>> q;

    q.emplace(0, start);
    cost[start] = 0;
    while (not q.empty())
    {
        int u = q.top().second, w = q.top().first;
        q.pop();

        for (auto v : g[u])
        {
            if (cost[v.second] > v.first + w)
            {
                cost[v.second] = v.first + w;
                q.emplace(cost[v.second], v.second);
            }
        }
    }

    return cost[end];
}
```

Figure 4: ./graphs/dfs/dfsRecursive.cpp

```cpp
#include "../../headers/headers/headers.h"
//Recursive (create visited filled with 1s)
void dfs_r(graph &g, vi &visited, int u)
{
    cout << u << '\n';
    visited[u] = 0;

    for (int v : g[u])
        if (visited[v])
            dfs_r(g, visited, v);
}
```

Figure 5: ./graphs/dfs/dfsIterative.cpp

```cpp
#include "../../headers/headers/headers.h"
//Iterative
void dfs_i(graph &g, int start)
{
    int n = g.size();
    vi visited(n, 1);
    stack<int> s;

    s.emplace(start);
    visited[start] = 0;

    while (not s.empty())
    {
        int u = s.top();
        s.pop();

        cout << u << '\n';

        for (int v : g[u])
            if (visited[v])
            {
                s.emplace(v);
                visited[v] = 0;
            }
    }
}
```

Figure 6: ./graphs/lca/lca.cpp

```cpp
#include "../../headers/headers/headers.h"
class LcaTree
{
    int n;
    vi parent;
    vi level;
    vi root;
    graph P;
public:
    LcaTree(int n){
        this->n = n;
        parent.assign(n,-1);
        level.assign(n,-1);
        P.assign(n,vi(lg(n)+1,-1));
        root.assign(n,-1);
    }
    void addLeaf(int index, int par){
        parent[index] = par;
        level[index] = level[par] + 1;
        P[index][0] = par;
        root[index] = root[par];
        for(int j=1; (1<<j) < n; ++j){
            if(P[index][j-1] != -1)
                P[index][j] = P[P[index][j-1]][j-1];
        }
    }
    void addRoot(int index){
        parent[index] = index;
        level[index] = 0;
        root[index] = index;
    }
    int lca(int u, int v){
        if(root[u] != root[v] || root[u] == -1)
            return -1;
        if(level[u] < level[v])
            swap(u,v);
        int dist = level[u] - level[v];
        while(dist != 0){
            int raise = lg(dist);
            u = P[u][raise];
            dist -= (1<<raise);
        }
        if(u == v)
            return u;
        for(int j = lg(n); j>=0; --j){
            if(P[u][j] != -1 && P[u][j] != P[v][j]){
                u=P[u][j];
                v=P[v][j];
            }
        }
        return parent[u];
    }
};
```

Figure 7: ./graphs/kruskal/kruskal.cpp

```cpp
#include "../../headers/headers/headers.h"
struct edge
{
    int u, v;
    ll w;
    edge(int u, int v, ll w) : u(u), v(v), w(w) {}

    bool operator<(const edge &o) const
    {
        return w < o.w;
    }
};

class Kruskal
{
  private:
    ll sum;
    vi p, rank;

  public:
    Kruskal(int n, vector<edge> E)
    {
        sum = 0;
        p.resize(n);
        rank.assign(n, 0);
        rep(i, n) p[i] = i;
        sort(E.begin(), E.end());
        for (auto &e : E)
            UnionSet(e.u, e.v, e.w);
    }
    int findSet(int i)
    {
        return (p[i] == i) ? i : (p[i] = findSet(p[i]));
    }
    bool isSameSet(int i, int j)
    {
        return findSet(i) == findSet(j);
    }
    void UnionSet(int i, int j, ll w)
    {
        if (not isSameSet(i, j))
        {
            int x = findSet(i), y = findSet(j);
            if (rank[x] > rank[y])
                p[y] = x;
            else
                p[x] = y;

            if (rank[x] == rank[y])
                rank[y]++;

            sum += w;
        }
```

Figure 8: ./graphs/unionFind/unionFind.cpp

```cpp
#include "../../headers/headers/headers.h"
class UnionFind
{
  private:
    int numSets;
    vi p, rank, setSize;

  public:
    UnionFind(int n)
    {
        numSets = n;
        rank.assign(n, 0);
        setSize.assign(n, 1);
        p.resize(n);
        rep(i, n) p[i] = i;
    }
    int findSet(int i)
    {
        return (p[i] == i) ? i : (p[i] = findSet(p[i]));
    }
    bool isSameSet(int i, int j)
    {
        return findSet(i) == findSet(j);
    }
    void UnionSet(int i, int j)
    {
        if (not isSameSet(i, j))
        {
            numSets--;
            int x = findSet(i), y = findSet(j);
            if (rank[x] > rank[y])
            {
                p[y] = x;
                setSize[x] += setSize[y];
            }
            else
            {
                p[x] = y;
                setSize[y] += setSize[x];
                if (rank[x] == rank[y])
                    rank[y]++;
            }
        }
    }
    int numSets()
    {
        return numSets;
    }
    int setOfSize(int i)
    {
        return setSize[i];
    }
};
```

Figure 9: ./graphs/bfs/bfs.cpp

```cpp
#include "../../headers/headers/headers.h"

void bfs(graph &g, int start)
{
    int n = g.size();
    vi visited(n, 1);
    queue<int> q;

    q.emplace(start);
    visited[start] = 0;
    while (not q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v : g[u])
        {
            if (visited[v])
            {
                q.emplace(v);
                visited[v] = 0;
            }
        }
    }
}
```

Figure 10: ./graphs/bellmanFord/bellmanFord.cpp

```cpp
#include "../../headers/headers/headers.h"
bool bellman_ford(wgraph &g, int start)
{
    int n = g.size();
    vector<int> dist(n, 1e9); //~INT_MAX/2
    dist[start] = 0;
    rep(i, n - 1) rep(u, n) for (ii p : g[u])
    {
        int v = p.first, w = p.second;
        dist[v] = min(dist[v], dist[u] + w);
    }

    bool hayCicloNegativo = false;
    rep(u, n) for (ii p : g[u])
    {
        int v = p.first, w = p.second;
        if (dist[v] > dist[u] + w)
            hayCicloNegativo = true;
    }

    return hayCicloNegativo;
}
```