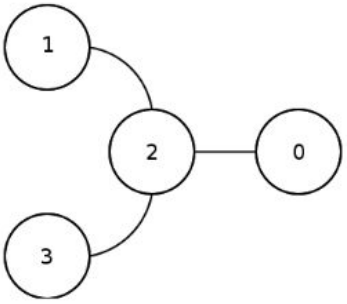
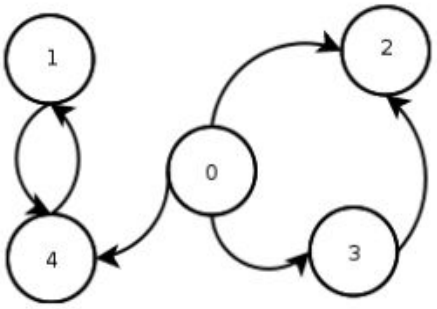
Grafos

Grafos é uma estrutura de dados que contém um conjunto de vértices e arestas. Para cada arestas pode ser, também, associado um valor (peso).

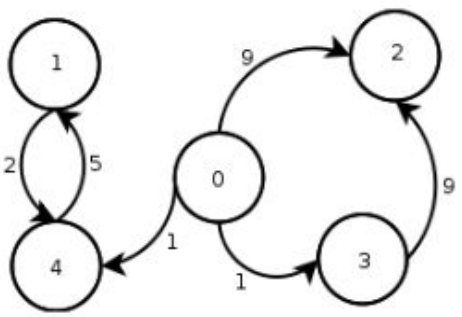
Os grafos podem ser direcionados ou não, a figura a seguir mostra um grafo não direcionado. Nos **grafos não direcionados**, nenhuma aresta tem sentido, ou seja, se existe uma aresta de A para B, A tem conexão com B assim como B tem conexão com A.



Quando temos **grafos direcionados** (também conhecidos como digrafos), toda aresta tem um sentido, ou seja, se existe uma aresta de A para B, A tem conexão com B, porém B não necessariamente tem uma conexão com A. A figura a seguir mostra um grafo dirigido.



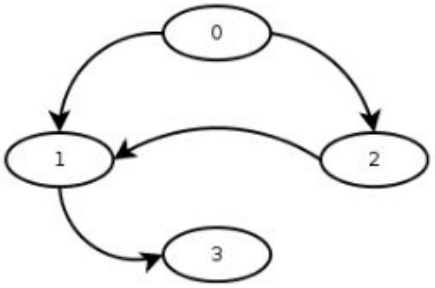
Nos **grafos ponderados** (ou valorados), as arestas possuem pesos, nesse caso arestas com pesos maiores geralmente representam transições mais custosas. Na figura a seguir é mostrado um grafo não dirigido ponderado, note-se que o caminho de 1-4, tem peso diferente de 4-1.



Representação de grafos

Predominantemente são utilizadas duas abordagens para representar grafos: **matrizes de adjacência** e **listas de adjacência**. Aqui utilizaremos listas de adjacência pois, em geral, apresentarem maior desempenho. Digamos que nosso grafo possui n elementos, algumas operações sobre matrizes de adjacência podem necessitar a análise de n^2 elementos (todas as ligações) enquanto com listas de adjacência verificaremos apenas o número de vezes igual o número de arestas do grafo (que é no máximo n^2).

Utilizaremos “vetores de vetores de inteiros” para representar os grafos, caso sejam grafos ponderados será utilizada pares de inteiros `pair<int, int>`. Desejamos representar o grafo da figura a seguir:



A declaração do **grafo não ponderado** (direcionado ou não direcionado) será:

```
vector< vector<int> > grafo;
```

Cada posição do vetor representa um vértice, nela está contido um outro vetor - a lista de adjacências. Essa lista possui as arestas que partem desse vértice - no caso de um grafo não dirigido adicionamos em ambas arestas.

Os dados contidos em cada uma das posições do vetor do grafo acima será:

grafo[0] = {1, 2}
grafo[1] = {3}
grafo[2] = {1}
grafo[3] = {}

Já a declaração do grafo do **grafo ponderado direcionado** será:

```
vector < pair<int, int> > grafo[N];
```

Cada posição do vetor representa um vértice, nela está contido um vetor de pares de inteiros. Cada par de inteiro representa o **vértice que é conectado e o peso**, respectivamente.

