Universidade Federal do Piauí – UFPI
Campus Senador Helvídio Nunes de Barros – CSHNB
Curso de Sistemas de Informação
Disciplina: Estruturas de Dados II
Professora: Juliana Oliveira de Carvalho
Acadêmico: Vinicius da Silva Nunes, Annyel Cordeiro

RELATÓRIO DE ESTRUTURA DE DADOS 2

Relatório Detalhado do Código da Questão 1

1. INTRODUÇÃO

O problema da Torre de Hanói é um desafio clássico e bem conhecido na ciência da computação e na teoria dos algoritmos. Este problema envolve mover uma torre de discos de um pino de origem para um pino de destino, usando um pino auxiliar, seguindo algumas regras específicas.

Descrição do Problema:Os discos são empilhados de maior para menor em um dos pinos, formando uma torre.A única operação permitida é mover um disco por vez.Um disco só pode ser colocado em cima de um disco maior ou em um pino vazio.

Representação e Modelagem: Cada configuração da Torre de Hanói pode ser representada por um vetor, onde cada posição do vetor corresponde a um pino, e o valor na posição indica o tamanho do disco presente no pino. O grafo é modelado de forma que cada vértice representa uma configuração da Torre de Hanói, e as arestas representam movimentos legais entre configurações.Grafo para 4 Discos:Para o caso de 4 discos, há várias configurações possíveis que podem ser alcançadas por movimentos legais. Cada vértice no grafo representa uma configuração possível dos discos, e as arestas ligam as configurações que podem ser alcançadas por um movimento legal.

Matriz de Adjacência:Utilizando uma matriz de adjacência, o grafo pode ser representado, onde o valor 1 em uma posição indica a existência de uma aresta entre os vértices correspondentes.

Algoritmo de Ford-Moore-Bellman:O Algoritmo de Ford-Moore-Bellman é utilizado para encontrar o menor caminho entre uma configuração inicial e a configuração final no grafo.Cada aresta tem peso 1, pois o objetivo é minimizar o número de movimentos.O tempo gasto para encontrar a solução é contabilizado.Em resumo, o problema da Torre de Hanói é um excelente exemplo para explorar conceitos de algoritmos, modelagem de grafos e análise de desempenho.

O código em questão implementa um algoritmo baseado no Algoritmo de Bellman-Ford para encontrar o menor caminho em um grafo que representa o desafio da Torre de Hanói. O grafo é representado por uma matriz de adjacência, onde os vértices representam configurações diferentes do jogo e as arestas indicam as possíveis transições entre essas configurações.

2. FUNÇÃO PARA INICIALIZAR O GRAFO

A função inicializarGrafo preenche a matriz de adjacência com zeros, representando a ausência de conexões entre os vértices.
Cada elemento grafo[i][j] é inicializado com 0, indicando a ausência de uma aresta entre os vértices i e j.

3. FUNÇÃO PARA ADICIONAR ARESTA AO GRAFO

A função adicionarAresta recebe a matriz de adjacência `grafo`, a origem e o destino da aresta, e atribui o valor 1 à posição correspondente na matriz, indicando a presença de uma aresta entre os vértices de origem e destino.

4. FUNÇÃO PARA EXIBIR A MATRIZ DE ADJACÊNCIA

A função exibirMatrizAdjacencia imprime a matriz de adjacência na tela, facilitando a visualização da representação do grafo.

5. FUNÇÃO PARA ENCONTRAR O MENOR CAMINHO (BELLMAN-FORD)

A função bellmanFord implementa o Algoritmo de Bellman-Ford para encontrar o menor caminho entre a origem e o destino no grafo.
Utiliza um array distancias para armazenar as distâncias mínimas até cada vértice a partir da origem.
A medição do tempo de execução é realizada usando a biblioteca `time.h`.

