Parte 2 - Análise Crítica: Comparação de Algoritmos de Grafos

Disciplina: Grafos

Grupo: Brandon de Oliveira Hunt, Lucas Rosati Cavalcanti Pereira, Luis Eduardo Vieira Melo, Ronaldo Tavares Souto Maior

---

1. Descrição do Dataset

1.1 Características Gerais

O dataset utilizado representa uma rede de transporte urbano simulada com as seguintes características:

Característica	Valor
Número de vértices (	V
Número de arestas (	E
Tipo	Grafo direcionado e ponderado
Peso mínimo	-30
Peso máximo	100
Peso médio	~48
Arestas positivas	4750 (99.9%)
Arestas negativas	6 (0.1%)
Grau médio	~10 conexões/vértice

1.2 Justificativa do Tamanho

O dataset possui 4756 arestas, bem abaixo do limite sugerido de 200k arestas. Este tamanho foi escolhido por:

1. **Representatividade**: Número suficiente para demonstrar diferenças de desempenho entre algoritmos
2. **Viabilidade**: Permite execução rápida de múltiplas iterações de teste
3. **Clareza**: Tamanho gerenciável para análise e visualização
4. **Controle**: Dataset modificado para garantir casos específicos de teste

1.3 Distribuição de Pesos

O grafo possui dois tipos de arestas:

- **Arestas positivas (99.9%)**: Representam tempos de viagem normais (1-100 minutos)
- **Arestas negativas (0.1%)**: 6 arestas especificamente selecionadas
  - 3 arestas negativas isoladas (-2, -9, -11)
  - 3 arestas formando um ciclo negativo controlado (estacao_999 ↔ estacao_998 ↔ estacao_997)

Esta distribuição foi projetada para testar os limites dos algoritmos, especialmente:

- A capacidade do Dijkstra de ignorar arestas negativas sem falhar
- A capacidade do Bellman-Ford de detectar ciclos negativos

1.4 Modificações no Dataset

Importante: O dataset original continha 255 arestas negativas (5%), o que criava um **ciclo negativo global** acessível de qualquer vértice. Isso impossibilitava testar o caso "peso negativo SEM ciclo negativo".

Solução implementada:

1. **Redução de arestas negativas**: Removemos a maioria das arestas negativas, mantendo apenas 3 isoladas
2. **Ciclo negativo controlado**: Adicionamos artificialmente um ciclo de 3 vértices:

```
estacao_999 --[peso: 10]--> estacao_998
estacao_998 --[peso: 15]--> estacao_997
estacao_997 --[peso: -30]--> estacao_999
Peso total do ciclo: 10 + 15 + (-30) = -5
```

Essa modificação garante:

- Caso 1: Aresta negativa isolada (ex: estacao_287 → estacao_339, peso: -11) **SEM ciclo negativo**
- Caso 2: Ciclo negativo detectável (estacao_999 → estacao_997) **COM ciclo negativo**

Ambos os requisitos da Parte 2 são cumpridos sem ambiguidade.

---

2. Resultados Experimentais

2.1 BFS (Busca em Largura)

Configuração: 3 origens distintas testadas

Resultados obtidos:

- Origens testadas: estacao_229, estacao_233, estacao_36
- Total de vértices visitados: 500 (grafo totalmente conexo)
- Tempo médio: 0.5ms
- Tempo total (3 execuções): 1.51ms

Observações:

- Algoritmo muito rápido, O(V + E)
- Ignora completamente os pesos das arestas
- Útil para encontrar o caminho com menos "saltos"
- Todas as origens alcançaram todos os 500 vértices

2.2 DFS (Busca em Profundidade)

Configuração: 3 origens distintas testadas

Resultados obtidos:

- Origens testadas: estacao_229, estacao_233, estacao_36
- Total de vértices visitados: 500 (todas as execuções)
- Ciclos detectados: **Sim** (em todas as 3 execuções)
- Tempo médio: 7.6ms
- Tempo total (3 execuções): 22.8ms

Observações:

- ~15x mais lento que BFS (devido à detecção de ciclos)
- Detecta ciclos eficientemente
- Também ignora pesos das arestas
- Grafo é fortemente conexo

2.3 Dijkstra

Configuração: 5 pares origem-destino (apenas com pesos não-negativos)

Resultados obtidos:

Par	Distância	Caminho	Tempo
estacao_327 → estacao_57	4	2 vértices	0.12ms
estacao_114 → estacao_71	85	8 vértices	5.98ms
estacao_456 → estacao_279	12	2 vértices	0.15ms

Par	Distância	Caminho	Tempo
estacao_15 → estacao_47	28	2 vértices	0.23ms
estacao_13 → estacao_287	26	2 vértices	0.72ms

- Tempo médio: 1.44ms
- Tempo total (5 execuções): 7.19ms

Observações:

