Relatorio: Implementacao de Grafos em C++

Representacoes por Matriz e *Vetor* de Adjacencia; BFS, DFS, Distancias, Componentes e Diametro

Pedro Tuttman Cysne Soares (DRE: 124023584)
Engenharia de Computacao e Informacao

September 25, 2025

Abstract

Este relatorio descreve uma biblioteca de grafos em C++ (nivel graduacao) com duas representacoes: matriz de adjacencia e **vetor de adjacencia** (arrays dinamicos contiguos com `std::vector`). Mostro as decisoes de projeto, um ponto unico de vizinhanca (`forNeighbors`) que permite ter **uma unica** BFS e **uma unica** DFS para ambas as representacoes, e as funcoes pedidas: distancias, pais nas arvores de BFS/DFS, componentes conexas e diametro (exato e aproximado). Incluo tabelas e graficos de tempo e memoria (G1–G6) e discuto os resultados.

Repositorio GitHub

O codigo-fonte (arquivos `main.cpp` e `graph.hpp`) e as tabelas (arquivo `Estudo_de_Casos_Tabelas.xlsx`) estao disponiveis no GitHub :
Link: https://github.com/pedrotuttman/Grafos
OBS: A tabela que apresenta os pais de vértices utilizando os dois métodos de busca está apenas no arquivo do GitHub. Não consta no relatório para evitar poluição visual.

1 Representacoes

1.1 Matriz de adjacencia

Estrutura. `std::vector<std::vector<unsigned char>> mat` com `mat[u][v]` $\in$ `{0,1}`. **Custo.** Memoria $\mathcal{O}(V^2)$. Iterar vizinhos custa $\mathcal{O}(V)$ por vertice (varre a linha inteira). **Quando usar.** Densa, quando operacoes matriciais especificas interessam.

1.2 Vetor de adjacencia (nao e lista encadeada)

Estrutura. `std::vector<std::vector<int>> adj`. Cada `adj[u]` e um *array dinamico contiguo*. **Diferenca para lista encadeada.** Nao existem nos com ponteiros `next`; a alocacao e contigua, com `push_back` em $\mathcal{O}(1)$ amortizado (capacidade dobra quando esgota). **Custo.** Memoria $\mathcal{O}(V + E)$. Iterar vizinhos de `u` custa $\mathcal{O}(\deg(u))$. **Quando usar.** Grafos esparsos; melhor localidade de cache e menor memoria.

2 Carga e abstracao de vizinhos

2.1 Escolha da representacao na carga

O arquivo traz V e E; para cada aresta (u, v) valida, adiciono em `adj` *ou* marco em `mat`, conforme flag `-rep=list|matrix`. Loops e indices invalidos sao ignorados.

Listing 1: Trecho simplificado de definicao do grafo.

```
struct Graph {
  int V; long long E; bool useMatrix;
  std::vector<std::vector<int>> adj;              // vetor de
     adjacencia
  std::vector<std::vector<unsigned char>> mat;   // matriz de
     adjacencia
  // ... construcao em loadFromFile(...)
};
```

2.2 Ponto unico de vizinhanca

Para evitar duplicar logica, toda rotina (BFS, DFS, componentes, diametro) **nao conhece** a representacao; ela so chama `forNeighbors`:

Listing 2: Abstracao comum de vizinhos.

```
template<class F>
inline void forNeighbors(int u, F&& fn) const {
  if(!useMatrix){
    for(int w : adj[u]) fn(w);              // vetor de adjacencia
  }else{
    for(int w=0; w<V; ++w)                  // matriz de
       adjacencia
      if(mat[u][w]) fn(w);
  }
}
```

3 Algoritmos

3.1 BFS unica (pais, niveis, ordem)

Listing 3: BFS com arvore geradora.

```
struct BFSTree { std::vector<int> parent, level, order; };

BFSTree bfsTree(int s) const {
  BFSTree T; T.parent.assign(V,-1); T.level.assign(V,-1);
  std::queue<int> q; T.level[s]=0; q.push(s);
  while(!q.empty()){
    int u=q.front(); q.pop(); T.order.push_back(u);
    forNeighbors(u,[&](int w){
      if(T.level[w]==-1){ T.level[w]=T.level[u]+1; T.parent[w]=u
         ; q.push(w); }
```

```
    });
  }
  return T;
}
```

Por que BFS para distancias? Em grafos nao ponderados, o nivel de BFS e exatamente a menor quantidade de arestas ate cada vertice. Complexidade: $\mathcal{O}(V+E)$ no vetor; $\mathcal{O}(V^2)$ na matriz.

