SUMÁRIO

LISTA DE CÓDIGOS

1 Grafos utilizando lista de adjacência

```c
p_grafo CriarGrafo(int n) {
    /*
     * Funcão CriarGrafo
     * Cria e inicializa um grafo com n vértices usando lista de adjacência.
     * Cada vértice terá uma lista (inicialmente vazia) para armazenar seus vizinhos.
     */
    int i;  // Variável para iteracão
    // Aloca espaco para a estrutura principal do grafo
    p_grafo g = malloc(sizeof(Grafo));
    // Define o número de vértices do grafo
    g->n = n;
    // Aloca um vetor de ponteiros para listas de adjacência, um para cada vértice
    g->adjacencia = malloc(n * sizeof(p_no));
    // Inicializa todas as listas de adjacência como vazias (NULL)
    for (i = 0; i < n; i++) {
        g->adjacencia[i] = NULL;
    }
    // Retorna o ponteiro para o grafo criado
    return g;
}
```

Código 1: Função criar grafo

```c
/*
 * Funcão LiberaLista
 * Libera recursivamente todos os nós de uma lista de adjacência.
 * Utilizada para liberar a memória de cada lista de vizinhos de um vértice.
 */
void LiberaLista(p_no lista) {
    // Se a lista não está vazia
    if (lista != NULL) {
        // Libera recursivamente o próximo nó da lista
        LiberaLista(lista->prox);
        // Libera o nó atual
        free(lista);
    }
}
```

Código 2: Destroi lista

```c
void DestroiGrafo(p_grafo g) {
    // Funcão que libera toda a memória alocada para o grafo.
    int i;
    // Para cada vértice, libera a lista de adjacência (os vizinhos).
    for (i = 0; i < g->n; i++) {
        LiberaLista(g->adjacencia[i]);
    }
    // Libera o vetor de ponteiros para as listas de adjacência.
    free(g->adjacencia);
    // Libera a estrutura principal do grafo.
    free(g);
}
```

Código 3: Destroi grafo

```c
p_no InsereNaLista(p_no lista, int v, int peso) {
    // Funcão que insere um novo elemento (v) no início da lista de adjacência.
```

```
    // Aloca memória para um novo nó da lista de adjacência.
    p_no novo = malloc(sizeof(No));
    // Define o valor do vértice armazenado neste nó.
    novo->v = v;
    // Define o peso da aresta
    novo->peso = peso;
    // Faz o novo nó apontar para o início da lista recebida (inserção no início).
    novo->prox = lista;
    // Retorna o ponteiro para o novo nó, que agora é o início da lista.
    return novo;
}
```

Código 4: Insere na lista

```
/*
 * Funcão InsereAresta
 * Insere uma aresta não direcionada entre os vértices u e v com peso padrão 1.
 * Para lista de adjacência: adiciona v na lista de u e u na lista de v.
 */
void InsereAresta(p_grafo g, int u, int v) {
    InsereArestaPonderada(g, u, v, 1);
}
```

Código 5: Insere aresta

```
/*
 * Funcão InsereArestaPonderada
 * Insere uma aresta não direcionada entre os vértices u e v com peso especificado.
 */
void InsereArestaPonderada(p_grafo g, int u, int v, int peso) {
    // Adiciona u na lista de adjacência de v (v passa a ter u como vizinho)
    g->adjacencia[v] = InsereNaLista(g->adjacencia[v], u, peso);
    // Adiciona v na lista de adjacência de u (u passa a ter v como vizinho)
    g->adjacencia[u] = InsereNaLista(g->adjacencia[u], v, peso);
}
```

Código 6: Insere aresta ponderada

```
/*
 * Funcão RemoveDaLista
 * Remove o nó com valor v da lista de adjacência.
 * Retorna o ponteiro para o início da lista após a remocão.
 * Utiliza recursão para encontrar e remover o nó desejado.
 */
p_no RemoveDaLista(p_no lista, int v) {
    p_no proximo;
    // Caso base: lista vazia, nada para remover
    if (lista == NULL) {
        return NULL;
    }
    // Se o nó atual tem o valor v, remove este nó
    else if (lista->v == v) {
        proximo = lista->prox;  // Guarda o próximo nó
        free(lista);            // Libera o nó atual
        return proximo;         // Retorna o próximo nó como novo início da lista
    }
    // Caso contrário, continua procurando na lista
    else {
```

```
        lista->prox = RemoveDaLista(lista->prox, v);  // Chama recursivamente para o próximo
        return lista;                                 // Retorna o início da lista (que não mudou)
    }
}
```