6. FUNÇÃO PRINCIPAL (MAIN)

A função principal main declara a matriz de adjacência `grafo` e define a origem e o destino para o cálculo do menor caminho.
Utiliza as funções previamente definidas para inicializar o grafo, adicionar as arestas e exibir a matriz de adjacência.
Chama a função bellmanFord para encontrar o menor caminho e imprime otempo de busca e o resultado final.

CONCLUSÃO

O código apresenta uma implementação eficiente do Algoritmo de Bellman-Ford para encontrar o menor caminho em um grafo que representa o desafio da Torre de Hanói. As funções são modularizadas e bem comentadas, facilitando a compreensão e manutenção do código. O uso de constantes e bibliotecas apropriadas contribui para a clareza e organização do programa.

Relatório Detalhado do Código da Questão 2

1. INTRODUÇÃO

O problema propõe a construção de um programa eficiente para encontrar o caminho mais confiável entre dois vértices em um grafo orientado, onde cada aresta tem um valor associado que representa a confiabilidade do canal de comunicação. A confiabilidade é interpretada como a probabilidade de que o canal não venha a falhar, e assume valores no intervalo de 0 a 1.

Descrição do Problema:O problema pode ser modelado como um problema de caminho mais longo, onde se busca maximizar a confiabilidade acumulada ao longo do caminho.

Grafo Orientado com Valores de Confiança:O grafo é representado como um conjunto de vértices \(V\) e arestas \(E\). Cada aresta \((u, v) \in E\) possui um valor \(r(u, v)\) representando a confiabilidade do canal de \(u\) até \(v\). Modelagem do Problema:O problema pode ser modelado como um problema de caminho mais longo, onde se busca maximizar a confiabilidade acumulada ao longo do caminho.
Algoritmo de Dijkstra Modificado:O algoritmo de Dijkstra é adaptado para maximizar a confiabilidade ao invés de minimizar o custo.O valor acumulado é atualizado multiplicando a confiabilidade da aresta pelo valor acumulado atual.Em resumo, a resolução do problema propõe uma abordagem eficiente para encontrar o caminho mais confiável em um grafo orientado, considerando as probabilidades de falha associadas às arestas. A adaptação do algoritmo de Dijkstra permite explorar eficazmente a confiabilidade do sistema.

O código implementa o algoritmo de Dijkstra para encontrar o caminho mais confiável entre dois vértices em um grafo direcionado ponderado, onde as arestas têm uma confiabilidade associada. O código utiliza estruturas de dados para representar o grafo e as operações relacionadas.

2. BIBLIOTECAS E CONSTANTES

As bibliotecas `stdio.h`, `stdlib.h`, e `limits.h` são incluídas para fornecer funcionalidades de entrada/saída, alocação de memória e valores extremos de variáveis.
A constante MAX_VERTICES define o número máximo de vértices no grafo.

3. ESTRUTURAS DE DADOS

A estrutura Aresta representa uma aresta no grafo, contendo o destino da aresta e sua confiabilidade.
A estrutura Vertice representa um vértice no grafo, contendo o índice do vértice e sua confiabilidade.
A estrutura Grafo representa o grafo em si, utilizando uma matriz de adjacência para armazenar as arestas e um número total de vértices.

4. FUNÇÃO PARA INICIALIZAR O GRAFO

A função inicializarGrafo() recebe um ponteiro para uma estrutura Grafo e o número de vértices, inicializando a matriz de adjacência com destinos e confiabilidades padrão.

5. FUNÇÃO PARA ADICIONAR ARESTA AO GRAFO

A função adicionarAresta() recebe um ponteiro para uma estrutura Grafo, a origem, o destino e a confiabilidade da aresta. Ela adiciona a aresta na matriz de adjacência.

6. ALGORITMO DE DIJKSTRA

A função dijkstra implementa o algoritmo de Dijkstra para encontrar o caminho mais confiável entre dois vértices.
Utiliza as estruturas Vertice e Aresta, bem como arrays auxiliares para controlar os vértices visitados, distâncias e confiabilidades.

