Avaliação Continuada 6

Questão 1

Enunciado: Considere o grafo abaixo:

Um teste de mesa consiste em executar manualmente os passos do algoritmo, anotando os estados das variáveis ao longo da execução. Realize um teste de mesa do algoritmo de busca em largura no grafo acima, iniciando no nó **a**. Forneça a distância e o predecessor de cada nó, conforme obtido através do algoritmo. A distância de um nó é o menor número de arestas do nó até o nó inicial **a**. O predecessor de um nó é o pai deste nó na árvore de busca em largura com raiz no nó **a**.

Obs.: Um mesmo grafo pode ter várias árvores de busca em largura corretas. Isto significa que cada aluno pode obter predecessores distintos para os nós. A distância, por outro lado, é única. Ou seja, todos os alunos devem obter as mesmas distâncias.

Resposta:

Para realizar o teste de mesa do algoritmo BFS (Breadth-First Search), vamos seguir os passos:

```
Algoritmo BFS:

1. Inicializar: d[a] = 0, π[a] = NIL, Q = {a}
2. Enquanto Q não estiver vazia:
3.     u = Dequeue(Q)
4.     Para cada vizinho v de u:
5.         Se v não foi visitado (d[v] = ∞):
6.             d[v] = d[u] + 1
7.             π[v] = u
```

```
8.              Enqueue(Q, v)
```

Nó	Distância (d)	Predecessor (π)
a	0	NIL
b	1	a
c	1	a
d	2	b
e	2	c
f	3	d

Execução passo a passo:

1. **Inicialização:** d[a] = 0, π[a] = NIL, Q = [a]
2. **Iteração 1:** u = a, vizinhos: b, c → d[b] = 1, π[b] = a, d[c] = 1, π[c] = a, Q = [b, c]
3. **Iteração 2:** u = b, vizinhos: a, d → d[d] = 2, π[d] = b, Q = [c, d]
4. **Iteração 3:** u = c, vizinhos: a, e → d[e] = 2, π[e] = c, Q = [d, e]
5. **Iteração 4:** u = d, vizinhos: b, f → d[f] = 3, π[f] = d, Q = [e, f]
6. **Iteração 5:** u = e, vizinhos: c (já visitado), Q = [f]
7. **Iteração 6:** u = f, vizinhos: d (já visitado), Q = []

Questão 2

Enunciado: Considere o grafo abaixo:

Um teste de mesa consiste em executar manualmente os passos do algoritmo, anotando os estados das variáveis ao longo da execução. Realize um teste de mesa do algoritmo de busca em profundidade no grafo acima, iniciando no nó **a**. Forneça (i) o predecessor de cada nó, (ii) o instante **d** que cada nó foi descoberto, e (iii) o instante **f** em que o nó foi finalizado, conforme obtido através da

execução do algoritmo recursivo visto em aula. O predecessor de um nó é o pai deste nó na árvore de busca em profundidade com raiz no nó **a**. Os nós vizinhos devem ser explorados em ordem alfabética, ou seja, todos os alunos devem obter a mesma árvore de busca em profundidade.

Resposta:

Para realizar o teste de mesa do algoritmo DFS (Depth-First Search), vamos seguir os passos:

```
Algoritmo DFS:
  1. Para cada vértice u ∈ V:
  2.     cor[u] = BRANCO
  3.     π[u] = NIL
  4. tempo = 0
  5. Para cada vértice u ∈ V:
  6.     Se cor[u] = BRANCO:
  7.         DFS-VISIT(u)
```

Nó	Predecessor (π)	Tempo Descoberta (d)	Tempo Finalização (f)
a	NIL	1	12
b	a	2	11
c	b	3	10
d	c	4	9
e	d	5	8
f	e	6	7

Execução passo a passo:

1. **DFS-VISIT(a):** d[a] = 1, explorar vizinhos em ordem alfabética: b
2. **DFS-VISIT(b):** d[b] = 2, explorar vizinhos: a (já visitado), c
3. **DFS-VISIT(c):** d[c] = 3, explorar vizinhos: b (já visitado), d

4. **DFS-VISIT(d):** d[d] = 4, explorar vizinhos: c (já visitado), e
5. **DFS-VISIT(e):** d[e] = 5, explorar vizinhos: d (já visitado), f
6. **DFS-VISIT(f):** d[f] = 6, explorar vizinhos: e (já visitado)
7. **Finalização:** f[f] = 7, f[e] = 8, f[d] = 9, f[c] = 10, f[b] = 11, f[a] = 12

Questão 3

Enunciado: Definição: Um grafo $G = (V, E)$ é dito bipartido se o seu conjunto de vértices V pode ser particionado em dois subconjuntos X e Y de modo que para toda aresta (u, v) temos que ou $u \in X$ e $v \in Y$, ou $u \in Y$ e $v \in X$.

Construa um algoritmo que dado um grafo G não orientado verifica se G é bipartido ou não. Forneça a complexidade de tempo do algoritmo proposto.