Código 7: Remove da lista

```
// Funcão que remove uma aresta não direcionada entre os vértices u e v.
// Ou seja, remove v da lista de adjacência de u e u da lista de adjacência de v.
void RemoveAresta(p_grafo g, int u, int v) {
    // Remove v da lista de adjacência de u.
    g->adjacencia[u] = RemoveDaLista(g->adjacencia[u], v);
    // Remove u da lista de adjacência de v.
    g->adjacencia[v] = RemoveDaLista(g->adjacencia[v], u);
    // Após essas operacões, a aresta entre u e v deixa de existir no grafo.
}
```

Código 8: Remove aresta

```
// Funcão que verifica se existe uma aresta entre os vértices u e v.
// Retorna 1 se existe, 0 caso contrário.
int TemAresta(p_grafo g, int u, int v) {
    p_no t;  // Ponteiro para percorrer a lista de adjacência de u.
    // Percorre a lista de adjacência de u.
    for (t = g->adjacencia[u]; t != NULL; t = t->prox) {
        // Se encontrar o vértice v na lista, retorna 1 (aresta existe).
        if (t->v == v) {
            return 1;
        }
    }
    // Se não encontrar v na lista, retorna 0 (aresta não existe).
    return 0;
}
```

Código 9: Tem aresta

```
// Funcão que lê um grafo da entrada padrão.
// Espera receber o número de vértices (n) e arestas (m), seguido de m pares de vértices representando as arestas.
p_grafo LeGrafo() {
    int n, m, i, u, v;  // n: número de vértices, m: número de arestas, u/v: vértices da aresta, i: contador
    p_grafo g;          // Ponteiro para o grafo

    // Lê o número de vértices e arestas
    scanf("%d %d", &n, &m);

    // Cria e inicializa o grafo com n vértices
    g = CriarGrafo(n);

    // Para cada aresta, lê os vértices u e v e insere a aresta no grafo
    for (i = 0; i < m; i++) {
        scanf("%d %d", &u, &v);
        InsereAresta(g, u, v);
    }

    // Retorna o ponteiro para o grafo preenchido
    return g;
```

```
}
```

Código 10: Lê grafo

```
// Funcão que imprime todas as arestas do grafo.
// Para cada vértice, percorre sua lista de adjacência e imprime cada conexão.
void ImprimeArestas(p_grafo g) {
    int u;   // Variável para iterar sobre os vértices
    p_no t;  // Ponteiro para percorrer a lista de adjacência de cada vértice
    // Para cada vértice u do grafo
    for (u = 0; u < g->n; u++) {
        // Percorre a lista de adjacência de u
        for (t = g->adjacencia[u]; t != NULL; t = t->prox) {
            // Imprime a aresta entre u e t->v
            printf("{%d %d}\n", u, t->v);
        }
    }
    // Ao final, todas as arestas do grafo são exibidas no formato {u v}
}
```

Código 11: Imprime arestas

```
/*
 * Funcão Grau
 * Calcula o grau (número de conexões) de um vértice u.
 * Para lista de adjacência: percorre a lista e conta os nós.
 * O grau representa quantos vizinhos (arestas) o vértice possui.
 */
int Grau(p_grafo g, int u) {
    int grau = 0;  // Variável para contar o grau
    p_no t;        // Ponteiro para percorrer a lista de adjacência
    // Percorre toda a lista de adjacência do vértice u
    for (t = g->adjacencia[u]; t != NULL; t = t->prox) {
        grau++;  // Incrementa o grau para cada vizinho encontrado
    }
    // Retorna o número total de vizinhos (grau do vértice)
    return grau;
}
```

Código 12: Grau

```
// Funcão que encontra o vértice mais popular (com maior grau)
int MaisPopular(p_grafo g) {
    int u, max, grauMax, grauAtual;
    max = 0;               // Inicializa o mais popular como o vértice 0
    grauMax = Grau(g, 0);  // Calcula o grau do vértice 0

    // Percorre todos os vértices a partir do 1
    for (u = 1; u < g->n; u++) {
        grauAtual = Grau(g, u);     // Calcula o grau do vértice u
        if (grauAtual > grauMax) {  // Se tiver grau maior, atualiza o máximo
            grauMax = grauAtual;
            max = u;
        }
    }
    return max;  // Retorna o vértice com maior grau
}
```

Código 13: Mais popular (maior grau)

```
// Funcão que imprime "recomendacões" de vértices conectados a amigos de 'u'
// Para lista: percorre a lista de adjacência de u, depois a lista de cada
    vizinho
```

```c
void ImprimeRecomendacoes(p_grafo g, int u) {
    p_no t, w;
    // Percorre todos os vizinhos de u (amigos diretos)
    for (t = g->adjacencia[u]; t != NULL; t = t->prox) {
        int v = t->v;  // v é um amigo de u
        // Percorre todos os vizinhos de v (amigos dos amigos)
        for (w = g->adjacencia[v]; w != NULL; w = w->prox) {
            // Se w não é u E w não é amigo de u, então recomenda
            if (w->v != u && !TemAresta(g, u, w->v)) {
                printf("%d\n", w->v);  // Recomenda w a u
            }
        }
    }
}
```