- Muito rápido para grafos de tamanho médio
- Implementação $O(V^2)$ sem heap
- **Ignora arestas com peso negativo** (modificado para não falhar, apenas não usa a aresta)

2.4 Bellman-Ford

Configuração: Casos com peso negativo sem ciclo + caso com ciclo negativo detectado

Caso 1 - Peso negativo SEM ciclo negativo:

- Par: estacao_287 → estacao_339
- Distância encontrada: **11**
- Ciclo negativo detectado: **Não**
- Tempo: 8.42ms

Caso 2 - Ciclo negativo detectado:

- Par testado: estacao_999 → estacao_997
- Ciclo negativo: **Sim**
- Vértices do ciclo: estacao_999 ↔ estacao_998 ↔ estacao_997
- Peso total do ciclo: -5
- Tempo: 276.80ms

Tempo médio: 142.6ms

Tempo total (2 execuções): 285.22ms

Observações:

- ~100x mais lento que Dijkstra (142.6ms vs 1.44ms)
- Único algoritmo que detecta ciclos negativos corretamente
- Complexidade $O(V \times E) = O(503 \times 4756) \approx 2{,}4$ milhões de operações
- Caso 1 é muito mais rápido (aresta direta) vs Caso 2 (precisa explorar todo o grafo)

---

3. Análise de Desempenho

3.1 Comparação de Tempos de Execução

Algoritmo	Tempo Médio	Tempo Total	Complexidade	Suporta Pesos Negativos	Detecta Ciclos Negativos
BFS	0.50ms	1.51ms (3x)	O(V + E)	Não (ignora pesos)	Não
DFS	7.60ms	22.80ms (3x)	O(V + E)	Não (ignora pesos)	Sim (ciclos gerais)
Dijkstra	1.44ms	7.19ms (5x)	$O(V^2)$	Não (ignora arestas <0)	Não
Bellman-Ford	142.61ms	285.22ms (2x)	O(V×E)	Sim	Sim (ciclos negativos)

3.2 Observações sobre Desempenho

1. **BFS** é o mais rápido (0.50ms), mas ignora pesos completamente
2. **Dijkstra** é muito eficiente para pesos não-negativos (1.44ms)
3. **DFS** é ~15x mais lento que BFS (7.60ms) devido à detecção de ciclos
4. **Bellman-Ford** é significativamente mais lento (142.61ms), mas é o único que resolve o problema geral

Fatores de lentidão em relação ao BFS:

- Dijkstra: ~3x mais lento
- DFS: ~15x mais lento
- Bellman-Ford: ~285x mais lento

Fator de lentidão Bellman-Ford vs Dijkstra: ~99x mais lento

---

4. Discussão Crítica

4.1 Quando Usar Cada Algoritmo

BFS (Busca em Largura)

Quando usar:

- Encontrar o caminho com **menor número de saltos** (não menor custo)
- Redes sociais: grau de separação entre pessoas
- Roteamento de pacotes: minimizar número de hops

- Broadcast em redes

Por que é adequado:

- Explora todos os vizinhos antes de ir para o próximo nível
- Garante encontrar o caminho mais curto em grafos não-ponderados
- Muito rápido: O(V + E)

Limitações:

- **Ignora completamente os pesos das arestas**
- Não otimiza custo real
- Inútil quando pesos são importantes

DFS (Busca em Profundidade)

Quando usar:

- Detectar ciclos em grafos direcionados
- Classificação topológica (scheduling, dependências)
- Análise de componentes fortemente conexos
- Resolver labirintos, puzzles

Por que é adequado:

- Explora em profundidade antes de retroceder
- Excelente para problemas de exploração completa
- Uso eficiente de memória (recursão ou stack)

Limitações:

- **Ignora pesos das arestas**
- Não garante caminho mais curto
- Pode percorrer caminhos desnecessariamente longos

Dijkstra

Quando usar:

- **90% dos casos práticos de caminho mínimo**
- GPS/navegação: rotas mais rápidas
- Roteamento de redes (OSPF, IS-IS)
- Jogos: pathfinding de NPCs

- Qualquer grafo com **garantia de pesos não-negativos**

Por que é adequado:

- Rápido: $O(V^2)$ sem heap, $O((V+E)\log V)$ com heap
- Simples de implementar
- Sempre encontra o caminho ótimo (se não houver pesos negativos)

Limitações CRÍTICAS:

- **FALHA com pesos negativos** (dá resultado ERRADO, não detecta)
- Assume que visitado = otimizado (incorreto com pesos negativos)
- Não detecta ciclos negativos

Exemplo de falha:

```
A --(-10)⟶ B --(-5)⟶ C
A --(1)-⟶ C

Dijkstra escolhe: A → C (custo 1)
Correto seria: A → B → C (custo -15)
```

Bellman-Ford

Quando usar:

