

Aula 8 · Introdução e Busca em Grafos

Desafios de Programação

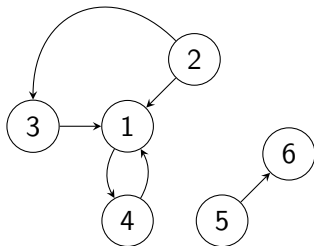
Fernando Kiotheka Vinicius Tikara Date

UFPR

30/10/2024

Introdução a Grafos

Grafos direcionados



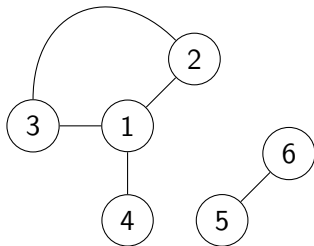
$$G = (V, E)$$

$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{(3, 1), (2, 1), (2, 3), (1, 4), (4, 1), (5, 6)\}$$

Um grafo direcionado é definido como um par ordenado (V, E) , onde V é um conjunto de vértices e E um conjunto de arestas (ou arcos) de pares ordenados (u, v) onde $u, v \in V$.

Grafos não direcionados



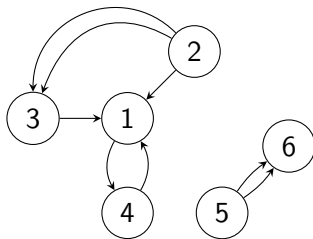
$$G = (V, E)$$

$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{\{3, 1\}, \{2, 1\}, \{2, 3\}, \{1, 4\}, \{5, 6\}\}$$

Um grafo não direcionado (que é o mais “simples”) é a mesma coisa, mas a direção das arestas não importa, então o conjunto de arestas contém conjuntos de dois elementos $\{u, v\}$ onde $u, v \in V$. Podemos definí-los como grafos direcionados onde para cada aresta (u, v) , existe uma aresta (v, u) correspondente.

Multigrafos



$$G = (V, E)$$

$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{(3, 1), (2, 1), (2, 3), (2, 3), (1, 4), (4, 1), (5, 6), (5, 6)\}$$

Pode ser o caso onde E é um conjunto com elementos repetidos (multiconjunto), esse grafo é caracterizado como multigrafo. Multigrafos podem ser direcionados ou não direcionados.

Mas por que classificar grafos?

Saber a natureza de um grafo é o primeiro passo para a resolução de um problema com grafos. Podemos com isso, por exemplo, pensar casos de borda possíveis:

- Podemos ir e voltar?
- Quais algoritmos são adequados?
- Ciclos?
- Arestas repetidas na entrada?

Veremos mais classificações de grafos a seguir, mas primeiro, vamos olhar para jeitos de se representar grafos no nosso programa.

Representações de grafos

Há pelo menos três jeitos de se representar grafos:

- Lista de adjacência. Cada vértice u contém uma lista de seus vizinhos. Exemplo: `vector<vector<edge>> adj (N)`.
- Matriz de adjacência. Cada célula da matriz denota a informação sobre a aresta (u, v) . Exemplo: `vector<vector<edge>> mat (N, vector<edge>(N))`.
- Vetor de arestas. Cada entrada do vetor representa uma aresta. Exemplo: `vector<edge> edges (M)`. Pode ser bem útil por exemplo para conseguir a aresta oposta rápido: `edges[e]` e `edges[e~1]` (assumindo inserção nesta ordem).

Cada jeito de representar será mais benéfico ao uso de certos algoritmos. Pode ser útil também ficar convertendo entre representações diferentes.

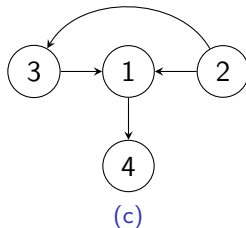
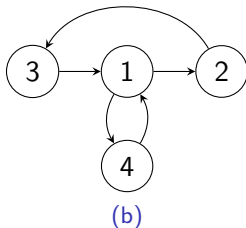
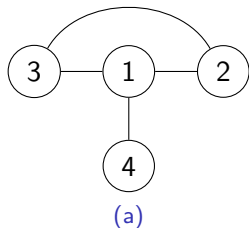
Informações associadas aos vértices, ou as arestas