Código 14: Imprime recomendações

```c
/*
 * RACIOCÍNIO GERAL:
 * Para encontrar componentes conexas em um grafo:
 * 1. Marcar todos os vértices como "não visitados" (-1)
 * 2. Para cada vértice não visitado:
 *    - Fazer uma busca (DFS) marcando todos os vértices alcancáveis com o mesmo número de componente
 *    - Incrementar o contador de componentes
 * 3. Retornar o array com o componente de cada vértice
 */
int* EncontraComponentes(p_grafo g) {
    // Declaracão das variáveis:
    // s = vértice sendo analisado
    // c = contador de componentes (comeca em 0)
    // componentes = array que armazena qual componente cada vértice pertence
    int s, c = 0, *componentes = malloc(g->n * sizeof(int));

    // PASSO 1: Inicializar todos os vértices como não visitados (-1)
    // Percorre todos os n vértices do grafo
    for (s = 0; s < g->n; s++) {
        componentes[s] = -1;  // -1 indica que o vértice ainda não foi visitado
    }

    // PASSO 2: Para cada vértice não visitado, explorar sua componente conexa
    // Percorre novamente todos os vértices
    for (s = 0; s < g->n; s++) {
        // Se o vértice s ainda não foi visitado (componentes[s] == -1)
        if (componentes[s] == -1) {
            // Visita recursivamente todos os vértices alcancáveis a partir de s
            // marcando-os com o número do componente atual (c)
            VisitaRec(g, componentes, c, s);

            // Incrementa o contador de componentes para a próxima componente conexa
            C;
        }
    }

    // PASSO 3: Retornar o array com os componentes
    // Cada posicão i do array contém o número do componente ao qual o vértice i pertence
    return componentes;
```

```
}
```

Código 15: Encontra componentes

```
/*
 * RACIOCÍNIO DA BUSCA EM PROFUNDIDADE (DFS):
 * Esta funcão implementa uma DFS (Depth-First Search) recursiva
 * que marca todos os vértices alcancáveis a partir de v com o mesmo número de componente
 */
void VisitaRec(p_grafo g, int* componentes, int comp, int v) {
    // t = ponteiro para percorrer a lista de adjacência
    p_no t;

    // PASSO 1: Marcar o vértice atual v como pertencente ao componente comp
    componentes[v] = comp;

    // PASSO 2: Percorrer todos os vizinhos do vértice v
    // t comeca no primeiro nó da lista de adjacência de v
    // e avanca enquanto não chegar ao fim (NULL)
    for (t = g->adjacencia[v]; t != NULL; t = t->prox) {
        // Se o vizinho t->v ainda não foi visitado (componentes[t->v] == -1)
        if (componentes[t->v] == -1) {
            // RECURSÃO: Visita o vizinho não visitado
            // Isso garante que todos os vértices alcancáveis serão marcados
            VisitaRec(g, componentes, comp, t->v);
        }
    }
    // Quando a recursão terminar, todos os vértices da componente conexa
    // estarão marcados com o mesmo número (comp)
}
```

Código 16: Visita recursiva (busca em profundidade)

```
int* encontraCaminhos(p_grafo g, int s) {
    int i, *pai = malloc(g->n * sizeof(int));
    for (i = 0; i < g->n; i++) {
        pai[i] = -1;
    }
    buscaEmProfundidade(g, pai, s, s);
    return pai;
}

void buscaEmProfundidade(p_grafo g, int* pai, int p, int v) {
    p_no t;
    pai[v] = p;
    for (t = g->adjacencia[v]; t != NULL; t = t->prox) {
        if (pai[t->v] == -1) {
            buscaEmProfundidade(g, pai, v, t->v);
        }
    }
}
```

Código 17: Encontra caminhos

```
void ordenacao_topologica(p_grafo g) {
    int s, *visitado = malloc(g->n * sizeof(int));
    for (s = 0; s < g->n; s++)
        visitado[s] = 0;
    for (s = 0; s < g->n; s++)
        if (!visitado[s])
            visita_rec(g, visitado, s);
```

```
    free(visitado);
    printf("\n");
}
```

Código 18: Ordenação topológica

```
void visita_rec(p_grafo g, int* visitado, int v) {
    p_no t;
    visitado[v] = 1;
    for (t = g->adjacencia[v]; t != NULL; t = t->prox)
        if (!visitado[t->v])
            visita_rec(g, visitado, t->v);
    printf("%d ", v);
}
```

