PDF Resposta: Aplicações de BFS e DFS

Pedro Schneider

1 Exercício 1: BFS - Encontrar Caminho Mais Curto

1.1 Parte A: Implementação Manual

1. **Desenhe o grafo** representando o campus com base nas conexões fornecidas.

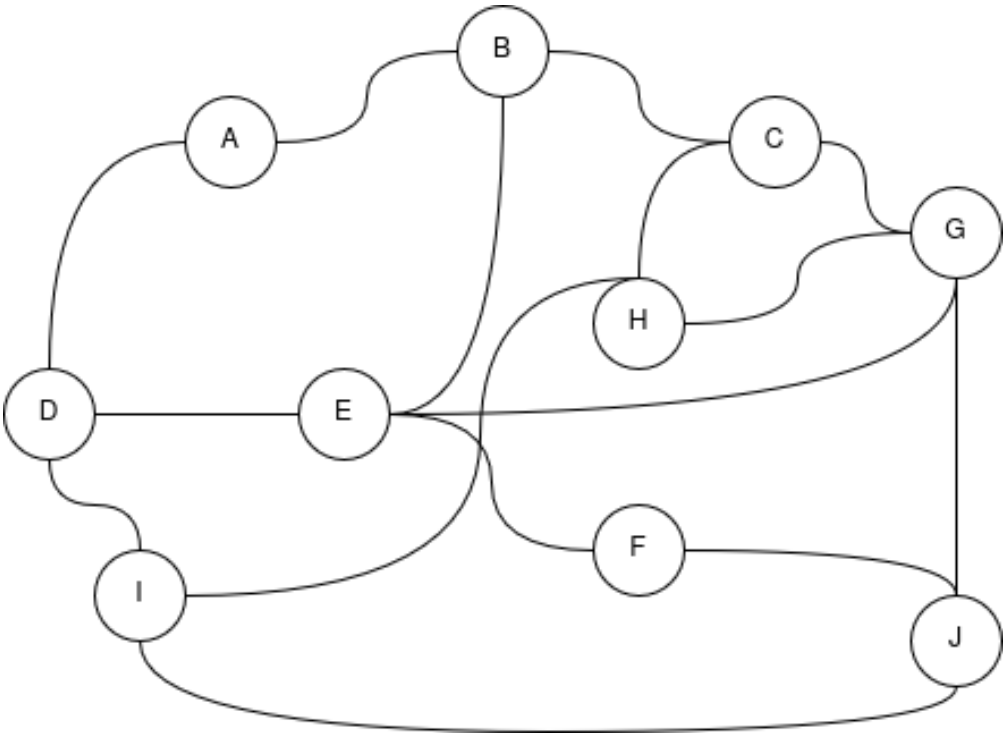

Figura 1: Grafo do campus

2. **Execute BFS manualmente** partindo da Portaria Principal (A) para encontrar o caminho mais curto até o Centro de Convivência (J):

   - Use a tabela abaixo para registrar cada passo
   - Registre o estado da fila a cada iteração
   - Marque as cores dos vértices (BRANCO, CINZA, PRETO)

Tabela 1: Execução do algoritmo BFS

Passo	Vértice atual	Fila	Vizinhos Descobertos	Distâncias Atualizadas
0	-	[A]	-	A.d = 0
1	A	[B,D]	B,D	B.d = 1, D.d = 1
2	B	[D,C,E]	C,E	C.d = 2, E.d = 2
3	D	[C,E,I]	I	I.d = 2
4	C	[E,I,G,H]	G,H	G.d = 3, H.d = 3
5	E	[I,G,H,F]	F	F.d = 3
6	I	[G,H,F,J]	J	J.d = 3
7	G	[H,F,J]	-	-
8	H	[F,J]	-	-
9	F	[J]	-	-
10	J	[]	-	-

3. **Identifique o caminho** de A para J e sua distância mínima.

   Caminho: A-D-I-J

   Distância mínima: 3

1.2 Parte B: Análise Comparativa

Execute BFS também partindo de A para encontrar os caminhos mais curtos para:

- Cantina (G)

  Caminho: A-B-C-G

  Distância mínima: 3

- Estacionamento (I)

  Caminho: A-D-I

  Distância mínima: 2

- Laboratório de Informática (F)

  Caminho: A-B-E-F

Distância mínima: 3

R.: O Estacionamento (I) é o mais próximo.

