Grafos

- ## LISTA DE ADJACÊNCIA:
  - Uma forma de representar um grafo como uma lista de adjacências em C++;
  - O par <nó, custo>: pair <int, int> → typedef pair<int, int> ii;
  - Cada linha é um vetor de pares: vector<ii> → typedef vector<ii> vii;
  - A matriz (que é a lista de adjacências) é um vetor de vetores de pares: vector<vii> adj_list;
  - Construindo o grafo:

```
// n = número de vértices
// m = número de arestas
adj_list.resize(n);
for (int i=0; i < m; ++i)
{
        scanf("%d%d%d", &u, &v, &w);
        adj_list[u].push_back( ii(v, w) );
        adj_list[v].push_back( ii(u, w) );
}
```

- ## BFS:
  - Breadth First Search: Busca em Largura;
  - Ideia:
    - Os nós não visitados são marcados de branco;
    - Os nós visitados mas que ainda não tiveram seus filhos visitados são marcados de cinza;
    - Os nós visitados e com todos os filhos visitados são marcados de preto;
    - Para cada nó, coloca os filhos diretos em uma fila;
    - Depois recupera o primeiro da fila e repete o procedimento;

  - d é a menor distância partindo da origem;
  - v - 1 arestas é o menor caminho;
  - Fila → FIFO (First In First Out);

- Algoritmo:

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

- Implementação:

```
void bfs(int s)
{
    queue<int> q;
    vii::iterator it;
    dist[s] = 0;
    q.push(s);

    while (! q.empty() )
    {
        int u = q.front();
        q.pop();
        // VISITANDO u
        visit(u);
        // colocando vizinhos na fila
        TRvii(adj_list[u], it)
        {
            if (visit_status[it->first] == WHITE)
            {
                visit_status[it->first] = GRAY;
                dist[it->first] = dist[u] + 1;
                q.push( it ->first );
            }
        }
        visit_status[u] = BLACK;
    }

}
```

- DFS:
    - Depth First Search: Busca em Profundidade;
    - Ideia:
        - Visita recursiva;
        - Quando sair da recursão significa que todos os filhos foram visitados (pintado de preto);
        - “time” é uma variável global que armazena quantas visitas foram feitas (pode ser utilizado para saber depois de quantos passos o nó foi encontrado);
        - Algoritmo:

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
        if v.color == WHITE
                v.π = u
                DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

        - Implementação:

```
void dfs(int u)
{
    visit(u);

    visit_status[u] = GRAY;

    vii::iterator neighbor;

    // para cada vértice adjacente
    TRvii(adj_list[u], neighbor)
    {
        // se não tiver visitado, visita-o
        if ( visit_status[neighbor->first] == WHITE )
        {
            dfs(neighbor->first);
        }
    }
    visit_status[u] = BLACK;
```

```
}

int main()
{
    graph1();
    memset(visit_status, WHITE, sizeof(int) *
vertex_size);
    cout << "DFS" << endl;
    dfs(hash('s')); // s
    return 0;
}
```

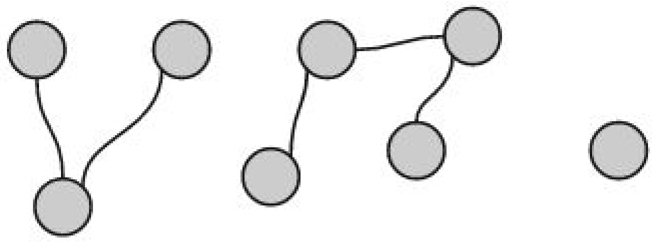
- Nova função: função que visita todos os nós em profundidade mas não repete a visitação;
    - Algoritmo:

DFS(G)

for each vertex $u \in G.V$
    $u.color = \text{WHITE}$
    $u.\pi = \text{NIL}$
$time = 0$
for each vertex $u \in G.V$
    if $u.color == \text{WHITE}$
        DFS-VISIT(G, u)