Código 19: Example C code

```
// Funções de fila de prioridade

/*
 * Funcao auxiliar para trocar dois itens na fila de prioridade
 */
void troca(p_fp h, int i, int j) {
    Item temp = h->v[i];
    h->v[i] = h->v[j];
    h->v[j] = temp;

    // Atualiza os indices
    h->indice[h->v[i].vertice] = i;
    h->indice[h->v[j].vertice] = j;
}

/*
 * Funcao auxiliar para subir um elemento no heap (heapify up)
 */
void sobe(p_fp h, int k) {
    int pai;
    while (k > 0) {
        pai = (k - 1) / 2;
        if (h->v[k].prioridade < h->v[pai].prioridade) {
            troca(h, k, pai);
            k = pai;
        } else {
            break;
        }
    }
}

/*
 * Funcao auxiliar para descer um elemento no heap (heapify down)
 */
void desce(p_fp h, int k) {
    int filho;
    while (2 * k + 1 < h->n) {
        filho = 2 * k + 1;

        // Escolhe o menor filho
        if (filho + 1 < h->n && h->v[filho + 1].prioridade < h->v[filho].
   prioridade) {
            filho++;
        }

```

```
45          if (h->v[filho].prioridade < h->v[k].prioridade) {
46              troca(h, k, filho);
47              k = filho;
48          } else {
49              break;
50          }
51      }
52  }
53
54  /*
55   * Funcao criar_fprio
56   * Cria e inicializa uma fila de prioridade (min-heap) para armazenar vertices
57   * tamanho: numero maximo de elementos (geralmente o numero de vertices do grafo
       )
58   */
59  p_fp criar_fprio(int tamanho) {
60      p_fp h = malloc(sizeof(FP));
61      h->v = malloc(tamanho * sizeof(Item));
62      h->indice = malloc(tamanho * sizeof(int));
63      h->n = 0;
64      h->tamanho = tamanho;
65
66      // Inicializa os indices como -1 (nao presente)
67      for (int i = 0; i < tamanho; i++) {
68          h->indice[i] = -1;
69      }
70
71      return h;
72  }
73
74  /*
75   * Funcao destroi_fprio
76   * Libera toda a memoria alocada pela fila de prioridade
77   */
78  void destroi_fprio(p_fp h) {
79      free(h->v);
80      free(h->indice);
81      free(h);
82  }
83
84  /*
85   * Funcao insere
86   * Insere um vertice com uma dada prioridade na fila
87   * vertice: identificador do vertice
88   * prioridade: valor da prioridade (menor = maior prioridade)
89   */
90  void insere(p_fp h, int vertice, int prioridade) {
91      h->v[h->n].vertice = vertice;
92      h->v[h->n].prioridade = prioridade;
93      h->indice[vertice] = h->n;
94      h->n++;
95      sobe(h, h->n - 1);
96  }
97
98  /*
99   * Funcao extrai_minimo
100  * Remove e retorna o vertice com menor prioridade
101  * Retorna o identificador do vertice
102  */
103 int extrai_minimo(p_fp h) {
```

```
    int minimo = h->v[0].vertice;
    h->indice[minimo] = -1;  // Marca como removido

    h->n--;
    if (h->n > 0) {
        h->v[0] = h->v[h->n];
        h->indice[h->v[0].vertice] = 0;
        desce(h, 0);
    }

    return minimo;
}

/*
 * Funcao vazia
 * Verifica se a fila de prioridade esta vazia
 * Retorna 1 se vazia, 0 caso contrario
 */
int vazia(p_fp h) {
    return h->n == 0;
}

/*
 * Funcao prioridade
 * Retorna a prioridade atual de um vertice
 * vertice: identificador do vertice
 * Retorna INT_MAX se o vertice nao estiver na fila
 */
int prioridade(p_fp h, int vertice) {
    int idx = h->indice[vertice];
    if (idx == -1) {
        return INT_MAX;  // Vertice nao esta na fila
    }
    return h->v[idx].prioridade;
}

/*
 * Funcao diminuiprioridade
 * Diminui a prioridade de um vertice ja presente na fila
 * vertice: identificador do vertice
 * nova_prioridade: novo valor de prioridade (deve ser menor que o atual)
 * IMPORTANTE: Esta funcao assume que nova_prioridade < prioridade atual
 */
void diminuiprioridade(p_fp h, int vertice, int nova_prioridade) {
    int idx = h->indice[vertice];

    if (idx == -1) {
        return;  // Vertice nao esta na fila
    }

    // Atualiza a prioridade
    h->v[idx].prioridade = nova_prioridade;

    // Sobe o elemento no heap (pois a prioridade diminuiu)
    sobe(h, idx);
}
```