- Colocamos de um jeito genérico o edge porque as informações associadas a arestas variam.
- Exemplos comuns incluem: O peso da aresta, o custo da aresta, a capacidade de fluxo da aresta, a probabilidade da aresta, etc.
- Geralmente podemos usar um `pair<int, T>` numa lista de adjacências, ou criar uma estrutura de dados na lista de arestas com todos os atributos necessários.
- Frequentemente o grafo é criado implicitamente no problema. Em um espaço 2D, podemos olhar para os quadrados vizinhos, e isso forma implicitamente uma aresta entre cada quadrado e seus vizinhos, porém não é necessário antes de resolver o problema criar essas arestas explicitamente.
- Podem existir informações associadas aos próprios vértices. Saber se uma informação é referente a uma aresta ou a um vértice é de suma importância.

Famílias de grafos

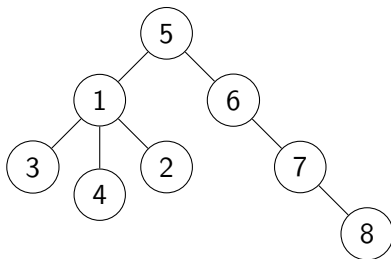
- Já classificamos os grafos de acordo com as suas arestas.
- Agora veremos outra classificação útil: Famílias de grafos.
- Não veremos todas, existem muitas e muitas, por exemplo:
 - Grafos *split*
 - Grafos planares
 - etc.
- Mas veremos os mais importantes *por enquanto*.

Grafos conectados



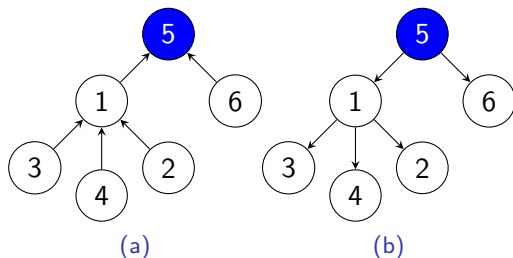
- Um grafo não direcionado é **conectado** (a) se todos os vértices são conectados por um caminho.
- Um grafo direcionado é **fortemente conectado** (b) se todos os vértices são conectados por um caminho.
- Um grafo direcionado é **fracamente conectado** (c) se todos os vértices são conectados ignorando a direção das arestas.

Árvores



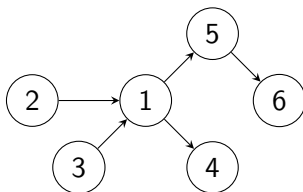
- Árvores são grafos não direcionados onde todos os vértices tem apenas **um** caminho para qualquer outro vértice.
- Também são chamados de grafos não direcionados acíclicos (sem ciclos).
- Quando um grafo representa várias árvores desconexas, ele é chamado de floresta.

Arborescências ou árvores enraizadas



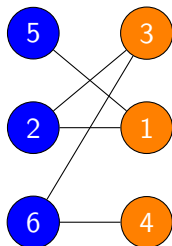
- Designamos um vértice da árvore como raiz (todos servem).
- Criamos um grafo direcionado a partir disso, sendo que as arestas ficam na mesma direção — arborescência (a) — ou em direção oposta a raiz — anti-arborescência (b).
- É essencial para a resolução de muitos problemas (e é parecido por exemplo, com o padrão de busca em profundidade que veremos a seguir).
- Toda arborescência é um DAG (grafo acíclico direcionado).

Acíclicos direcionados (DAGs)



- Todos os grafos direcionados sem ciclos.
- Modelagem de dependências (dependência cíclica é problema).
- Podem ser topologicamente ordenados (isto é, existe uma ordem que permite que todas as dependências sejam resolvidas antes da própria tarefa).
- Todo grafo direcionado com ciclos pode ser reduzido a uma versão “compactada” sem ciclos, veremos esse processo na próxima vez que revisitarmos grafos.

Bipartidos



- Geralmente é um grafo não direcionado.
- Vértices de um lado se ligam apenas a vértices do outro lado.
- Toda árvore é um grafo bipartido, o inverso não é verdade.
- Um grafo é bipartido se e somente se não tem ciclo ímpar.
- Associado ao problema do emparelhamento máximo.
- Existem variações: Grafos k -partidos.