Os dados contidos em cada uma das posições do vetor do grafo ponderado (página anterior) será:

grafo[0] = {<2, 9>, <3, 1>, <4, 1>}
grafo[1] = {<4, 2>}

grafo[2] = {}
grafo[3] = {<2, 9>}
grafo[4] = {<1, 5>}

Caso seja necessário realizar uma ordenação dos pesos das arestas (ex: algoritmo de Kruskal), pode-se usar a seguinte estrutura:

```
vector< pair< int, pair<int,int> > > grafo;
```

Cada posição do vetor representa uma aresta, nela está contido um inteiro e um par de inteiros, que representam, respectivamente, o peso e o par de vértices ligados.

Vale ressaltar que representando o grafo desta forma, não estaremos usando lista de adjacência.

Exemplo de leitura de um grafo

No exemplo abaixo lemos um grafo e posteriormente mostramos as ligações de cada vértice. A primeira linha da entrada possui dois inteiros N e M que representa o número de vértices no grafo e o número de arestas. É seguida por M linhas contendo inteiros A e B, representando uma aresta (direcionada) de A para B. O exemplo anterior teria o formato:

4 4
0 1
0 2
1 3
2 1

O programa abaixo lê um grafo nesse formato e depois percorre todo o grafo mostrando as arestas.

```
#include<iostream>
#include<vector>

using namespace std;

int main()
{
	vector< vector<int> > g;

	int n, m; // número de vértices e arestas
	cin >> n >> m;

	g.resize(n);

	int a, b;
```

```
	for(int i = 0; i < m; i++)
	{
		cin >> a >> b;
		g[a].push_back(b);
	}

	for(int i = 0; i < g.size(); i++)
	{
		cout << i << ":";

		for(int j = 0; j < g[i].size(); j++)
			cout << " " << g[i][j];

		cout << "\n";
	}

	return 0;
}
```

Busca em largura

Esta seção trata da busca de um caminho em um grafo, em especial, uma busca onde garantimos que os **vértices visitados primeiramente terão suas arestas examinadas anteriormente a outros**. Eis que, partindo de um vértice “a” qualquer e garantindo essa restrição, na primeira vez que atingirmos o vértice “b” podemos afirmar que este caminho possui o número mínimo de passos até tal, ou seja, que este é o menor caminho entre “a” e “b”. Isso funciona pois os vértices visitados mais cedo estão mais próximos do vértice inicial, e consequentemente tem distância menor até a origem.

Eis que para garantir esta restrição, basta que utilizemos uma **fila** para armazenar os vértices visitados. O exemplo a seguir mostra uma rotina para busca em largura utilizando listas de adjacência.

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

void bfs (int N, int inicio, int final, vector<vector<int>> adj)
{
	bool visitados[N];
	int sucessor[N], distancia[N];
	queue<int> f;
```

```
        for(int i = 0; i < N; i++)
        visitados[i] = false;

        distancia[inicio] = 0;
        f.push(inicio);

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

            for(int i = 0; i < adj[a].size(); i++)
                if(!visitados[adj[a][i]])
                {
                    visitados[adj[a][i]] = true;
                    sucessor[adj[a][i]] = a;
                    distancia[adj[a][i]] = distancia[a] + 1;
                    f.push(adj[a][i]);

                    if(adj[a][i] == final)
                        return;
              }
        }
}
```

O algoritmo de Busca em Largura também serve para algumas outras aplicações, são elas:

- Descobrir se um grafo é **bipartido**

Na implementação da BFS será preciso guardar em que "lado do grafo" você está e um novo vetor que te diga a que "lado do grafo" pertence cada vértice. Caso algum dos vértices que tem conexão com o vértice atual já tenha sido setado como pertencente ao mesmo "lado do grafo" que o vértice atual está, então o grafo não é bipartido.

- Descobrir o **menor caminho** de A para B em um grafo com arestas de valor 1

Os vizinhos diretos de A estão a uma distância de 1 para A, os vizinhos dos vizinhos de A estão a uma distância de 2 para A e assim sucessivamente. (vetor *distancia* do algoritmo)

- Encontrar todos os **nós de uma componente conexa**

Guardar a informação de quais nós foram visitados ou não.

Busca em profundidade

Outra estratégia possível para realizar uma busca, é **visitar as arestas do vértice mais recentemente visitado**, esta estratégia é especialmente eficiente quando desejamos apenas verificar a existência de um caminho entre dois vértices (ou quando por definição existe apenas um). Esta busca é a busca em profundidade.

A busca em profundidade possui ainda diversas aplicações, das quais podemos citar **busca de ciclos**, **bipartição de grafos**. Embora sua simplicidade, o entendimento da busca em profundidade é necessário para compreensão de diversos algoritmos úteis.

Como dito anteriormente, verificamos primeiro os vértices atingidos mais recentemente, algoritmicamente falando, utilizamos uma **pilha** para guardar a ordem de verificação. A dinâmica é a mesma da busca em largura, no exemplo abaixo testamos apenas se existe um caminho entre um nó inicial e final.

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

bool dfs(int N, int inicio, int final, vector<vector<int>> adj)
{
      bool visitados[N];
      stack<int> s;

      for(int i = 0; i < N; i++)
      visitados[i] = false;

      s.push(inicio);

      while(!s.empty())
      {
          int a = s.top();
          s.pop();

          for(int i = 0; i < adj[a].size(); i++)
              if(!visitados[adj[a][i]])
              {
                  visitados[adj[a][i]] = true;
                  s.push(adj[a][i]);

                  if(adj[a][i] == final)
                      return true;
```

```
                }
        }

        return false;
}
```