Código 20: Funções de fila de prioridade

```
/*
 * Funcao dijkstra
```

```
 * Implementa o algoritmo de Dijkstra para encontrar caminhos minimos
 * g: grafo com arestas ponderadas
 * s: vertice origem
 * Retorna: array de pais representando a arvore de caminhos minimos
 *          pai[v] = -1 se v nao eh alcancavel de s
 *          pai[v] = v se v eh a origem
 *          pai[v] = u significa que u eh o predecessor de v no caminho minimo
 */
int* dijkstra(p_grafo g, int s) {
    int v, *pai = malloc(g->n * sizeof(int));
    int* dist = malloc(g->n * sizeof(int));  // Array de distâncias
    p_no t;
    p_fp h = criar_fprio(g->n);

    // Inicializa todos os vertices com distancia infinita
    for (v = 0; v < g->n; v++) {
        pai[v] = -1;
        dist[v] = INT_MAX;
        insere(h, v, INT_MAX);
    }

    // Define o vertice origem
    pai[s] = s;
    dist[s] = 0;
    diminuiprioridade(h, s, 0);

    // Processa vertices em ordem de distancia crescente
    while (!vazia(h)) {
        v = extrai_minimo(h);

        // Se a distancia eh infinita, vertices restantes sao inalcancaveis
        if (dist[v] != INT_MAX) {
            // Relaxa todas as arestas saindo de v
            for (t = g->adjacencia[v]; t != NULL; t = t->prox) {
                // Se encontrou um caminho mais curto para t->v
                int nova_dist = dist[v] + t->peso;
                if (nova_dist < dist[t->v]) {
                    dist[t->v] = nova_dist;
                    diminuiprioridade(h, t->v, nova_dist);
                    pai[t->v] = v;
                }
            }
        }
    }

    destroi_fprio(h);
    free(dist);
    return pai;
}
```

Código 21: Função de Dijkstra

2 Grafos utilizando matriz de adjacências

```
/*
 * Funcao criar_pilha
 * Cria e inicializa uma pilha dinamica para armazenar inteiros
 * Capacidade inicial de 100 elementos (pode ser expandida)
 */
```

```c
p_pilha criar_pilha() {
    p_pilha p = malloc(sizeof(Pilha));
    p->capacidade = 100;
    p->dados = malloc(p->capacidade * sizeof(int));
    p->topo = -1;  // Pilha vazia tem topo -1
    return p;
}

/*
 * Funcao destroi_pilha
 * Libera toda a memoria alocada pela pilha
 */
void destroi_pilha(p_pilha p) {
    free(p->dados);
    free(p);
}

/*
 * Funcao empilhar
 * Insere um valor no topo da pilha
 * Expande a capacidade se necessario
 */
void empilhar(p_pilha p, int valor) {
    // Se a pilha estiver cheia, dobra a capacidade
    if (p->topo == p->capacidade - 1) {
        p->capacidade *= 2;
        p->dados = realloc(p->dados, p->capacidade * sizeof(int));
    }
    // Incrementa o topo e insere o valor
    p->topo++;
    p->dados[p->topo] = valor;
}

/*
 * Funcao desempilhar
 * Remove e retorna o valor do topo da pilha
 * Retorna -1 se a pilha estiver vazia (cuidado: pode ser um valor valido)
 */
int desempilhar(p_pilha p) {
    if (p->topo == -1) {
        return -1;  // Pilha vazia
    }
    int valor = p->dados[p->topo];
    p->topo--;
    return valor;
}

/*
 * Funcao pilha_vazia
 * Verifica se a pilha esta vazia
 * Retorna 1 se vazia, 0 caso contrario
 */
int pilha_vazia(p_pilha p) {
    return p->topo == -1;
}
```

Código 22: Funções de pilha (usado em busca em profundidade (*DFS*))

```c
/*
 * Funcao criar_fila
 * Cria e inicializa uma fila circular dinamica para armazenar inteiros
```

```
 * Capacidade inicial de 100 elementos (pode ser expandida)
 */
p_fila criar_fila() {
    p_fila f = malloc(sizeof(Fila));
    f->capacidade = 100;
    f->dados = malloc(f->capacidade * sizeof(int));
    f->inicio = 0;
    f->fim = 0;
    f->tamanho = 0;
    return f;
}

/*
 * Funcao destroi_fila
 * Libera toda a memoria alocada pela fila
 */
void destroi_fila(p_fila f) {
    free(f->dados);
    free(f);
}

/*
 * Funcao enfileira
 * Insere um valor no final da fila
 * Expande a capacidade se necessario
 */
void enfileira(p_fila f, int valor) {
    // Se a fila estiver cheia, dobra a capacidade
    if (f->tamanho == f->capacidade) {
        int nova_capacidade = f->capacidade * 2;
        int* novos_dados = malloc(nova_capacidade * sizeof(int));

        // Copia os elementos para o novo array
        for (int i = 0; i < f->tamanho; i++) {
            novos_dados[i] = f->dados[(f->inicio + i) % f->capacidade];
        }

        free(f->dados);
        f->dados = novos_dados;
        f->inicio = 0;
        f->fim = f->tamanho;
        f->capacidade = nova_capacidade;
    }

    // Insere o valor no final da fila
    f->dados[f->fim] = valor;
    f->fim = (f->fim + 1) % f->capacidade;
    f->tamanho++;
}

/*
 * Funcao desenfileira
 * Remove e retorna o valor do inicio da fila
 * Retorna -1 se a fila estiver vazia (cuidado: pode ser um valor valido)
 */
int desenfileira(p_fila f) {
    if (f->tamanho == 0) {
        return -1;  // Fila vazia
    }