1.3 Parte C: Modificação do Algoritmo

Modifique o algoritmo BFS para:

1. **Parar** quando encontrar o vértice destino J
2. **Reconstruir** o caminho completo usando um array de predecessores
3. **Retornar** tanto a distância quanto a sequência de vértices

Escreva o pseudocódigo da função modificada: BFS_CaminhoMaisCurto(G, origem, destino)

Resposta

Listing 1: Pseudocódigo BFS_CaminhoMaisCurto(G, origem, destino)

```
BFS_CaminhoMaisCurto(G, origem, destino):
   para cada vertice v em G.V:
      v.cor <- branco
      v.d <- infinito
      v.pred <- nulo

   origem.cor <- cinza
   origem.d <- 0
   origem.pred <- nulo

   criar fila Q
   enfileirar(Q, origem)

   enquanto Q nao estiver vazia:
      u <- desenfileirar(Q)
      se u == destino:
         interromper laco // Encontrou o destino

      para cada v em Adj[u]:
         se v.cor == branco:
            v.cor <- cinza
            v.d <- u.d + 1
            v.pred <- u
            enfileirar(Q, v)

      u.cor <- preto

   // Reconstruir caminho
   caminho <- lista vazia
   v <- destino
   enquanto v != nulo:
      inserir v no inicio de caminho
      v <- v.pred

   retornar (caminho, destino.d)
```

1.4 Parte D: Questões Teóricas

1. Por que o BFS garante encontrar o caminho mais curto em grafos não-ponderados?

   Resposta O BFS explora o grafo em camadas de distância crescente a partir do vértice de origem, visitando primeiro todos os vértices a uma aresta de distância ($d = 1$), depois os que estão a duas arestas ($d = 2$), e assim sucessivamente. Como todas as arestas têm o mesmo peso (1), a primeira vez que um vértice é alcançado pelo BFS corresponde necessariamente ao menor número possível de arestas entre ele e a origem. Portanto, a distância registrada é a do caminho mais curto.

2. Se as arestas tivessem pesos diferentes, BFS ainda funcionaria? Justifique.

   Resposta Não, a BFS não funcionaria corretamente em grafos com arestas de pesos diferentes, pois ela assume que todas as arestas têm o mesmo custo e explora os vértices em ordem de número de arestas, não de peso total; quando os pesos variam, um caminho com menos arestas pode ter custo maior que outro com mais arestas, fazendo necessário o uso de outro algoritmo.

2 Exercício 2: DFS - Detecção de Ciclos

2.1 Parte A: Representação e Análise

1. **Desenhe o grafo direcionado** com as dependências.

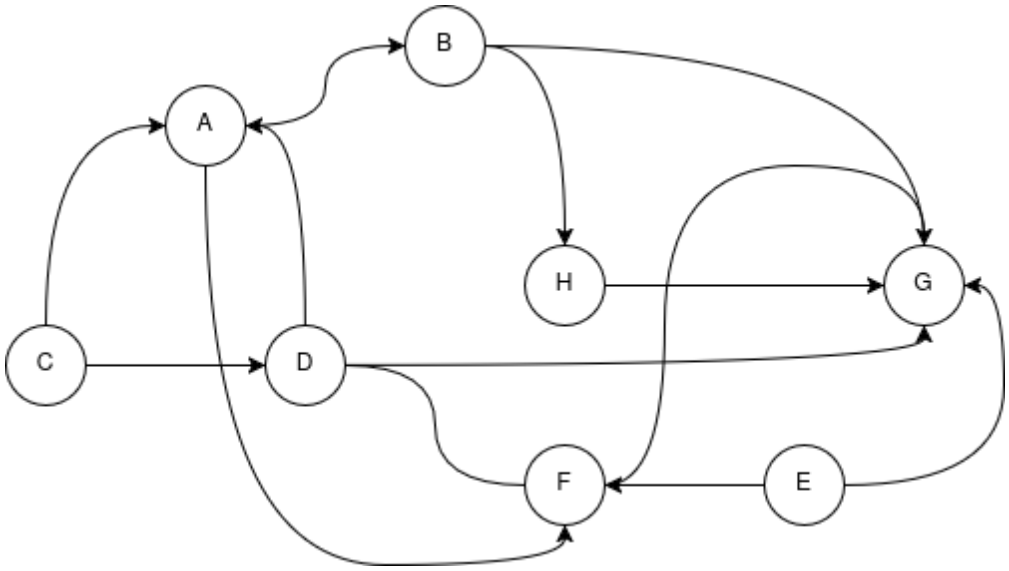

Figura 2: Grafo software

2. **Identifique visualmente** se existem ciclos óbvios.

   Resposta Não há.

