Grafos

Hugo Lopes
Nº 30516

Março de 2024

—

Estruturas de Dados Avançados

Professor
Luís Gonzaga Martins Ferreira

Índice

Resumo pag.3

Introdução pag.4

Trabalho desenvolvido pag.5

Implementação pag.7

O Desenvolvimento pag.12

Conclusão pag.13

Resumo

Este relatório descreve a implementação de um grafo utilizando uma estrutura de lista ligada, realizada como parte do segundo trabalho prático da disciplina de Estruturas de Dados Avançadas do curso de Engenharia de Sistemas de Informática. A escolha por listas ligadas deve-se à sua flexibilidade e eficiência em termos de memória, especialmente adequada para o grafo em questão.

O trabalho desenvolvido inclui funcionalidades essenciais como a criação e modificação de grafos e vértices, inserção e remoção de arestas, e métodos avançados de travessia e busca em grafos. Funções específicas permitem salvar e carregar grafos em arquivos binários, assegurando a persistência dos dados. Além disso, foram implementadas operações de contagem de caminhos entre vértices e verificação da existência de vértices.

As decisões focaram-se na eficiência das operações dinâmicas, proporcionando uma manipulação flexível e eficaz dos grafos. A análise dos resultados destaca as funcionalidades implementadas e sugere melhorias futuras.

Em conclusão, o trabalho demonstrou com sucesso a importância de escolher estruturas de dados adequadas para a implementação eficiente de grafos, proporcionando uma base sólida para futuras melhorias e expansões.

Introdução

Os grafos são estruturas de dados fundamentais em Ciência da Computação, usados para modelar e resolver uma ampla gama de problemas em diversas áreas, como redes de computadores, logística, redes sociais, e muito mais. Um grafo é composto por um conjunto de vértices (ou nós) e um conjunto de arestas (ou conexões) que ligam pares de vértices. Dependendo da aplicação, as arestas podem ser direcionadas ou não, e podem possuir pesos, representando diferentes medidas como distância, custo ou tempo.

Neste relatório, descrevemos a implementação de um grafo utilizando uma estrutura de lista ligada. Esta abordagem foi escolhida por sua flexibilidade e eficiência em termos de uso de memória. As listas ligadas permitem a inserção e remoção dinâmicas de vértices e arestas de forma mais eficiente comparada a outras estruturas, como matrizes de adjacência.

O objetivo principal deste trabalho é desenvolver uma série de funções que permitam criar, modificar e analisar grafos de maneira eficiente. Isso inclui operações básicas, como a inserção e remoção de vértices e arestas, e operações mais avançadas, como travessias de grafos e contagem de caminhos. Além disso, é implementada a funcionalidade de preservação dos grafos em ficheiros binários, garantindo que os dados possam ser armazenados e recuperados conforme necessário.

O trabalho realizado é parte do segundo trabalho prático da unidade curricular de Estruturas de Dados Avançadas do curso de Engenharia de Sistemas de Informática. A implementação das funções segue os requisitos especificados no enunciado do trabalho, e são feitas considerações sobre a eficiência e a robustez das operações realizadas.

Neste relatório, apresenta-se a análise e modelação da estrutura escolhida e detalha-se a implementação das funcionalidades. Concluí-se com uma avaliação do trabalho realizado e sugestões para futuras melhorias e expansões.

Trabalho desenvolvido

O trabalho desenvolvido focou na implementação de um grafo utilizando uma estrutura de lista ligada, atendendo aos requisitos do segundo trabalho prático da unidade curricular de Estruturas de Dados Avançadas do curso de Engenharia de Sistemas de Informática. A seguir, detalha-se as principais etapas e funcionalidades implementadas.

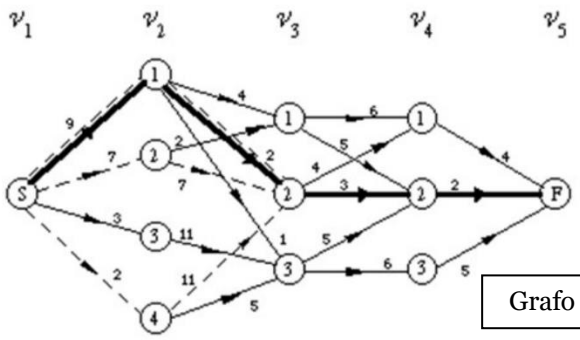
Análise e Modelação

Representação Escolhida para o Grafo

Optou-se pela representação do grafo utilizando uma lista ligada. Cada vértice do grafo é representado por um nó de lista que contém uma lista de adjacências. Cada adjacência representa uma aresta direcionada e possui um peso associado, refletindo a complexidade e os requisitos do problema a ser resolvido.

Tipo de Grafo

O grafo implementado é direcionado e ponderado, onde as arestas têm uma direção específica e um peso que pode representar, por exemplo, custo ou distância.