```

```
    int valor = f->dados[f->inicio];
    f->inicio = (f->inicio + 1) % f->capacidade;
    f->tamanho--;
    return valor;
}

/*
 * Funcao fila_vazia
 * Verifica se a fila esta vazia
 * Retorna 1 se vazia, 0 caso contrario
 */
int fila_vazia(p_fila f) {
    return f->tamanho == 0;
}
```

Código 23: Funções de fila (usado em busca em largura (*BFS*))

```
// Função que cria um grafo com 'n' vértices
p_grafo CriarGrafo(int n) {
    int i, j;
    p_grafo g = malloc(sizeof(Grafo));  // Aloca memória para a estrutura principal do grafo
    g->n = n;                           // Define o número de vértices
    g->adj = malloc(n * sizeof(int*));  // Aloca memória para o vetor de ponteiros das listas de adjacência (matriz)

    for (i = 0; i < n; i++) {
        g->adj[i] = malloc(n * sizeof(int));  // Aloca memória para cada linha da matriz de adjacência
    }

    // Inicializa a matriz de adjacência com zeros (sem arestas)
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            g->adj[i][j] = 0;  // 0 indica ausência de aresta
        }
    }

    return g;  // Retorna o ponteiro para o grafo criado
}
```

Código 24: Cria grafo

```
// Função que libera toda a memória usada pelo grafo
void DestroiGrafo(p_grafo g) {
    int i;
    for (i = 0; i < g->n; i++) {
        free(g->adj[i]);  // Libera cada linha da matriz
    }
    free(g->adj);  // Libera o vetor de ponteiros
    free(g);       // Libera a estrutura principal do grafo
}
```

Código 25: Destroí grafo

```
// Função que insere uma aresta entre os vértices u e v (grafo não direcionado)
void InsereAresta(p_grafo g, int u, int v) {
    g->adj[u][v] = 1;  // Marca a conexão de u para v
    g->adj[v][u] = 1;  // Marca a conexão de v para u (pois o grafo é não direcionado)
}
```

Código 26: Insere aresta

```c
// Função que remove uma aresta entre os vértices u e v
void RemoveAresta(p_grafo g, int u, int v) {
    g->adj[u][v] = 0;  // Remove a conexão de u para v
    g->adj[v][u] = 0;  // Remove a conexão de v para u
}
```

Código 27: Remove aresta

```c
// Função que verifica se há uma aresta entre u e v
int TemAresta(p_grafo g, int u, int v) {
    return g->adj[u][v];  // Retorna 1 se há aresta, 0 caso contrário
}
```

Código 28: Tem aresta

```c
// Função que lê um grafo da entrada padrão
p_grafo LeGrafo() {
    int n, m, i, u, v;
    p_grafo g;
    scanf("%d %d", &n, &m);  // Lê número de vértices (n) e número de arestas (m
    )
    g = CriarGrafo(n);       // Cria o grafo com n vértices

    for (i = 0; i < m; i++) {
        scanf("%d %d", &u, &v);  // Lê os pares de vértices conectados por
    arestas
        InsereAresta(g, u, v);   // Insere cada aresta no grafo
    }
    return g;  // Retorna o grafo criado
}
```

Código 29: Lê grafo

```c
// Função que imprime todas as arestas do grafo
void ImprimeArestas(p_grafo g) {
    int u, v;
    for (u = 0; u < g->n; u++) {
        for (v = 0; v < g->n; v++) {
            if (g->adj[u][v]) {              // Se existe uma aresta entre u e v
                printf("{%d %d}\n", u, v);  // Imprime a aresta
            }
        }
    }
}
```

Código 30: Imprime arestas

```c
// Função que calcula o grau (número de conexões) de um vértice u
int Grau(p_grafo g, int u) {
    int v, grau = 0;
    for (v = 0; v < g->n; v++) {
        if (g->adj[u][v]) {  // Conta quantas arestas saem de u
            grau++;
        }
    }
    return grau;  // Retorna o grau total
}
```