O algoritmo de Busca em Profundidade também serve para algumas outras aplicações, são elas:

- Descobrir **componentes conexas**

Em um grafo não-direcionado, ao rodar dfs(V) - algoritmo acima não parando quando encontra o caminho entre os dois vértices - sendo V um vértice qualquer, o algoritmo visita todos os vértices da componente conexa de V.

- Descobrir **componentes fortemente conexas**

Sejam G um grafo direcionado, G' o grafo inverso (com as direções das arestas invertidas) e V um vértice qualquer. Se ambos dfs(G, V) e dfs(G', V) - mesmo algoritmo da aplicação acima - visitam todos os vértices, então o grafo G é fortemente conexo (por consequência, G' também).

Caminhos Mínimos em grafos ponderados

Esta seção trata do problema de encontrar o caminho mínimo entre dois pontos num grafo ponderado, isto é, dado um grafo com arestas valoradas **descobrir qual o caminho de um vértice A para outro B que possui o menor custo total** (valor somado).

Algoritmo de Dijkstra

Nesta seção apresentamos o algoritmo proposto por Dijkstra para encontrar o menor caminho em um grafo ponderado de **pesos positivos** (Se o grafo possui também pesos negativos é necessário utilizar o algoritmo de Bellman-Ford).

Este algoritmo é semelhante a busca em largura, entretanto, aqui sempre iremos expandir o nó que atualmente tem menor custo, isto é, tentaremos atingir o nó final sempre pelo caminho que possui menor custo total e para isso, sempre devemos escolher o menor para cada nó intermediário. A primeira vez que atingimos um vértice não é o menor caminho até este.

Para memorizar quais os caminhos de menor custo, utilizaremos uma fila de prioridade (priority_queue), será necessário ainda armazenar o menor custo para cada vértice (objetivo do algoritmo).

```
#include <iostream>
#include <vector>
```

```cpp
#include <queue>
#define INF 0x3F3F3F3F

using namespace std;

typedef pair<int,int> ii;

// vértice que será calculado distância para os outros,
// lista de adjacência e número de vértices
void dijkstra(int s, vector<ii> *adj, int N)
{
      int D[N], pi[N]; // distância e antecessor

      for(int i = 0; i < N; i++)
      {
          D[i] = INF;
          pi[i] = -1;
      }

      priority_queue< ii, vector< ii >, greater< ii > > Q;

      D[s] = 0;
      Q.push(ii(0, s));

      while(!Q.empty())
      {
          ii top = Q.top();
          Q.pop();

          int u = top.second, d = top.first;

          if( d <= D[u] )
              for(int i = 0; i < adj[u].size(); i++)
              {
                  int v = adj[u][i].first, cost = adj[u][i].second;

                  if( D[v] > (D[u] + cost) )
                  {
                      D[v] = D[u] + cost;
                      pi[v] = u;
                      Q.push(ii(D[v], v));
                  }
            }
      }
}
```

No primeiro loop da função inicializamos o vetor com as menores distâncias, atribuímos como “infinito” para identificar que este vértice não foi visitado ainda, também inicializamos os sucessores.

A busca acontece no `while(!Q.empty())`. Avaliamos o vértice que possui a menor distância até o vértice inicial, e verificamos cada uma das suas adjacências (`for(int i = 0; i < adj[u].size(); i++)`). O teste mais característico do algoritmo é encontrado no `if(D[v] > (D[u] + cost))`.