Grafo direcionado e ponderado

Dados das Adjacências

Cada adjacência armazena o identificador do vértice de destino e o peso da aresta, permitindo uma representação precisa e eficiente das conexões entre os vértices.

Preservação do Grafo

Para garantir a persistência dos dados, implementou-se a funcionalidade de salvar e carregar grafos a partir de arquivos binários:

Tipo de Preservação: Binária, para garantir uma representação compacta e eficiente dos dados.

Registros Usados: Estruturas de dados que representam vértices e suas adjacências.

Representação dos Registros no Ficheiro: Os vértices e suas adjacências são armazenados sequencialmente.

Regras para a Preservação: Garantir a consistência e a integridade dos dados ao salvar e carregar o grafo.

Implementação

Funções Relacionadas com a Criação da Lista Ligada:

Node2 novoNo(int peso2);*

Cria um novo nó Node2 com o peso especificado e retorna um apontador para ele.

Node2 InsereFim(Node2* head, Node2* peso2);*

Insere um novo nó Node2 no fim da lista ligada, com o peso especificado. Retorna o inicio da lista atualizada.

Node2 readFile(const char* ficheiro, int* totV, int* totA, bool* res);*

Lê um ficheiro para criar e inicializar um grafo, retornando um apontador para o inicio da lista de nós Node2. Também atualiza totV e totA com o número total de vértices e arestas, respetivamente, e res com o status da operação.

Funções Relacionadas com os Vértices:

Node CreateVertice(int id, bool* res);*

Cria um novo vértice Node com o identificador especificado e retorna um apontador para ele, atualizando res com o status da operação.

Node InsertVertice(Node* vertices, Node* novo, bool* res);*

Insere um novo vértice Node na lista de vértices. Retorna o inicio da lista atualizada e atualiza res com o status da operação.

Node DeleteVertice(Node* vertices, int codVertice, bool* res);*

Remove o vértice com o identificador especificado da lista de vértices. Retorna o inicio da lista atualizada e atualiza res com o status da operação.

Node WhereIsVertice(Node* inicio, int id);*

Encontra e retorna um apontador para o vértice com o identificador especificado na lista de vértices inicio.

Node DeleteAllAdjVert(Node* vertices, int codAdj, bool* res);*

Remove todas as adjacências do vértice especificado. Retorna o inicio da lista de vértices atualizada e atualiza res com o status da operação.

bool ExistVertice(Node inicio, int id);*

Verifica se existe um vértice com o identificador especificado na lista de vértices inicio. Retorna true se existir, caso contrário, false.

bool ShowGraph(Node graph);*

Exibe o estado atual do grafo, representado pela lista de vértices graph.

bool DestroiVertice(Node ptNode);*

Destrói um vértice específico, libertando a memória associada a ele. Retorna true se a operação for bem-sucedida.

Funções Relacionadas com Adjacências

Adjacent NewAdjacent(int id, int peso);*

Cria uma nova adjacência com o identificador de destino e o peso especificados, retornando um apontador para ela.

Adjacent InsertAdj(Adjacent* listaAdj, int idDestino, int peso);*

Insere uma nova adjacência na lista de adjacências listaAdj com o identificador de destino e o peso especificados. Retorna o inicio da lista atualizada.

bool DestroyAdjacent(Adjacent ptAdjacent);*

Destroi uma adjacência específica, libertando a memória associada a ela. Retorna true se a operação for bem-sucedida.

Adjacent DeleteAdj(Adjacent* listAdj, int codAdj, bool* res);*

Remove uma adjacência específica da lista de adjacências listAdj. Retorna o inicio da lista atualizada e atualiza res com o status da operação.

Adjacent* DeleteAllAdj(Adjacent* listaAdj, bool* res);

Remove todas as adjacências da lista listaAdj. Retorna o inicio da lista vazia e atualiza res com o status da operação.

Funções Relacionadas com Graph

Graph CreateGraph(int* totV, bool* res);*

Cria um novo grafo, inicializando suas estruturas internas, e retorna um apontador para ele. Atualiza totV com o número total de vértices e res com o status da operação.

Graph InsertVertGraph(Graph* G, Node* new, bool* res);*

Insere um novo vértice no grafo G. Retorna o grafo atualizado e atualiza res com o status da operação.

Graph InsertAdjaGraph(Graph* G, int idOrigin, int idDestiny, int peso, bool* res);*

Insere uma nova aresta no grafo G entre os vértices especificados, com o peso dado. Retorna o grafo atualizado e atualiza res com o status da operação.

Graph DeleteAdjGraph(Graph* G, int origin, int destiny, bool* res);*

Remove uma aresta entre os vértices especificados no grafo G. Retorna o grafo atualizado e atualiza res com o status da operação.