Código 31: Grau

```c
// Função que encontra o vértice mais popular (com maior grau)
int MaisPopular(p_grafo g) {
```

```c
    int u, max, grauMax, grauAtual;
    max = 0;               // Inicializa o mais popular como o vértice 0
    grauMax = Grau(g, 0);  // Calcula o grau do vértice 0

    for (u = 1; u < g->n; u++) {
        grauAtual = Grau(g, u);     // Calcula o grau do vértice u
        if (grauAtual > grauMax) {  // Se tiver grau maior, atualiza o máximo
            grauMax = grauAtual;
            max = u;
        }
    }
    return max;  // Retorna o vértice com maior grau
}
```

Código 32: Mais popular (maior grau)

```c
// Função que imprime "recomendações" de vértices conectados a amigos de 'u'
void ImprimeRecomendacoes(p_grafo g, int u) {
    int v, w;
    for (v = 0; v < g->n; v++) {
        if (g->adj[u][v]) {  // Se v é amigo de u
            for (w = 0; w < g->n; w++) {
                if (g->adj[v][w] && w != u && !g->adj[u][w])
                    // Se w é amigo de v, mas não de u e não é o próprio u
                    printf("%d\n", w);  // Recomenda w a u
            }
        }
    }
}
```

Código 33: Imprime recomendações

```c
// Função auxiliar recursiva para busca de caminho (DFS)
int BuscaRec(p_grafo g, int* visitado, int v, int t) {
    int w;
    if (v == t) {  // Se encontrou o vértice destino
        return 1;
    }
    visitado[v] = 1;  // Marca o vértice atual como visitado
    for (w = 0; w < g->n; w++) {
        if (g->adj[v][w] && !visitado[w]) {   // Se existe aresta e w ainda não
    foi visitado
            if (BuscaRec(g, visitado, w, t))  // Continua a busca a partir de w
                return 1;                     // Se achou o destino, retorna 1
        }
    }
    return 0;  // Se não encontrou, retorna 0
}

// Função que verifica se existe caminho entre os vértices s e t
int ExisteCaminho(p_grafo g, int s, int t) {
    int encontrou, i, *visitado = malloc(g->n * sizeof(int));  // Vetor para
    marcar vértices visitados
    for (i = 0; i < g->n; i++) {
        visitado[i] = 0;  // Inicializa todos como não visitados
    }
    encontrou = BuscaRec(g, visitado, s, t);  // Faz busca recursiva (DFS)
    free(visitado);                           // Libera a memória
    return encontrou;                         // Retorna 1 se encontrou caminho,
    0 se não
```

```
}
```

Código 34: Existe caminho

```
/*
 * RACIOCINIO GERAL:
 * Para encontrar componentes conexas em um grafo:
 * 1. Marcar todos os vertices como "nao visitados" (-1)
 * 2. Para cada vertice nao visitado:
 *    - Fazer uma busca (DFS) marcando todos os vertices alcancaveis com o mesmo numero de componente
 *    - Incrementar o contador de componentes
 * 3. Retornar o array com o componente de cada vertice
 */
int* EncontraComponentes(p_grafo g) {
    // Declaracao das variaveis:
    // s = vertice sendo analisado
    // c = contador de componentes (comeca em 0)
    // componentes = array que armazena qual componente cada vertice pertence
    int s, c = 0, *componentes = malloc(g->n * sizeof(int));

    // PASSO 1: Inicializar todos os vertices como nao visitados (-1)
    // Percorre todos os n vertices do grafo
    for (s = 0; s < g->n; s++) {
        componentes[s] = -1;  // -1 indica que o vertice ainda nao foi visitado
    }

    // PASSO 2: Para cada vertice nao visitado, explorar sua componente conexa
    // Percorre novamente todos os vertices
    for (s = 0; s < g->n; s++) {
        // Se o vertice s ainda nao foi visitado (componentes[s] == -1)
        if (componentes[s] == -1) {
            // Visita recursivamente todos os vertices alcancaveis a partir de s
            // marcando-os com o numero do componente atual (c)
            VisitaRec(g, componentes, c, s);

            // Incrementa o contador de componentes para a proxima componente conexa
            c++;
        }
    }

    // PASSO 3: Retornar o array com os componentes
    // Cada posicao i do array contem o numero do componente ao qual o vertice i pertence
    return componentes;
}
```

Código 35: Encontra componentes

```
/*
 * RACIOCINIO DA BUSCA EM PROFUNDIDADE (DFS):
 * Esta funcao implementa uma DFS (Depth-First Search) recursiva
 * que marca todos os vertices alcancaveis a partir de v com o mesmo numero de componente
 * Versao para matriz de adjacencia
 */
void VisitaRec(p_grafo g, int* componentes, int comp, int v) {
    int w;

    // PASSO 1: Marcar o vertice atual v como pertencente ao componente comp
    componentes[v] = comp;
```

```
12
13      // PASSO 2: Percorrer todos os possiveis vizinhos do vertice v
14      // Para matriz de adjacencia, verifica todos os vertices
15      for (w = 0; w < g->n; w++) {
16          // Se existe aresta entre v e w E w ainda nao foi visitado
17          if (g->adj[v][w] && componentes[w] == -1) {
18              // RECURSAO: Visita o vizinho nao visitado
19              // Isso garante que todos os vertices alcancaveis serao marcados
20              VisitaRec(g, componentes, comp, w);
21          }
22      }
23      // Quando a recursao terminar, todos os vertices da componente conexa
24      // estarao marcados com o mesmo numero (comp)
25  }
```