Eis que, se visitarmos todos os caminhos que chegam num dado vértice e memorizarmos o de menor custo, podemos afirmar que esse é o menor custo entre a origem e o vértice em questão. Esse algoritmo faz isso para todos vértices, partido de ”s”; Uma vez executado a partir de ”s”, a menor distância de ”s” até qualquer vértice ”v”, será aquela armazenada em `D[v]`.

Bellman-Ford

Nesta seção apresentamos o algoritmo de Bellman-Ford para encontrar o menor caminho em um grafo ponderado que também possua **pesos negativos**.

```
#include <iostream>
#include <vector>
#define INF 0x3F3F3F3F

using namespace std;

typedef pair<int,int> ii;

// retorna false se tem ciclo negativo, true caso contrário
bool bellmanFord(int N, int s, vector<ii> *adj)
{
      int dis[N];
      bool ret = true;

      for (int i = 0; i < N; i++)
      dis[i] = INF;

      dis[s] = 0;

      // Relaxamento das arestas
      for (int i = 0; i < (N-1); i++)
      {
          for(int j = 0; j < N; j++)
          {
              for(int k = 0; k < adj[j].size(); k++)
              {
```

```
                    if (dis[j] + adj[j][k].second <
dis[adj[j][k].first])
                        dis[adj[j][k].first] = dis[j] +
adj[j][k].second;
                }
            }
        }

        // Checando se existe ciclos negativos
        for(int i = 0; i < N; i++)
        {
            for(int j = 0; j < adj[i].size(); j++)
            {
                if(dis[i] != INF && ((dis[i] + adj[i][j].second) <
                                        dis[adj[i][j].first]))
                    ret = false;
            }
        }

        for (int i = 0; i < N; i++)
            cout << i << " " << dis[i] << endl;

        return ret;
}
```

Algoritmo de Floyd-Warshall

Em alguns casos é necessário determinar o menor caminho entre todos os pares de vértices do grafo, nesse caso utiliza-se o algoritmo Floyd-Warshall.

```
#include <iostream>
#include <vector>
#define INF 0x3F3F3F3F

using namespace std;

typedef pair<int,int> ii;

void floydWarshall(int N, vector<ii> *adj)
{
        int dis[N][N];

        for (int i = 0; i < N; i++)
        {
```

```
            for(int j = 0; j < N; j++)
            {
                if(i != j)
                    dis[i][j] = INF;
                else
                    dis[i][j] = 0;
            }
        }

        for(int i = 0; i < N; i++)
            for(int j = 0; j < adj[i].size(); j++)
                dis[i][adj[i][j].first] = adj[i][j].second;

    for(int k = 0; k < N; k++)
        for(int i = 0; i < N; i++)
            for(int j = 0; j < N; j++)
                if(dis[i][j] > dis[i][k] + dis[k][j])
                    dis[i][j] = dis[i][k] + dis[k][j];

    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            cout << i << " " << j << " = " << dis[i][j] << endl;
}
```

Árvore Geradora Mínima (Minimum Spanning Tree - MST)

Esta seção trata de árvores geradoras em um grafo não dirigido. Árvore geradora é qualquer subárvore (**n vértices, n - 1 arestas e conexo**) que contenha todos os vértices do grafo.

Se considerarmos um grafo ponderado, o custo de uma subárvore é a soma dos valores de suas arestas. Eis que o objetivo desta seção é **determinar a árvore geradora com o menor custo total**, nominada árvore geradora mínima.

Algoritmo de Prim

O algoritmo de Prim encontra a árvore geradora mínima para um dado grafo, seu funcionamento é baseado nas condições de otimalidade das árvores geradoras, mais especificamente na propriedade dos cortes:

> Se T é uma MST de um grafo, então cada uma das arestas t de T é uma aresta mínima dentre as que atravessam o corte determinado por T-t.

Segue abaixo uma implementação do algoritmo de Prim, o retorno da função é o custo total da MST.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#define INF 0x3F3F3F3F

using namespace std;

typedef pair<int,int> ii;

int prim(int N, vector<ii> *adj)
{
    int D[N], pi[N];
    bool visited[N];
    int ans = 0;

    for(int i = 0; i < N; i++)
    {
       D[i] = INF;
       pi[i] = -1;
       visited[i] = false;
    }

    priority_queue< ii, vector<ii>, greater<ii>> Q;

    D[0] = 0;
    Q.push(ii(0,0));

    while(!Q.empty())
    {
       ii top = Q.top();
       Q.pop();
       int u = top.second, d = top.first;

       if(!visited[u])
       {
             ans += d;
             visited[u] = true;

             for(int i = 0; i < adj[u].size(); i++)
             {
                   int v = adj[u][i].first, cost = adj[u][i].second;

                   if(!visited[v] && (D[v] > cost))
                   {
                         D[v] = cost;
```

