O Labirinto do Horror II

Olivia Maite Furquim Araújo Livak

Escola Politécnica — PUCRS

25 de novembro de 2024

Resumo

Este relatório apresenta a solução para o Labirinto do Horror II, desafio do segundo trabalho da disciplina Algoritmos e Estruturas de Dados II. O problema envolve desenvolver uma solução computacional para identificar o número de regiões isoladas que existem em um labirinto e determinar o ser mais frequente em cada região, utilizando o algoritmo de Busca em Profundidade (DFS).

Introdução

O desafio proposto "O Labirinto do Horror II" consiste em analisar e resolver questões relacionadas a um labirinto representado por uma matriz *m* x *n*. Cada célula dessa matriz é codificada em hexadecimal, utilizando 4 bits binários para descrever a presença ou ausência de paredes em suas bordas nas posições superior, direita, inferior e esquerda. Essa configuração é responsável por determinar as conexões possíveis entre as células do labirinto.
Os principais desafios são:

1. **Identificar as regiões isoladas:**

As regiões isoladas são agrupamentos de células conectadas que não possuem comunicação com outras áreas do labirinto devido à presença de paredes nas quatro direções, bloqueando as passagens. A identificação dessas regiões é realizada utilizando algoritmos como a Busca em Profundidade (DFS), que exploram as conexões entre células adjacentes e delimitam as fronteiras de cada região.

2. **Determinar o ser mais frequente em cada região**

Cada célula do labirinto pode conter um ser, representado por caracteres maiúsculos, que devem ser contabilizados para determinar o mais frequente em cada região. Os caracteres e seus respectivos significados são:

A: Anão
B: Bruxa
C: Cavaleiro
D: Duende
E: Elfo
F: Feijão

Desenvolvimento

Processo de solução:

O algoritmo desenvolvido para identificar as regiões isoladas de um labirinto e determinar o ser mais frequente em cada região foi estruturado em três classes, cada uma com um papel essencial, conforme detalhado a seguir, com exemplos de pseudocódigos:

Main: A classe Main solicita ao usuário o caminho de um arquivo e chama o método executar da classe App para iniciar a execução do programa.

Solicitar ao usuário o caminho do arquivo
Ler o caminho do arquivo fornecido
Criar uma instância da classe App
Chamar o método executar da classe App, passando o caminho do arquivo

Labirinto: A classe Labirinto realiza a leitura de um arquivo e representa o labirinto como uma matriz de caracteres. Ela utiliza a busca em profundidade (DFS) para identificar as regiões isoladas, contar o total de regiões encontradas e determinar o ser mais frequente em cada uma

delas. Além disso, a classe implementa um mecanismo para verificar a conectividade entre as células do labirinto, com base nas representações binárias das paredes.

Principais métodos:

leituraArquivo():

O método leituraArquivo é responsável por realizar a leitura de um arquivo que descreve o labirinto e transformá-lo em uma matriz de caracteres. Cada célula dessa matriz contém um caractere que representa um elemento específico, como paredes, caminhos, entradas, saídas e seres.

Função leituraArquivo(caminhoArquivo):
Abrir o arquivo de leitura
Criar uma lista de linhas
Para cada linha no arquivo:
Remover espaços extras e converter para um array de caracteres
Adicionar o array de caracteres à lista de linhas
Fechar o arquivo
Retornar a lista de linhas convertida para um array de arrays de caracteres

nomeDoSer():

O método nomeDoSer retorna o nome correspondente a um caractere que representa um personagem no labirinto. A implementação utiliza um switch para mapear os caracteres válidos aos respectivos nomes dos personagens do labirinto. Caso o caractere não corresponda a nenhum dos seres definidos, o método retorna "Ser inválido".

Função nomeDoSer(ser): Se ser é
'A' então retornar "Anão"
Se ser é 'B' então retornar "Bruxa"
Se ser é 'C' então retornar "Cavaleiro"

Se ser é 'D' então retornar "Duende"

Se ser é 'E' então retornar "Elfo"

Se ser é 'F' então retornar "Feijão"

Caso contrário, retornar "Ser inválido"

encontrarRegiões():

O método encontrarRegiões possui dois objetivos importantes:

- Identificar as regiões conectadas: Cada região é formada por um conjunto de células que estão conectadas entre si, sem serem separadas por paredes nas quarro direções (superior, direita, inferior e esquerda).
- Contagem dos seres em cada região: Verifica a quantidade de seres, representados pelos caracteres A, B, C, D, E e F que existem em cada região encontrada.

Esse método percorre o labirinto, buscando por posições não visitadas e a partir de cada célula não visitada, executa uma busca para explorar toda a região a que ela pertence. Tal busca é realizada através da função dfs, que é chamada recursivamente para explorar as células adjacentes, levando em consideração as paredes do labirinto e evitando visitar posições que foram processadas.

