Comentários – Classe Pilha

Como as Estruturas de Dados Foram Utilizadas

Pilha: Para armazenar informações, foi utilizado um array('I'), que consome menos memória. Ela segue a regra do "último a entrar, primeiro a sair" (LIFO) e realiza operações básicas como empilhar, desempilhar e trocar. O tamanho da pilha depende do tamanho do array.

Matriz: O grid 2D é representado por uma lista de listas. Cada posição possui um valor: '0' se estiver preenchida e '1' se estiver vazia. É possível acessar qualquer posição em tempo constante, O(1).

Set de Visitados: Para evitar retrabalho, foi utilizado um conjunto (set) para armazenar as posições já processadas. Assim, é possível verificar rapidamente (O(1)) se uma posição já foi visitada.

Codificação das Posições: As coordenadas 2D foram convertidas em valores 1D utilizando a fórmula pos = linha * cols + col. Essa abordagem economiza aproximadamente 50% de memória em comparação com outras representações.

Estrutura do Código

Parte Principal: Responsável pelo controle do fluxo do programa, exibição das informações e medição de desempenho.

Estruturas de Dados: Foi criada uma classe Pilha com exceções personalizadas para tratar casos de pilha cheia (PilhaCheiaErro), vazia (PilhaVaziaErro) ou com tipo de dado incorreto (TipoErro).

Algoritmos: Foram implementadas duas versões do algoritmo Flood Fill: uma iterativa e outra recursiva.

Entrada/Saída e Utilitários: Responsável por ler e validar arquivos, exibir resultados e capturar os dados fornecidos pelo usuário.

Descrição das Funções

preencher_regiao_pilha(): Implementa o algoritmo Flood Fill utilizando uma pilha explícita. Inicia com a posição inicial, remove a célula do topo da pilha, verifica se pode ser preenchida, preenche e adiciona os vizinhos à pilha. Essa abordagem evita o estouro da pilha de chamadas.

preencher_regiao_recursiva(): Implementa o algoritmo Flood Fill de forma recursiva. Verifica as condições de parada, preenche a célula atual e faz chamadas recursivas para os quatro vizinhos ortogonais.

Pilha.empilha/desempilha(): Realiza as operações fundamentais da pilha, com validação de tipo e limites.

Funções auxiliares: Funções para visualização, validação de entrada e manipulação da matriz.

Complexidade de Tempo e Espaço

Tempo: Ambas as implementações possuem complexidade O(n×m), já que cada célula pode ser visitada no máximo uma vez. No melhor caso (posição inválida), o tempo é O(1). No pior caso (toda a matriz conectada), é O(n×m).

Espaço: Ambas usam O(n×m). A versão iterativa utiliza uma pilha de tamanho O(k) e um conjunto (set) também de O(k), com k ≤ n×m. A versão recursiva utiliza uma pilha de chamadas com complexidade O(k) no pior caso.

Problemas e Soluções

Estouro da Pilha: A versão recursiva não é adequada para matrizes maiores que 1000×1000. Para esses casos, a versão iterativa é mais robusta.

Processamento Repetido: A versão inicial processava células repetidamente. O uso de um conjunto de posições visitadas resolveu esse problema, melhorando significativamente o desempenho.

Validação: Arquivos malformados causavam falhas no programa. Foi implementada uma validação completa, incluindo verificação de codificação, formato e consistência das dimensões.

Decisões Técnicas: A escolha por array em vez de lista reduz o uso de memória. A codificação das posições torna o código mais eficiente, embora mais complexo. A visualização passo a passo facilita o entendimento, mas reduz o desempenho.

Conclusão

O projeto implementou com sucesso o algoritmo Flood Fill em duas abordagens distintas. A versão iterativa demonstrou ser mais adequada para aplicações práticas devido à sua escalabilidade, enquanto a recursiva oferece maior clareza conceitual.

Principais Resultados: Aprofundamento no uso de estruturas de dados, análise de escolhas algorítmicas, implementação robusta com tratamento de erros e otimizações relevantes de desempenho.

Resultados Quantitativos: Ambas as versões funcionaram de forma consistente. A versão iterativa foi aproximadamente 30% mais rápida, suportando matrizes com dimensões de 500×500 ou superiores, com uso eficiente da memória.

Aprendizados: A escolha entre recursão e iteração depende do contexto. Um bom programa não se resume à correção funcional, mas também deve ser robusto, usável e de fácil manutenção. Validação e otimização são aspectos essenciais para aplicações reais.

O desenvolvimento consolidou princípios fundamentais da engenharia de software e da análise algorítmica.

Link github: https://github.com/jfvdrocha/BD-2025.1-desafio1

Nome: João Felipe Veras Dantas Rocha
DRE: 117202753

Nome: Lucas dos Santos Melo
DRE:123312281

Nome: Rafael Augusto Santos
DRE:124155331