Graph WhereIsVertGraph(Graph* G, int idVertice);*

Encontra e retorna um apontador para o vértice com o identificador especificado no grafo G.

Graph DeleteVertGraph(Graph* G, int codVertice, bool* res);*

Remove um vértice do grafo G com o identificador especificado. Retorna o grafo atualizado e atualiza res com o status da operação.

bool ExistVertGraph(Graph inicio, int idVertice);*

Verifica se existe um vértice com o identificador especificado no grafo inicio. Retorna true se existir, caso contrário, false.

bool ShowGraph2(Graph Gr);*

Exibe o estado atual do grafo Gr.

Graph DestroyGraph(Graph* G, bool* res);*

Destrói o grafo G, libertando a memória associada a ele. Retorna true se a operação for bem-sucedida e atualiza res com o status da operação.

Graph ins_vert_adj(Graph* G, Node2* ini, int* vert, int* adj, bool* res);*

Insere vértices e adjacências no grafo G a partir de uma lista Node2 inicial. Atualiza vert e adj com os números de vértices e adjacências, e res com o status da operação.

int SaveGraph(Graph G, char* fileName);*

Salva o estado do grafo G em um arquivo especificado por fileName. Retorna 0 em caso de sucesso, ou um código de erro.

Graph LoadGraphB(char* fileName, bool* res);*

Carrega um grafo a partir de um ficheiro binário especificado por fileName. Retorna o grafo carregado e atualiza res com o status da operação.

Node FindVerticeId(Graph* g, int cod);*

Encontra e retorna um apontador para o vértice com o identificador especificado no grafo g.

Funções de Caminhos e Busca

int CountPaths(Graph g, int src, int dst, int pathCount);*

Conta o número de caminhos entre os vértices src e dst no grafo g, atualizando pathCount com o número de caminhos encontrados.

int CountPathsVertices(Graph g, int src, int dest);*

Conta o número de caminhos entre os vértices src e dest no grafo g. Retorna o número de caminhos encontrados.

bool DepthFirstSearchRec(Graph g, int origem, int dest);*

Realiza uma busca em profundidade recursiva a partir do vértice origem até o vértice dest no grafo g. Retorna true se um caminho for encontrado, caso contrário, false.

Graph ResetVerticesVisitados(Graph* g);*

Faz reset ao status de visita de todos os vértices no grafo g, preparando-o para uma nova travessia.

Funções de Melhor Caminho

Best BestPath(Graph g, int n, int v);*

Encontra o melhor caminho no grafo g com base em critérios específicos entre n e v. Retorna uma estrutura Best contendo os detalhes do caminho.

void ShowAllPath(Best b, int n, int v);

Exibe todos os detalhes do melhor caminho encontrado, conforme a estrutura Best, entre n e v.

Estas funções cobrem uma ampla gama de operações em grafos, desde a criação e modificação até a travessia e análise de caminhos, proporcionando uma base sólida para a manipulação eficiente de grafos em diversas aplicações.

O Desenvolvimento

As decisões tomadas ao longo do desenvolvimento foram guiadas principalmente pela necessidade de criar uma base sólida e funcional que pudesse ser facilmente expandida e melhorada no futuro. Algumas considerações importantes incluem:

Eficiência e Flexibilidade: A escolha por listas ligadas para representar o grafo se baseou na necessidade de uma estrutura de dados que permitisse inserções e remoções dinâmicas de maneira eficiente. Esta abordagem mostrou-se adequada para o grafos em questão.

Persistência de Dados: Optou-se por salvar os grafos em arquivos binários devido à sua eficiência em termos de espaço e velocidade de leitura/escrita, o que é crucial para a manipulação de grandes grafos.

Modularidade e Extensibilidade: A implementação foi feita de maneira modular, com funções bem definidas para cada operação. Isso não só facilita a manutenção do código, mas também permite a adição de novas funcionalidades no futuro sem grandes refatorações.

Simplicidade e Clareza: Manter a implementação simples e clara foi uma prioridade, especialmente para garantir que o código fosse fácil de entender e modificar. Isso foi considerado mais importante do que incluir todas as possíveis otimizações e funcionalidades avançadas desde o início.

Conclusão

O projeto desenvolvido teve como objetivo criar um sistema de manipulação de grafos utilizando listas ligadas. Durante o desenvolvimento, foram implementadas diversas funções para a criação, inserção, remoção e travessia de vértices e arestas, além de operações de preservação e carregamento de grafos a partir de ficheiros. O sistema foi projetado para ser eficiente e flexível, permitindo a manipulação dinâmica dos grafos.

Em conclusão, o trabalho realizado estabelece uma base robusta para a manipulação de grafos, cumprindo os objetivos iniciais e abrindo várias oportunidades para melhorias e expansões futuras. A implementação de funcionalidades adicionais e a otimização do sistema são passos naturais que poderão aumentar significativamente a utilidade e a eficiência deste projeto.