Código 36: Busca em profundidade (*DFS*)

```
1   /*
2    * Funcao encontraCaminhos
3    * Encontra caminhos de um vertice origem s para todos os vertices alcancaveis
4    * Retorna um array onde pai[i] indica o pai do vertice i na arvore de busca
5    * -1 indica vertice nao alcancavel, i indica raiz (pai de si mesmo)
6    */
7   int* encontraCaminhos(p_grafo g, int s) {
8       int i, *pai = malloc(g->n * sizeof(int));
9       // Inicializa todos os vertices como nao visitados (-1)
10      for (i = 0; i < g->n; i++) {
11          pai[i] = -1;
12      }
13      // Inicia busca em profundidade a partir de s
14      // s e pai de si mesmo (raiz da busca)
15      buscaEmProfundidade(g, pai, s, s);
16      return pai;
17  }
```

Código 37: Encontra caminhos

```
1   /*
2    * Funcao buscaEmProfundidade (recursiva)
3    * Implementa DFS recursiva para construir arvore de caminhos
4    * g = grafo, pai = array de pais, p = pai do vertice atual, v = vertice atual
5    * Versao para matriz de adjacencia
6    */
7   void buscaEmProfundidade(p_grafo g, int* pai, int p, int v) {
8       int w;
9       // Marca o pai do vertice atual
10      pai[v] = p;
11
12      // Percorre todos os possiveis vizinhos
13      for (w = 0; w < g->n; w++) {
14          // Se existe aresta entre v e w E w ainda nao foi visitado
15          if (g->adj[v][w] && pai[w] == -1) {
16              // Continua a busca recursivamente
17              // v se torna o pai de w
18              buscaEmProfundidade(g, pai, v, w);
19          }
20      }
21  }
```

Código 38: Busca em profundidade (recursiva)

```
1   /*
```

```c
 * Funcao busca_em_profundidade (iterativa com pilha)
 * Implementa DFS iterativa usando pilha explicita
 * Alternativa a versao recursiva, evita estouro de pilha em grafos grandes
 * Retorna array de pais formando arvore de busca
 */
int* busca_em_profundidade(p_grafo g, int s) {
    int w, v;
    int* pai = malloc(g->n * sizeof(int));
    int* visitado = malloc(g->n * sizeof(int));
    p_pilha p = criar_pilha();
    for (v = 0; v < g->n; v++) {
        pai[v] = -1;
        visitado[v] = 0;
    }
    empilhar(p, s);
    pai[s] = s;
    while (!pilha_vazia(p)) {
        v = desempilhar(p);
        visitado[v] = 1;
        for (w = 0; w < g->n; w++)
            if (g->adj[v][w] && !visitado[w]) {
                pai[w] = v;
                empilhar(p, w);
            }
    }
    destroi_pilha(p);
    free(visitado);
    return pai;
}
```

Código 39: Busca em profundidade (usando pilha)

```c
/*
 * Funcao busca_em_largura (BFS - Breadth-First Search)
 * Implementa BFS usando fila para explorar o grafo nivel por nivel
 * Retorna array de pais formando arvore de busca
 * BFS encontra o caminho mais curto (em numero de arestas) entre vertices
 */
int* busca_em_largura(p_grafo g, int s) {
    int w, v;
    int* pai = malloc(g->n * sizeof(int));
    int* visitado = malloc(g->n * sizeof(int));
    p_fila f = criar_fila();

    // Inicializa todos os vertices como nao visitados
    for (v = 0; v < g->n; v++) {
        pai[v] = -1;
        visitado[v] = 0;
    }

    // Enfileira o vertice inicial
    enfileira(f, s);
    pai[s] = s;
    visitado[s] = 1;

    // Processa vertices nivel por nivel
    while (!fila_vazia(f)) {
        v = desenfileira(f);

        // Explora todos os vizinhos de v
        for (w = 0; w < g->n; w++) {
```

```
30              if (g->adj[v][w] && !visitado[w]) {
31                  visitado[w] = 1;  // Marca como visitado para evitar repeticao
    na fila
32                  pai[w] = v;       // Define v como pai de w
33                  enfileira(f, w);  // Enfileira w para processar seus vizinhos
    depois
34              }
35          }
36      }
37
38      destroi_fila(f);
39      free(visitado);
40      return pai;
41  }
```

Código 40: Função de busca em largura (*BFS*)