3. **Liste** todas as dependências diretas de cada módulo.

   - A depende de **B, F**
   - B depende de **G, H**
   - C depende de **A, D**
   - D depende de **A, G**
   - E depende de **F, G**
   - F depende de **G, F**
   - G depende de **-**
   - H depende de **G**

2.2 Parte B: Execução Manual do DFS

Execute DFS manualmente para detectar ciclos.

- Use a tabela abaixo para registrar tempos de descoberta e finalização

Tabela 2: Execução do algoritmo BFS

Vértice]	Cor	Tempo Descoberta	Tempo Finalização	Predecessor
A	PRETO	1	10	NULL
B	PRETO	2	7	A
D	PRETO	11	14	NULL
C	PRETO	12	13	C
E	PRETO	15	16	NULL
I	PRETO	8	9	A
G	PRETO	4	5	H
H	PRETO	3	6	B

- Registre cada back edge encontrada (aresta que vai para um vértice CINZA):

  Back edge: Não há

  Ciclo detectado: Não há

- Classifique todas as arestas
  - **Tree edges** (arestas da árvore DFS): [A,..,H]
  - **Back edges** (causam ciclos): -
  - **Forward edges** (para descendentes): -
  - **Cross edges** (entre subárvores): -

2.3 Parte C: Implementação do Detector de Ciclos

Modifique o algoritmo DFS para:

1. Retornar **true** se encontrar pelo menos um ciclo
2. **Armazenar informações** sobre todos os ciclos encontrados
3. Para cada ciclo, identifique os **vértices que o compõem**

Escreva o pseudocódigo da função modificada: DetectaCiclo_DFS(G)

Resposta

Listing 2: Pseudocódigo DetectaCiclo_DFS(G)

```
DetectaCiclo_DFS(G):
   ciclos <- lista vazia # Armazena todos os ciclos encontrados

   para cada v em G.V:
      v.cor <- branco
      v.pred <- nulo

   para cada v em G.V:
      se v.cor == branco:
         DFS_Visit(v, ciclos)

   retornar ciclos

DFS_Visit(u, ciclos):
   u.cor <- cinza

   para cada v em Adj[u]:
      se v.cor == branco:
         v.pred <- u
         DFS_Visit(v, ciclos)
      senao se v.cor == cinza e v != u.pred:
         # Encontrou um ciclo
         ciclo <- lista vazia
         w <- u
         enquanto w != v:
            inserir w no inicio do ciclo
            w <- w.pred
         inserir v no inicio do ciclo
         inserir ciclo em ciclos

   u.cor <- preto
```

2.4 Parte D: Análise de Resultados

1. **Quantos ciclos** existem no grafo de dependências?

   Resposta: 0, o grafo é acíclico

2. **Quais módulos** estão envolvidos em dependências circulares?

   Resposta: Nenhum

3. **Proponha uma solução** para quebrar os ciclos mantendo o máximo de dependências possível.

   Resposta: Se houvessem ciclos, seria necessário extrair funcionalidades comuns para um módulo base compartilhado

2.5 Parte E: Questões Teóricas

1. Por que back edges em DFS indicam ciclos em grafos direcionados?

   Resposta: Em um grafo direcionado, um back edge é uma aresta que conecta um vértice a um de seus ancestrais na árvore de DFS (ou seja, um vértice ainda em processo de visita, cor cinza). Isso indica que existe um caminho de volta para um vértice já visitado, formando assim um ciclo. Portanto, a presença de back edges é exatamente a condição que identifica ciclos em grafos direcionados.

2. Como você modificaria o algoritmo para grafos não-direcionados?

   Resposta: Em grafos não-direcionados, uma DFS também pode detectar ciclos, mas é preciso ignorar a aresta que leva ao pai (o vértice de onde chegamos). Ou seja, um ciclo existe se encontrarmos uma aresta para um vértice cinza que não seja o predecessor direto. Essa modificação evita contar a aresta bidirecional do grafo como um ciclo falso.

3. Qual é a diferença entre detectar ciclos e encontrar componentes fortemente conexos?

   Resposta: Detectar ciclos verifica apenas se existe pelo menos um caminho fechado em um grafo (um ciclo), sem se preocupar com o restante da estrutura. Encontrar componentes fortemente conexos em grafos direcionados identifica subconjuntos máximos de vértices em que todo vértice pode alcançar todo outro vértice do mesmo conjunto. Assim,

ciclos podem existir dentro de componentes fortemente conexos, mas nem todos os ciclos formam uma componente fortemente conexa.