COMPONENTES CONECTADOS:

- Para casos de nem todos os nós estarem conectados;
- Grupos de componentes conectados (No exemplo abaixo temos um grafo com 3 componentes conectados):



- Implementação:

```
#define REP(i, a, b) \
for (int i = int(a); i <= int(b); i++)

// inside int main()
```

```
numComponent = 0;
memset(dfs_num, DFS_WHITE, sizeof dfs_num);

REP (i, 0, V - 1)
{
      if (dfs_num[i] == DFS_WHITE) {
            printf("Component %d, visit:",
      ++numComponent);
            dfs(i);
            printf("\n");
      }
}

printf("There are %d connected components\n",
numComponent);
```

- FLOOD FILL:
  - Em alguns casos é útil separar os componentes enquanto os visita com DFS. Nesses casos, devemos adaptar o DFS para receber um parâmetro a mais, no caso, a cor (ou um número), ao invés de pintar de preto;
  - Implementação:

```
// dfs adaptado, com um parâmetro a mais
void floodfill(int u, int color)
{
      dfs_num[u] = color; // not just a generic DFS_BLACK
      TRvii (AdjList[u], v) // try all neighbors v of vertex u
            if (dfs_num[v->first] == DFS_WHITE) // avoid cycle
                  floodfill(v->first, color); // v is an
            (neighbor, weight) pair

}

…

// na main():
REP (i, 0, V - 1) // for each vertex u in [0..V-1]
if (dfs_num[i] == DFS_WHITE) // if not visited

      floodfill(i, ++numComponent);
```

- ORDENAÇÃO TOPOLÓGICA:
  - Funciona como uma pilha: FILO (First In Last Out);
  - O primeiro nó a ser pintado de preto é a última tarefa a ser feita;
  - Deve-se modificar o DFS para empilhar quando pintar de preto;
  - Porém isso não funciona se o grafo tiver ciclos;

  - Algoritmo de Khan:
    - Seja n o número de vértices. Devemos enfileirar todos os vértices com indegree 0 em uma fila Q;
    - Implementação:

```
while (!Q.empty()) {
      vertice u = Q.dequeue()
      adiciona u no final de um vetor x
      remover u e todas as arestas que se originam em u
      para cada vértice v adjacente a u
            se v ficou com indegree==0, enfileira v em Q
}
Se sizeof(x) == n, então foi possível encontrar uma
ordenação topológica
```

- MAX-FLOW
  - Ford-Fulkerson:

    Implementação:

```
/* Returns true if there is a path from source 's' to sink 't' in
  residual graph. Also fills parent[] to store the path */
bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
    // Create a visited array and mark all vertices as not visited
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    // Create a queue, enqueue source vertex and mark source vertex
    // as visited
    queue <int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    // Standard BFS Loop
    while (!q.empty())
    {
        int u = q.front();
        q.pop();

        for (int v=0; v<V; v++)
        {
            if (visited[v]==false && rGraph[u][v] > 0)
            {
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }

    // If we reached sink in BFS starting from source, then return
```

```
    // true, else false
    return (visited[t] == true);
}

// Returns tne maximum flow from s to t in the given graph
int fordFulkerson(int graph[V][V], int s, int t)
{
    int u, v;

    int rGraph[V][V]; // Residual graph where rGraph[i][j] indicates
                     // residual capacity of edge from i to j (if
there
                     // is an edge. If rGraph[i][j] is 0, then there
is not)
    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
             rGraph[u][v] = graph[u][v];

    int parent[V];  // This array is filled by BFS and to store path

    int max_flow = 0;  // There is no flow initially

    // Augment the flow while tere is path from source to sink
    while (bfs(rGraph, s, t, parent))
    {
        // Find minimum residual capacity of the edhes along the
        // path filled by BFS. Or we can say find the maximum flow
        // through the path found.
        int path_flow = INT_MAX;
        for (v=t; v!=s; v=parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        // update residual capacities of the edges and reverse edges
        // along the path
        for (v=t; v != s; v=parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }

        // Add path flow to overall flow
        max_flow += path_flow;
    }

    // Return the overall flow
    return max_flow;
}
```