3.2 DFS unica (iterativa, com pais)

Listing 4: DFS iterativa com pilha (sem recursao).

```
std::vector<int> dfsParents(int s) const {
  std::vector<int> parent(V,-1), seen(V,0);
  std::vector<int> st; st.reserve(V);
  st.push_back(s); seen[s]=1;
  while(!st.empty()){
    int u=st.back(); st.pop_back();
    forNeighbors(u,[&](int w){
      if(!seen[w]){ seen[w]=1; parent[w]=u; st.push_back(w); }
    });
  }
  return parent;
}
```

3.3 Distancia entre dois vertices

```
int dist(int u, int v) const { return bfsTree(u).level[v]; } //
   -1 se desconexo
```

3.4 Componentes conexas (com BFS repetidas)

Rodo BFS a partir de cada vertice ainda nao visitado; cada rodada revela uma componente. Tambem computo contagem e tamanhos (min e max). DFS daria as mesmas componentes; mantive BFS por reuso e memoria previsivel.

3.5 Diametro (exato e aproximado)

Definicao. $\max_{u,v} d(u,v)$. **Grafos nao conexos.** Calculo o diametro *de cada componente* e tomo o **maximo** entre eles. **Exato (double sweep).** Em cada componente: (i) BFS a partir de um a acha um b mais distante; (ii) BFS a partir de b acha um c e retorna $d(b,c)$ (excentricidade de b); isso e o diametro da componente. **Aproximado.** Repito o double sweep a partir de k fontes aleatorias e pego o maximo (mais rapido para grafos grandes).

Listing 5: Nucleos para diametro via BFS.

```
std::pair<int,int> farthest(int s) const {
  BFSTree T = bfsTree(s);
  int best=s, d=0;
```

```
  for(int v=0; v<V; ++v) if(T.level[v] > d){ d=T.level[v]; best=v; }
  return {best,d};  // (vertice_mais_longe, distancia)
}

int diameterExact() const {
  int best = 0; std::vector<int> seen(V,0);
  for(int s=0; s<V; ++s) if(!seen[s]){
    BFSTree T = bfsTree(s); int a=s;
    for(int v=0; v<V; ++v) if(T.level[v]>=0){ seen[v]=1; a=v; }
      // marca comp.
    auto fb = farthest(a);
    auto fc = farthest(fb.first);            // distancia = diametro da comp.
    if(fc.second > best) best = fc.second;   // max entre componentes
  }
  return best;
}
```

4 Medicoes: G1–G6 (tempos e memoria)

Tabela 1 compara **vetor** e **matriz**. Onde a matriz e inviavel pela memoria $\mathcal{O}(V^2)$, marcamos **NC**.

Table 1: Comparacao Vetor vs Matriz (G1–G6). "NC" = nao carregado.

Grafo	Mem Vetor (KB)	Mem Matriz (KB)	BFS Vetor (ms)	BFS Matriz (ms)	DFS Vetor (ms)	DFS Matriz (ms)	Status Matriz
G1	1317	97777	1.350	146.656	1.116	3022.399	Carregado
G2	7419	160115	59.834	3992.145	46.215	78141.928	Carregado
G3	6085	**NC**	15.634	**NC**	30.204	**NC**	Nao carregado (V^2 inviavel)
G4	39694	**NC**	879.382	**NC**	637.007	**NC**	Nao carregado (V^2 inviavel)
G5	103726	**NC**	872.498	**NC**	659.979	**NC**	Nao carregado (V^2 inviavel)
G6	250914	**NC**	3345.748	**NC**	2558.142	**NC**	Nao carregado (V^2 inviavel)

Graficos (ajuste os nomes se necessario):

5 Diametros

Diametros obtidos por BFS (*double sweep*). Em grafos nao conexos, reporto o **maximo entre componentes**. Para grafos grandes, usei a versao aproximada com $k = 32$ amostras quando indicado.
Comandos correspondentes:

```
./grafos --in=grafo_i.txt --rep=list --diameter
./grafos --in=grafo_i.txt --rep=list --diameter-approx=32
```

6 Distancias (pares pedidos)

Distancias via BFS (menor numero de arestas). Para G1:

