

Grafos

Travessias

Descrição

- dfs: (Depth-First Search) Travessia por profundidade complexidade **$O(N)$** sendo N o número de arestas do vértice u
- bfs: (Breadth-First Search) Travessia por largura; Armazena no vetor dist, a distância do vertice u para os demais vertice; o próprio vértice e vértices que não foram visitados tem distância 0; complexidade **$O(N)$** sendo N o número de arestas do vértice u


```
#define MAXN 1009
bitset<MAXN> visited;
vector<int> adj_list[MAXN];

void dfs(int u){
    visited[u]=true;
    for(auto v:adj_list[u]){
        if(not visited[v]){
            dfs(v);
        }
    }
}

int dist[MAXN];
void bfs(int u){
    queue<int> q;
    q.push(u);
    visited[u]=true;
    memset(&dist, 0, sizeof dist);
    while(not q.empty()){
        auto x = q.front();
        q.pop();
        for(auto v:adj_lis[x]){
            if(visited[v]) continue;

            visited[v]=true;
            dist[v] = dist[x]+1;
            q.push(v);
        }
    }
}
```

Caminhos mínimos

Calcula o caminho mínimo de **s** para **t** em um grafo com **n** arestas numeradas de **1 a n**. Armazena em  **a** aresta antecessora a **x**, desse modo pode-se recuperar o caminho mínimo.

```
//Macros
#define ff first
#define ss second
using ii = pair<int, int>;
const int INF = 1e9;
```

Algoritmo de Bellman-Ford

Tambem pode verificar se o grafo tem ciclo negativo; Complexidade **O(VE)**

```
#define MAXN 1009
vector<ii> adj_list[MAXN];
int dist[MAXN];
int pred[MAXN];

int bellmanford(int s, int t, int n){
    for(int i=1; i<=n; ++i){
        dist[i]=INF;
        pred[i]=-1;
    }
    dist[s]=0; pred[s]=s;
    bool has_negative_cicle = false;
    for(int i=1; i<=n; ++i){
        for(int u=1; u<=n; ++u){
            for(auto x:adj_list[u]){
                auto v=x.ff, w=x.ss;
                if(i==n and dist[v]>dist[u]+w)
                    has_negative_cicle=true;
                else{
                    if(dist[v]>dist[u]+w){
                        dist[v] = dist[u]+w;
                        pred[v] = u;
                    }
                }
            }
        }
    }
    return dist[t];
}
```

Shortest Path Faster Algorithm (SPFA)

Otimização do algoritmo de Bellman-Ford. Retorna **-1** se o grafo tiver ciclo negativo..Complexidade média igual ao algoritmo de dijkstra, no pior caso **O(VE)**;

```
#define MAXN 100009
vector<ii> adj_list[MAXN];
int dist[MAXN];
bool inq[MAXN];
int vis[MAXN];
int pred[MAXN];

int spfa(int s, int t, int n){
    for(int i=1; i<=n; ++i){
        dist[i] = INF;
        pred[i]=-1;
    }
    memset(&inq, false, sizeof inq);
    memset(&vis, 0, sizeof vis);
    dist[s]=0; pred[s]=s;
    queue<int> q;
    q.push(s);
    inq[s]=true;
    while(not q.empty()){
        int u=q.front(); q.pop();
        if(vis[u]>n) return -1;
        inq[u] = false;
        for(auto x:adj_list[u]){
            auto v=x.ff, w=x.ss;
            if(dist[u]+w<dist[v]){
                dist[v]=dist[u]+w;
                pred[v]=u;
                if(not inq[v]){
                    ++vis[v]; q.push(v);
                    inq[v] = true;
                }
            }
        }
    }
    return dist[t];
}
```

Algoritmo de Dijkstra

Tem complexidade melhor que os outros algoritmos porém, **assume que o grafo não contém ciclo negativo**, essa implementação aceita arestas negativas; Complexidade **O(V log V + E)**

```
#define MAXN 100009
vector<ii> adj_list[MAXN];
int dist[MAXN];
int pred[MAXN];
```

```

bitset<MAXN> visited;

int dijkstra(int s, int t, int n){
    for(int i=1; i<=n; ++i){
        dist[i]=INF;
        pred[i]=-1;
    }
    visited.reset();
    dist[s]=0; pred[s]=s;
    set<ii> pq;
    pq.insert(ii(0, s));
    while(not pq.empty()){
        auto u = pq.begin()->second;
        pq.erase(pq.begin());
        if(visited[u]) continue;
        for(auto x:adj_list[u]){
            auto v=x.ff, w=x.ss;
            if(dist[v]>dist[u]+w){
                pq.erase(ii(dist[v], v));
                dist[v]=dist[u]+w;
                pred[v]=u;
                pq.insert(ii(dist[v], v));
            }
        }
    }
    return dist[t];
}

```

Algoritmo de Floyd-Warshall

Calcula todos os caminhos mínimos do grafo em uma matriz de adjacências; **Para a utilização do algoritmo é necessário inicializar os vértices inexistentes com infinito; Complexidade $O(V^3)$**

```

#define MAXN 409
int dist[MAXN][MAXN];
int pred[MAXN][MAXN];

void floyd_warshall(int n){
    for(int u=1; u<=n; ++u)
        pred[u][u] = u;
    for(int k=1; k<=n; ++k){
        for(int u=1; u<=n; ++u){
            for(int v=1; v<=n; ++v){
                if(dist[u][v]>dist[u][k]+dist[k][v]){
                    dist[u][v] = dist[u][k] + dist[k][v];
                    pred[u][v] = pred[k][v];
                }
            }
        }
    }
}

```

Algoritmo sem recuperação do caminho mínimo

[illegible]