7. FUNÇÃO PRINCIPAL (MAIN)

A função principal main declara uma estrutura Grafo e variáveis para armazenar o número de vértices, origem e destino.
Utiliza as funções previamente definidas para inicializar o grafo e adicionar arestas.
Solicita ao usuário a entrada de dados para origem e destino.
Chama a função `dijkstra` para encontrar o caminho mais confiável e exibe o resultado.

CONCLUSÃO

O código fornece uma implementação eficiente do algoritmo de Dijkstra para encontrar o caminho mais confiável entre dois vértices em um grafo ponderado. A escolha de estruturas de dados apropriadas e a modularização das funções contribuem para a legibilidade e manutenção do código. O programa é interativo, permitindo a entrada do usuário para o número de vértices e vértices de origem e destino, proporcionando flexibilidade e adaptabilidade.

Relatório Detalhado do Código da Questão 3

1. struct Funcionario: Define a estrutura de um funcionário com os campos matricula, nome, função e salário.

2. typedef struct Hash: Define a estrutura de uma tabela hash com os campos `qtd` (quantidade de elementos), `tamanho` (tamanho do vetor destino), e `vetor_destino` (vetor de ponteiros para funcionários).

Funções de Hashing e Tratamento de Colisões

3. Função de Hashing (a) funcao_hash_a(char *matricula, int tamanho)
Realiza uma rotação de 2 dígitos para a esquerda na matrícula.
Extrai o 2º, 4º e 6º dígitos e obtém o resto da divisão pelo tamanho do vetor destino.
Trata colisões somando ao resto da divisão o primeiro dígito da matrícula.
- Retorna o índice final.

4. Tratamento de Colisões para a Função de Hashing (a) trata_colisao_a(int hash, char *digito)
Recebe o índice original e o primeiro dígito da matrícula.
Soma o primeiro dígito ao índice original.
Retorna o novo índice.

5. Função de Hashing (b) `funcao_hash_b(char *matricula, int tamanho)
Realiza um fold shift com 3 dígitos na matrícula.
Obtém o resto da divisão pelo tamanho do vetor destino.
Colisões são tratadas somando 7 ao valor obtido.
Retorna ao índice final.

6. Tratamento de Colisões para a Função de Hashing (b) trata_colisao_b(int hash)
Recebe o índice original.
Soma 7 ao índice original para tratar colisões.
Retorna o novo índice.

Funções de Manipulação da Tabela Hash

7. Hash *cria_tabela_hash(int tamanho)
Aloca dinamicamente uma tabela hash com o tamanho especificado.
Inicializa os campos da tabela.
Retorna um ponteiro para a tabela hash.

8.void insere_funcionario(Hash *hs, struct Funcionario func, int (*funcao_hash)(char *, int), int (*trata_colisao)(int, char *))
Insere um funcionário na tabela hash usando a função de hash e tratamento de colisões especificados.
Calcula a posição inicial usando a função de hash.
Trata colisões chamando a função de tratamento.

Aloca espaço para o funcionário na posição encontrada.
Atualiza a quantidade de elementos na tabela.

Função Principal

9. int main()
Cria quatro tabelas hash com diferentes tamanhos e funções de hash.
Inicializa um array de funcionários (exemplo).
Realiza a inserção dos funcionários nas tabelas hash usando diferentes configurações.
Imprime informações sobre a inserção (posição e número de colisões).

O código utiliza macros para definir tamanhos e número de funcionários. As funções de hash e tratamento de colisões são flexíveis, permitindo a troca dinâmica de estratégias. Durante a inserção, o código imprime informações sobre a posição e o número de colisões para cada funcionário.

Conclusão

O código implementa um sistema de tabela hash para armazenar informações de funcionários, utilizando duas funções de hash diferentes e duas estratégias de tratamento de colisões. Essa abordagem oferece flexibilidade na escolha de técnicas de hash e tratamento de colisões, permitindo uma comparação de desempenho entre diferentes configurações.