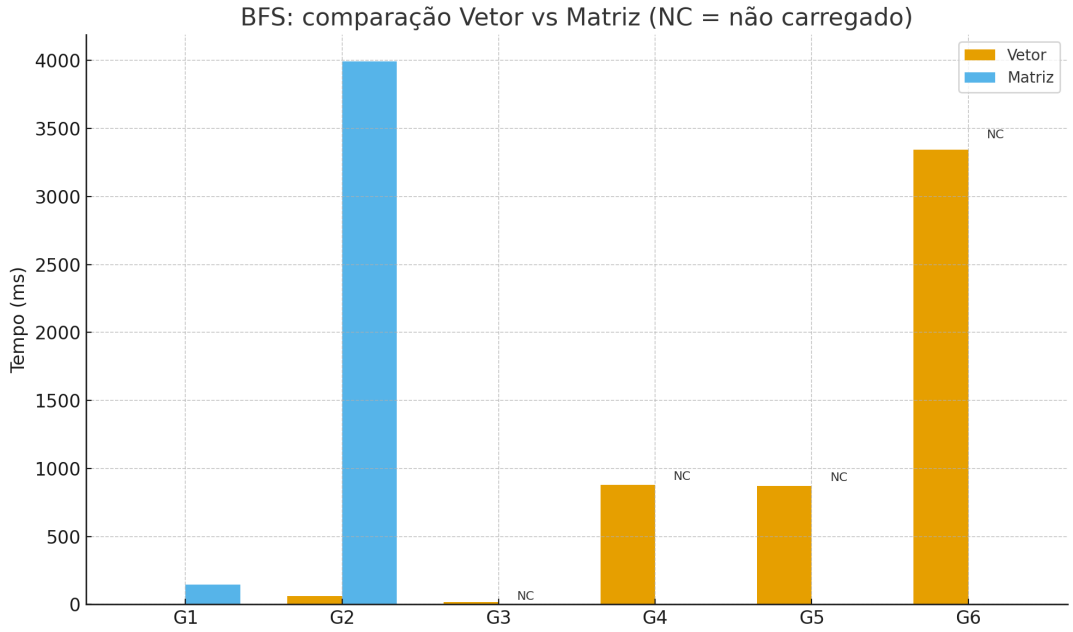

Figure 1: BFS: Vetor vs Matriz (NC = nao carregado).

Table 2: Diametro por grafo. Aprox usa `-diameter-approx=32`.

Grafo	Diametro Exato	Diametro Aprox (k=32)
G1	–	5
G2	–	6
G3	–	7
G4	–	7
G5	–	8
G6	–	9

7 Componentes: como e por que

Usei BFS repetidas (ou DFS daria o mesmo). Para cada vertice ainda nao visto, inicio BFS, marco os alcancados, computo tamanho da componente e atualizo menor/maior. **Motivo da escolha.** Ja tenho BFS pronta e com `forNeighbors`, aproveitando cache/localidade e evitando duplicacao. No G1, por exemplo, o `-stats` retornou 1 componente (tamanho 10000).

8 Analise dos resultados

Memoria. Vetor de adjacencia $\mathcal{O}(V + E)$ foi muito menor (G1: $\sim$1.3 MB) que a matriz $\mathcal{O}(V^2)$ ($\sim$95.5 MB); para G3–G6 a matriz ficou inviavel. **Tempo.** Com vetor, BFS/DFS percorrem apenas vizinhos reais; com matriz, cada linha exige varrer V colunas. Por isso G1 teve BFS $\sim$1.35 ms (vetor) vs 146.656 ms (matriz) e DFS $\sim$1.116 ms vs 3022.399 ms. **BFS x DFS.** Ambas sao $\mathcal{O}(V + E)$ no vetor; a BFS tendeu a ser mais estavel e, para distancias/diametro, e a escolha correta. DFS e util para arvores de profundidade e, na pratica, pode revisitar mais o cache conforme ordem dos vizinhos. **Diametro.** Small-world