Resposta:

Um grafo é bipartido se e somente se é 2-colorível (pode ser colorido com 2 cores sem que vértices adjacentes tenham a mesma cor). Podemos usar uma modificação do BFS para verificar isso.

```
Algoritmo: VerificarBipartido(G)

Entrada: Grafo não orientado G = (V, E)

Saída: Verdadeiro se G é bipartido, Falso caso contrário

  1. Para cada vértice v ∈ V:
  2.     cor[v] = NÃO_COLORIDO
  3. Para cada vértice v ∈ V:
  4.     Se cor[v] = NÃO_COLORIDO:
  5.         cor[v] = AZUL
  6.         Q = {v}
  7.         Enquanto Q não estiver vazia:
  8.             u = Dequeue(Q)
```

```
            Para cada vizinho w de u:
                Se cor[w] = NÃO_COLORIDO:
                    cor[w] = cor oposta à cor[u]
                    Enqueue(Q, w)
                Senão se cor[w] = cor[u]:
                    Retorne FALSO
Retorne VERDADEIRO
```

Análise de Complexidade:

Tempo: O(|V| + |E|)

- Cada vértice é visitado no máximo uma vez
- Cada aresta é examinada no máximo duas vezes
- As operações de fila são O(1) amortizado

Espaço: O(|V|) para a fila e o array de cores

Questão 4

Enunciado: Projete um algoritmo que determina se um grafo não orientado $G(V, E)$ contém ciclo. O algoritmo projetado deve ter complexidade de tempo $O(|V|)$.

Resposta:

Para detectar ciclos em um grafo não orientado com complexidade O(|V|), podemos usar uma modificação do DFS. Um grafo não orientado tem ciclo se e somente se durante o DFS encontramos uma aresta de retorno (back edge).

Algoritmo: DetectarCiclo(G)

```
Entrada: Grafo não orientado G = (V, E)
Saída: Verdadeiro se G contém ciclo, Falso caso contrário

1. Para cada vértice v ∈ V:
2.     visitado[v] = FALSO
3.     pai[v] = NIL
4. Para cada vértice v ∈ V:
5.     Se visitado[v] = FALSO:
6.         Se DFS-Ciclo(v, pai[v]) = VERDADEIRO:
7.             Retorne VERDADEIRO
8. Retorne FALSO
```

Função: DFS-Ciclo(v, pai_v)

```
1. visitado[v] = VERDADEIRO
2. Para cada vizinho u de v:
3.     Se visitado[u] = FALSO:
4.         pai[u] = v
5.         Se DFS-Ciclo(u, v) = VERDADEIRO:
6.             Retorne VERDADEIRO
7.     Senão se u ≠ pai_v:
8.         Retorne VERDADEIRO
9. Retorne FALSO
```

Análise de Complexidade:

Tempo: O(|V|)

- Cada vértice é visitado no máximo uma vez
- Cada aresta é examinada no máximo duas vezes
- Como o grafo é não orientado, |E| ≤ |V| - 1 para grafos acíclicos
- Se |E| > |V| - 1, o grafo certamente tem ciclo

Espaço: O(|V|) para os arrays visitado e pai

Questão 5

Enunciado: Implemente uma busca em profundidade (DFS) usando uma pilha (de forma a eliminar a recursão). O seu algoritmo deverá devolver uma floresta de busca em profundidade representada por um vetor π e deve executar em tempo O(V+E). Você pode utilizar as sub-rotinas de uma pilha como caixas-pretas: CriarPilha(Q), Topo(Q), Desempilhar(Q), Empilhar(Q, v).

Resposta:

Vamos implementar o DFS iterativo usando uma pilha para eliminar a recursão:

```
Algoritmo: DFS-Iterativo(G)

Entrada: Grafo G = (V, E)

Saída: Vetor π representando a floresta de busca em profundidade

  1. Para cada vértice v ∈ V:
  2.     visitado[v] = FALSO
  3.     π[v] = NIL
  4. Para cada vértice v ∈ V:
  5.     Se visitado[v] = FALSO:
  6.         P = CriarPilha()
  7.         Empilhar(P, v)
  8.         visitado[v] = VERDADEIRO
  9.         Enquanto P não estiver vazia:
 10.             u = Topo(P)
 11.             Se existe vizinho não visitado de u:
 12.                 w = próximo vizinho não visitado de u
```

```
                Empilhar(P, w)
                visitado[w] = VERDADEIRO
                π[w] = u
            Senão:
                Desempilhar(P)
Retorne π
```

Análise de Complexidade:

Tempo: O(|V| + |E|)

- Cada vértice é empilhado e desempilhado no máximo uma vez
- Cada aresta é examinada no máximo duas vezes
- As operações de pilha são O(1)

Espaço: O(|V|) para a pilha e os arrays visitado e π

Vantagens da versão iterativa:

- Evita estouro de pilha para grafos muito profundos
- Mais eficiente em termos de memória
- Mais fácil de otimizar