- SHORTEST PATHS:
  - WEIGHTED:
    - Qual o menor caminho a partir de um nó s para qualquer outro nó?
    - Dijkstra: A partir da origem verifica os vizinhos para decidir para onde ir;
      Agenda os caminhos priorizando o menor;
      Faz uma escolha gulosa, expandindo a aresta de menor distância e marca como visitado o nó (evitando loops);
      Vai fazendo as escolhas gulosas de acordo com a fila de prioridades;
      Se o nó já estiver visitado ele estará pintado e então você não precisa visitá-lo novamente;
      E então saberemos o menor custo de um determinado nó para todos os outros;

      Implementação:

```
void dijkstra(int s)
{
    /*
    (peso, vertice)
    Essa fila de prioridade coloca sempre o menor peso nas primeiras posições.
    Assim, o algoritmo prioriza sempre os potenciais menores caminhos.
    */
    priority_queue<ii, vii, less<ii>> queue;


    dist[s] = 0;
    queue.push( {dist[s],s} );

    while (!queue.empty())
    {
        int v = queue.top().second;
        queue.pop();

        if (visited[v]) continue;

        visited[v] = true;


        // verifica as arestas que saem desse vértice
        for (int j = 0; j < adj_list[v].size(); ++j)
        {
            ii u = adj_list[v][j]; // v --> u
            /*
            Se percorrer a aresta para chegar em u for menor
            do que o menor u até o momento, então encontramos
            um novo menor caminho até u
            */
            if (dist[v] + u.second <  dist[u.first])
            {
                dist[u.first] = dist[v] + u.second;
                queue.push({dist[u.first], u.first});
            }
        }
    }
}
```

  - UNWEIGHTED:
    - Faz uma redução: Unweighted shortest paths $\rightarrow$ weighted shortest paths;
    - (Add weights of any number > 0);
  - WITH NEGATIVE WEIGHTS:
    - Bellman-Ford:

- Em um grafo G = (V, E), se existir um caminho entre dois vértices, esse caminho tem no máximo V-1 arestas. Logo, se um grafo tiver ciclos negativos, podemos limitar o loop a V-1 vezes. Ou seja, iremos percorrer o grafo V-1 vezes;
- Se houver um ciclo negativo, o custo desse caminho fica indefinido. Afinal, a depender da quantidade de vezes que o ciclo for executado, o custo vai diminuindo. Mas é sempre possível guardar o custo em si (basta guardar a aresta incidente cada vez que um menor custo for encontrado);
- Para detectar um ciclo negativo: rodar mais uma vez os 2 loops internos, se ainda encontrarmos um menor caminho para qualquer nó, é porque temos um ciclo negativo;
- Implementação:

```
vi dist(V, INF); dist[s] = 0;
for (int i = 0; i < V - 1; i++)              // relax all E edges V-1 times
  for (int u = 0; u < V; u++)      // these two loops = O(E), overall O(VE)
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];            // record SP spanning here if needed
      dist[v.first] = min(dist[v.first], dist[u] + v.second);     // relax
    }
```

```
// after running the O(VE) Bellman Ford's algorithm shown above
  bool hasNegativeCycle = false;
  for (int u = 0; u < V; u++)                          // one more pass to check
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (dist[v.first] > dist[u] + v.second)   // if this is still possible
        hasNegativeCycle = true;            // then negative cycle exists!
    }
  printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");
```