```
                        pi[v] = u;
                        Q.push( ii(D[v], v));
                    }
                }
            }
        }

    return ans;
}
```

O algoritmo consiste em **acrescentar à árvore um vértice por vez, usando como critério a aresta com menor custo entre a árvore atual e os vértices restantes**.

Algoritmo de Kruskal

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef pair<int,int> ii;
typedef pair<int,ii> iii;

int kruskal(int n, vector<iii> arestas)
{
      int u, v;
      int pi[n], comp_sz[n];
      int result = 0;

      iii e;

      for(int i = 0; i < n; i++)
      {
      comp_sz[i] = 1;
      pi[i] = i;
      }

      // ordenação das arestas
      sort(arestas.begin(), arestas.end());

      for(int i = 0, k = 0; i < arestas.size() && k < n-1; i++)
      {
          e = arestas[i];
```

```
            //union-find
            for(u = (e.second).first; u != pi[u]; u = pi[u]);
            for(v = (e.second).second; v != pi[v]; v = pi[v]);

            if(u == v) //se a aresta gera um ciclo
                continue;

            if(comp_sz[u] < comp_sz[v])
            {
                pi[u] = v;
                comp_sz[v] += comp_sz[u];
            }

            else
            {
                pi[v] = u;
                comp_sz[u] += comp_sz[v];
            }

            result += arestas[i].first;
            k++;
        }

        for(int i = 0; i<n; i++)
            cout << comp_sz[i] << endl;

        return result;
}
```

O algoritmo consiste em **transformar uma floresta geradora em uma árvore geradora, acrescentando as árvores por meio da inserção ordenada de arestas (menos custosas para mais custosas)**.

Fluxo máximo em redes

Esta seção trata de fluxo máximo em redes. Uma rede de fluxo é um grafo orientado, onde cada aresta possui dois valores associados: **capacidade e fluxo**. Uma rede começa no vértice de partida (fonte) e termina no vértice terminal (sumidouro). É importante lembrar que toda a **quantidade enviada na fonte deve ser recebida no sumidouro**, o **fluxo de cada aresta não ultrapassa sua capacidade** e a **soma dos fluxos que chegam num vértice é igual a soma dos que saem**.

Algoritmo de Edmonds-Karp

```
#include <iostream>
#include <string.h>
#include <queue>
#define INF 0x3F3F3F3F

using namespace std;

typedef pair<int, int> ii;

bool bfs(int n, vector<vector<int>> rGraph, int s, int t, int* parent)
{
    bool visited[n];

    for(int i = 0; i < n; i++)
       visited[i] = false;

    queue <int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

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

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

int fordFulkerson(int n, int s, int t, vector<ii> *G)
{
    // grafo residual
    vector<vector<int>> rGraph;
```

```
    rGraph.resize(n);

    for(int i = 0; i < n; i++)
    {
       rGraph[i].resize(n);
       for(int j = 0; j < n; j++)
             rGraph[i][j] = 0;
    }

    //first é o vértice que é conectado, second é a capacidade da aresta
    for(int i = 0; i < n; i++)
       for(int j = 0; j < G[i].size(); j++)
             rGraph[i][G[i][j].first] = G[i][j].second;

    int parent[n]; // preenchida pelo BFS, guarda o caminho
    int max_flow = 0;

    while (bfs(n, rGraph, s, t, parent))
    {
       int path_flow = INF;
       for (int v = t; v != s; v = parent[v])
       {
             int u = parent[v];
             path_flow = min(path_flow, rGraph[u][v]);
       }

       // atualizando a capacidade das arestas
       for (int v = t; v != s; v = parent[v])
       {
             int u = parent[v];
             rGraph[u][v] -= path_flow;
             rGraph[v][u] += path_flow;
       }

       max_flow += path_flow;
    }

    return max_flow;
}
```

A implementação acima do **Algoritmo de Ford-Fulkerson** é chamada de **Algoritmo de Edmonds-Karp**. A ideia de Edmonds-Karp é usar o BFS na implementação do Ford Fulkerson, pois o BFS sempre escolhe um caminho com um número mínimo de arestas. Quando o BFS é usado, a pior complexidade de tempo é reduzida.

A ideia do algoritmo é fazer um **grafo residual** e **enquanto houver um caminho aumentante, aumentar o fluxo** do caminho com o mínimo dos fluxos.