- Grafos com **pesos negativos**
- **Detecção de ciclos negativos** (fraude, arbitragem)
- Sistemas financeiros: detectar oportunidades de arbitragem
- Análise de vulnerabilidades em sistemas de crédito
- Quando não há garantia de pesos não-negativos

Por que é adequado:

- Único algoritmo que **detecta ciclos negativos**
- Sempre encontra o caminho ótimo (se não houver ciclo negativo)
- Relaxa todas as arestas V-1 vezes, garantindo correção

Limitações:

- **Muito lento**: $O(VE)$ = até 1000x mais lento que Dijkstra
- Para grafos grandes ($|E| > 100k$), pode ser inviável

- Se não há pesos negativos, Dijkstra é sempre melhor

Uso crítico:

```
Detecção de ciclo negativo:
  A → B → C → A com peso total -10

Interpretação: "Ganho infinito possível"
  - Em finanças: arbitragem
  - Em sistemas: fraude/bug
  - Em jogos: exploit
```

4.2 Limites do Design de Pesos

Pesos Negativos: Realismo vs. Abstração

No contexto de transporte urbano:

Não fazem sentido físico

- Tempo de viagem não pode ser negativo
- Não existe "voltar no tempo" ao pegar uma rota
- Distâncias são sempre positivas

Como abstração, podem representar:

1. **Descontos/créditos**: Programas de fidelidade que dão créditos
2. **Incentivos**: Rotas subsidiadas que "economizam" custo total
3. **Atalhos virtuais**: Representação abstrata de economias no sistema
4. **Conversão de moedas**: Em sistemas financeiros

Exemplo prático:

```
Sistema de transporte com programa de pontos:
  - Rota normal A → B: 30 minutos
  - Rota com desconto A → B: -20 minutos (ganha 20 min de crédito)

Interpretação: Pegar esta rota dá 20 minutos de crédito para usar depois
```

O Problema do Ciclo Negativo

No dataset testado:

```
Ciclo detectado: estacao_481 → ... → estacao_470 → ... → estacao_481
Peso total: -75 minutos

Implicação: Cada volta no ciclo "economiza" 75 minutos
```

Por que isso é um problema:

1. **Matematicamente**: Caminho ótimo = -∞ (impossível)
2. **Sistemas reais**: Indica **erro de design** ou **vulnerabilidade**
3. **Aplicações práticas**:
   - Finanças: arbitragem (ganho sem risco)
   - Jogos: exploit (ganho infinito de recursos)
   - Sistemas de crédito: fraude

Solução em sistemas reais:

- Bellman-Ford detecta o ciclo
- Sistema deve **bloquear** essas rotas
- Rebalancear pesos para eliminar ciclos negativos

4.3 Escolha do Algoritmo: Fluxograma de Decisão

```
Seu grafo tem pesos nas arestas?
  └─ NÃO → Use BFS (menor número de saltos) ou DFS (detecção de ciclos)
  └─ SIM → Continue...

Você precisa encontrar caminho mínimo?
  └─ NÃO → Use DFS para exploração
  └─ SIM → Continue...

Todos os pesos são não-negativos?
  └─ SIM → Use DIJKSTRA (rápido e eficiente)
  └─ NÃO → Use BELLMAN-FORD

Você precisa detectar ciclos negativos?
  └─ SIM → Use BELLMAN-FORD (único que detecta)
```

---

5. Conclusões

5.1 Principais Aprendizados

1. **Não existe algoritmo universal**: Cada algoritmo tem seu nicho
2. **Performance vs. Generalidade**: Dijkstra é rápido mas limitado; Bellman-Ford é geral mas lento
3. **Pesos negativos são raros** mas exigem algoritmos específicos
4. **Detecção de ciclos negativos** é essencial para validação de sistemas

5.2 Recomendações Práticas

Para aplicações reais:

1. **GPS/Navegação**: Dijkstra (distâncias são sempre positivas)
2. **Redes de computadores**: Dijkstra (latências são positivas)
3. **Sistemas financeiros**: Bellman-Ford (preços podem ser negativos, precisa detectar arbitragem)
4. **Redes sociais**: BFS (grau de separação, não há pesos)
5. **Análise de dependências**: DFS (detecção de ciclos, ordenação topológica)

Regra de ouro:

> "Use Dijkstra se puder, Bellman-Ford se precisar."

5.3 Dataset Sintético vs. Real

Vantagens do dataset sintético:

- Permite testar casos extremos (ciclos negativos)
- Controle sobre distribuição de pesos
- Demonstra claramente as diferenças entre algoritmos

Limitações:

- Pesos negativos não refletem cenários reais de transporte
- Estrutura pode não capturar propriedades de redes reais (small-world, scale-free)

Para trabalhos futuros:

- Usar datasets reais (GTFS, OpenStreetMap)
- Testar com grafos maiores ($|E| > 100k$)
- Implementar otimizações (heap para Dijkstra, early termination)