Percorrendo Grafos

Como busco um vértice?

A busca em grafos é iniciada por um vértice inicial. O vértice inicial adequado depende da operação que se vai fazer, sendo que muitas vezes qualquer vértice do grafo serve.

Em profundidade (*dfs, depth-first search*) Cada nó é explorado o mais fundo possível antes de voltar (backtracking). Controle feito por uma pilha.

Em largura (*bfs, breadth-first search*) “Camada por camada”, visitando todos os vértices que estão a distância 1, então 2, e assim por diante do vértice inicial. Controle feito por uma fila.

Ambos os algoritmos são $O(V + E)$, pois consideramos todos os vértices e todas as arestas (mesmo se não as visitamos).

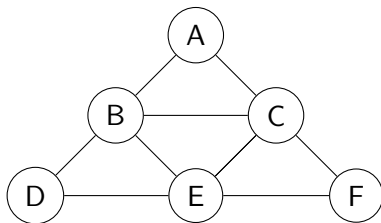
Problema da contagem de vértices

Vamos formular um problema simples para aplicar estas buscas.

- Temos um grafo de entrada não direcionado e várias consultas.
- A consulta quer saber quantos vértices são *alcançáveis* de um determinado vértice u (existe um caminho de u para eles).

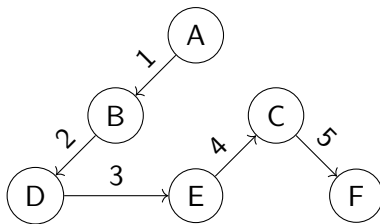
Como resolver?

Exemplo em um grafo



Note que a ordem de caminhamento no grafo é **arbitrária**.
Mostraremos um exemplo, mas dependendo da ordem das arestas, a ordem da visita pode ser completamente diferente. Vamos supor que queremos contar a partir do vértice A.

DFS



Busca em profundidade recursiva

Utilizamos a pilha implícita da chamadas recursivas.

Código dfsrec.h

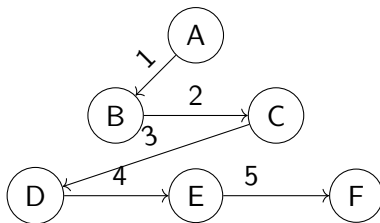
```
int explore(int u) {  
    visited[u] = true;  
    int c = 1;  
    for (int v : g[u]) if (!visited[v])  
        c += explore(v);  
    return c;  
}
```

Busca em profundidade com pilha

Código dfsstack.h

```
int explore(int u) {
    int c = 0;
    stack<int> s;
    s.push(u);
    visited[u] = 1;
    while (!s.empty()) {
        int u = s.top(); s.pop();
        for (int v : g[u]) if (!visited[v]) {
            c++;
            visited[v] = 1;
            s.push(v);
        }
    }
    return c;
}
```

BFS



Busca em largura

Código bfs.h

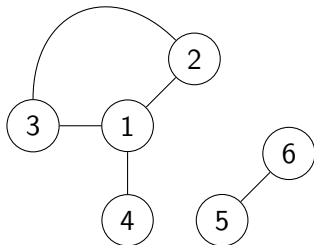
```
int explore(int u) {
    int c = 0;
    queue<int> q;
    q.push(u);
    visited[u] = 1;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : g[u]) if (!visited[v]) {
            c++;
            visited[v] = 1;
            q.push(v);
        }
    }
    return c;
}
```

Resolvendo o problema da contagem de vértices

Código count.cpp

```
#include <bits/stdc++.h>
using namespace std;
vector<vector<int>> g (1e5+15);
vector<bool> visited (1e5+15);
#include "bfs.h"
int main() {
    int n, m; cin >> n >> m;
    while (m--) {
        int u, v; cin >> u >> v;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    int q; cin >> q; while (q--) {
        int u; cin >> u;
        fill(visited.begin(), visited.end(), false);
        cout << explore(u) << "\n";
    }
}
```


Exemplo de contagem de vértices



Entrada	Saída
6 4	4
3 1	4
1 4	4
2 3	4
5 6	2
6	2
1 2 3 4 5 6	