- SPFA (Chinês):
  - Uma melhoria sobre o Bellman-Ford;
  - A ideia usada é basicamente a mesma do Bellman-ford, mas ao invés de tentar todos os vértices “cegamente”, o SPFA mantém uma fila de vértices candidatos e adiciona um vértice para a fila somente se aquele vértice é “relaxed” (termo usado no wikipedia em inglês);
  - Pseudo-algoritmo:

```
procedure Shortest-Path-Faster-Algorithm(G, s)
     for each vertex v ≠ s in V(G)
         d(v) := ∞
     d(s) := 0
```

```
 offer s into Q
 while Q is not empty
     u := poll Q
     for each edge (u, v) in E(G)
         if d(u) + w(u, v) < d(v) then
             d(v) := d(u) + w(u, v)
             if v is not in Q then
                 offer v into Q
```

- Implementação:

```
/*
Retorna true caso exista um ciclo negativo, falso
caso contrário.
Ao final do algoritmo, o vector dist terá todas as
distâncias de s para cada vértice.
*/
bool spfa(int s)
{
	int u, v, w;
	queue<int> q;
	vi counter(n, 0); // usado para evitar o loop
	vector<bool> in_queue(n, false);

	fill(dist.begin(), dist.end(), INF);
	dist[s] = 0;
	q.push(s);
	counter[s]++;
	in_queue[s] = true;

	while (! q.empty() )
	{
		u = q.front(); q.pop();
		in_queue[u] = false;

		for (int j=0; j < adj_list[u].size();
++j)
		{
			v = adj_list[u][j].first;
			w = adj_list[u][j].second;

			if (dist[u] + w < dist[v])
			{
				dist[v] = dist[u] + w;

				if ( !in_queue[v] )
				{
					q.push(v);

					in_queue[v] = true;
					counter[v]++;
					if (counter[v] > n-1)
```

```
                                        {
                                                // ciclo
negativo encontrado. Pára o algoritimo
                                                return true;
                                        }
                                }
                        }
                }
        }
        return false;
}
```

- PONTOS DE ARTICULAÇÃO:
  - É um vértice que se removido deixará algum nó desconectado no grafo;
  - Algoritmo naive:
    - Rode o DFS e conte o número de componentes conectados;
    - Para cada vértice V do grafo:
      - Retire o vértice V e suas arestas;
      - Rode o DFS e verifique se o número de componentes aumenta;
      - Se aumentar, V é um ponto de articulação;
      - Coloque o vértice V de volta no grafo e suas arestas;
      - Complexidade: O(v^2 + VE);
  - Algoritmo sagaz:
    - Rodar o DFS;
    - Ao visitar o nó, guardamos o momento que ele foi visitado;
    - Em cada nó, guardamos também o menor momento de visitação de todos os seus filhos;
    - “O único caminho para V é através de U e, portanto U é um ponto de articulação, pois se ele não existir, V não seria alcançado”;
    - Implementação:

```
void dfs(int u)
{
        int children = 0;
        discovery[u] = low[u] = discovery_time++;
        color[u] = GRAY;
        TRvi ( adj_list[u] , v)
        {
                if (color[*v] == WHITE)
                {
                        parent[*v] = u;
                        ++children;
                        dfs(*v);
                        bool is_root = parent[u] == NONE;
                        if ( is_root  && children>1) // caso
especial
                        {
                                articulation_vertex[u] = true;
                        }
                        else if (!is_root && discovery[u] <=
```

```
low[*v] )
                {
                        articulation_vertex[u] = true;
                }
                low[u] = MIN(low[u], low[*v]);
        }
        else if ( *v != parent[u] ) // back edge, mas
não direto
        {
                low[u] = MIN( low[u] , discovery[*v] );
        }
    }
}
```

- Caso especial: root

```
if (color[*v] == WHITE)
{
        ...
        bool is_root = parent[u] == -1;
        if ( is_root  && children>1)
        {
                articulation_vertex[u] = true;

        }
...
```

- Na main:

```
for (int u=0; u< places; ++u)
{
        if (color[u]==WHITE)
        {
                dfs(u);
        }
}
for (int u=0; u< places; ++u)
{
        if (articulation_vertex[u])
        {
                ++total_articulation_vertex;
        }
}
```

- PONTE:
    - `if (discovery[u] < low[*v])`
      `então (u,v) é ponte`