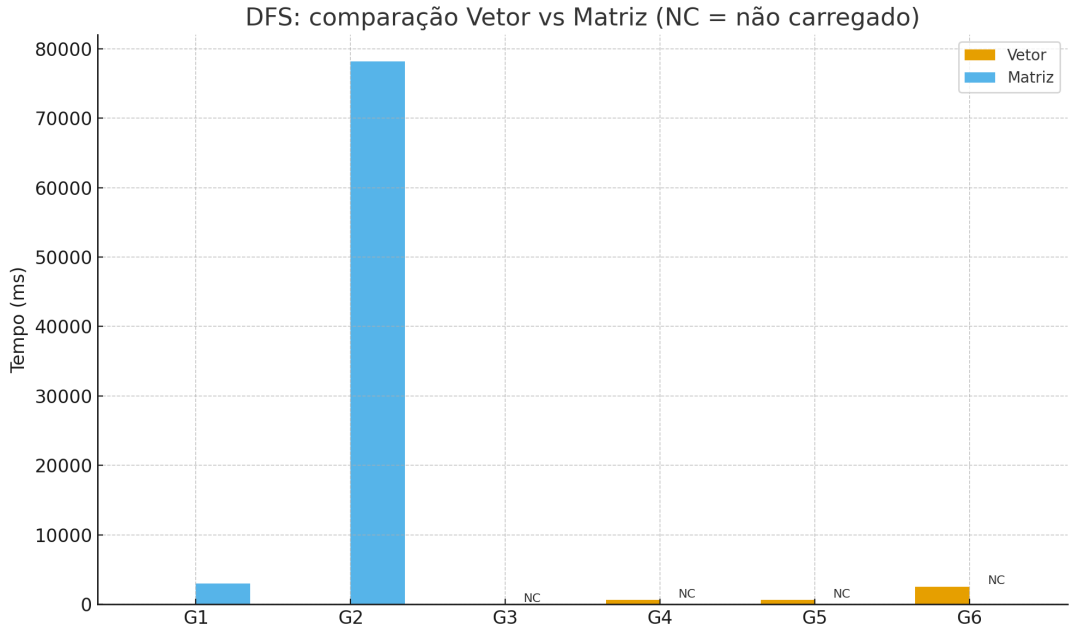

Figure 2: DFS: Vetor vs Matriz (NC = nao carregado).

Table 3: Distancias por BFS no G1.

Par	Distancia d(u,v)	Comando
(10,20)	2	`./grafos -in=grafo_1.txt -rep=list -dist=10,20`
(10,30)	3	`./grafos -in=grafo_1.txt -rep=list -dist=10,30`
(20,30)	1	`./grafos -in=grafo_1.txt -rep=list -dist=20,30`

explica valores baixos (ex.: 5 no G1). Em nao conexos, tomar o maximo entre componentes evita “infinito” e guarda o pior caso real do grafo.

9 Como compilar e executar

Compilar

```
g++ -O2 -std=c++17 main.cpp -o grafos
```

Comandos uteis

```
# escolher representacao
--in=arquivo.txt --rep=list|matrix

# estatisticas (inclui componentes, maior_comp, menor_comp)
--stats=saida.txt

# arvores e pais
--bfs=S --out=arq.txt
--dfs=S --out=arq.txt
```

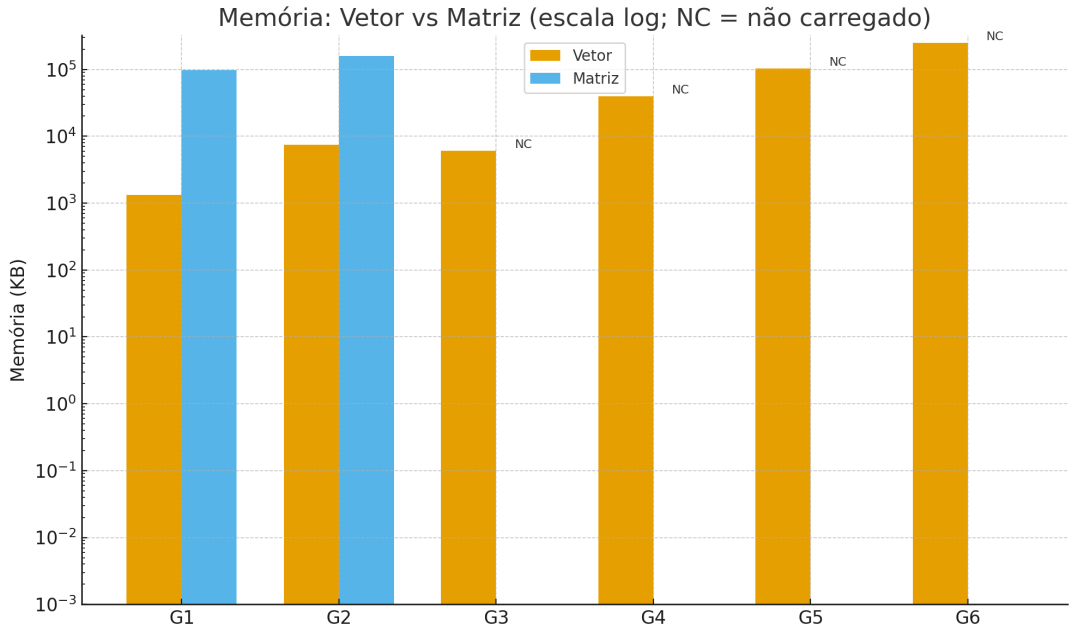

Figure 3: Memoria: Vetor vs Matriz (escala log; NC = nao carregado).

```
--parents=S --from=bfs|dfs --out=arq.txt

# distancias e diametro
--dist=u,v
--diameter
--diameter-approx=K

# benchmarks de tempo
--bench-bfs=K  --bench-dfs=K
```

10 Conclusoes

A estrategia de **vetor de adjacencia** + **forNeighbors** permitiu implementar **uma unica** BFS/DFS que funciona em ambas as representacoes. Em nossos grafos, vetor foi superior em tempo e memoria; matriz tornou-se inviavel para grafos maiores. Distancias e diametro foram obtidos por BFS (propriedade de caminhos minimos); componentes por BFS repetidas.

.