Função encontrarRegioes(labirinto):

Obter o número de linhas (m) e colunas (n) do labirinto

Criar uma matriz de booleanos 'visitado' com o mesmo tamanho do labirinto

Criar uma lista para armazenar as regiões encontradas

Para cada célula no labirinto:

Se não foi visitado:

Criar um contador para os seres na região

Chamar a função dfs para explorar a região e contar os seres

Adicionar o contador da região à lista de regiões

Retornar a lista de regiões

dfs():

O método dfs realiza uma exploração recursiva no labirinto a partir de uma posição inicial, para identificar todos os elementos conectados que não são bloqueados por paredes. Durante a execução, a posição atual é marcada como visitada e caso contenha um ser, a contagem desse ser é incrementada na variável seresContagem. Em seguida, o método explora as posições adjacentes nas quatro direções possíveis: superior (-1, 0), direita (0, 1), inferior (1, 0) e esquerda (0, -1). Para cada direção, verifica-se se a posição está dentro dos limites do labirinto, não foi visitada e não é bloqueada por uma parede. Se as condições forem atendidas, a busca continua recursivamente, até que todas as posições conectadas à inicial sejam visitadas e os seres presentes nelas contados.

Função dfs(labirinto, m, n, i, j, visitado, seresContagem):
Marcar a célula como visitada
Obter o valor da célula (ser)
Se o valor for uma letra maiúscula (um ser):
Incrementar a contagem desse ser
Definir as direções possíveis para explorar (cima, direita, baixo, esquerda)
Para cada direção:
Calcular a posição da próxima célula (nx, ny)
Se a célula estiver dentro dos limites do labirinto e não foi visitada:
Se a célula for acessível (não bloqueada):
Chamar dfs recursivamente para explorar a célula

estaDentro():

O método estaDentro verifica se as coordenadas (x, y) estão dentro dos limites válidos de uma matriz com dimensões *m* x *n*.

Função estaDentro(x, y, m, n):
Retornar verdadeiro se a célula (x, y) estiver dentro dos limites do labirinto

paredes():

O método paredes verifica a presença de uma parede entre dois elementos, representados pelos caracteres valorAtual e proximoValor, em uma direção específica. Em primeiro lugar, os valores hexadecimais desses elementos são convertidos para números binários de 4 bits, onde cada bit indica a presença de uma parede nas quatro direções possíveis. Em seguida, o método verifica se, na direção indicada não há parede em valorAtual, indicando um '0' nesse bit e se, na direção oposta, calculada como (direcao + 2) % 4, também não há parede em proximoValor. Se essas condições forem verdadeiras, o método retorna true, indicando que não há paredes entre os dois elementos nas direções especificadas; caso contrário, retorna false.

Função paredes(valorAtual, proximoValor, direcao):
Converter os valores atuais e próximos para representações binárias
Verificar se há uma parede entre as células baseando-se na direção
Retornar verdadeiro se as células não estiverem bloqueadas pela parede

contarRegioes():

O método contarRegioes retorna a quantidade de regiões presentes em um labirinto.

Função contarRegioes(regioes):
Retornar o número de regiões encontradas

obterSerMaisFrequente():

O método obterSerMaisFrequente busca o ser que aparece com maior frequência entre todas as regiões de um labirinto. Ele percorre cada região, comparando a quantidade de vezes que cada ser aparece. Se encontrar uma contagem maior, atualiza o ser e a frequência.
Ao final, retorna o nome do ser mais frequente e o número de vezes que ele apareceu.

Função obterSerMaisFrequente(regioes):
Inicializar variável serMaisFrequente com '-'
Inicializar variável maiorFrequencia com 0

Para cada mapa de seres nas regiões

Para cada entrada (ser, contagem) no mapa de seres

Se a contagem for maior que maiorFrequencia

Atualizar maiorFrequencia e serMaisFrequente

Retornar o ser mais frequente e sua contagem

App: A classe App gerencia a execução do programa, lendo o labirinto, identificando regiões isoladas e determinando o ser mais frequente em cada uma delas. Ela exibe os resultados e calcula o tempo de execução do algoritmo.

Marcar o tempo de início da execução (startTime)

Tentar

Chamar o método leituraArquivo da classe Labirinto passando o caminho do arquivo

Armazenar o labirinto retornado em uma variável (labirinto)

Chamar o método encontrarRegioes da classe Labirinto, passando o labirinto

Armazenar as regiões isoladas em uma lista (regioes)

Contar o número de regiões isoladas chamando o método contarRegioes da classe Labirinto

Exibir o número de regiões

Obter o ser mais frequente chamando o método obterSerMaisFrequente da classe Labirinto

Exibir o ser mais frequente e sua quantidade

Caso ocorra um erro de leitura

Exibir mensagem de erro

Marcar o tempo de fim da execução (endTime)

Calcular e exibir o tempo total de execução (duration)

Resultados:

A tabela a seguir apresenta os resultados dos casos de teste executados, destacando a quantidade de regiões isoladas encontradas em cada labirinto e o ser mais frequente em cada região, juntamente com a quantidade de vezes que ele apareceu.

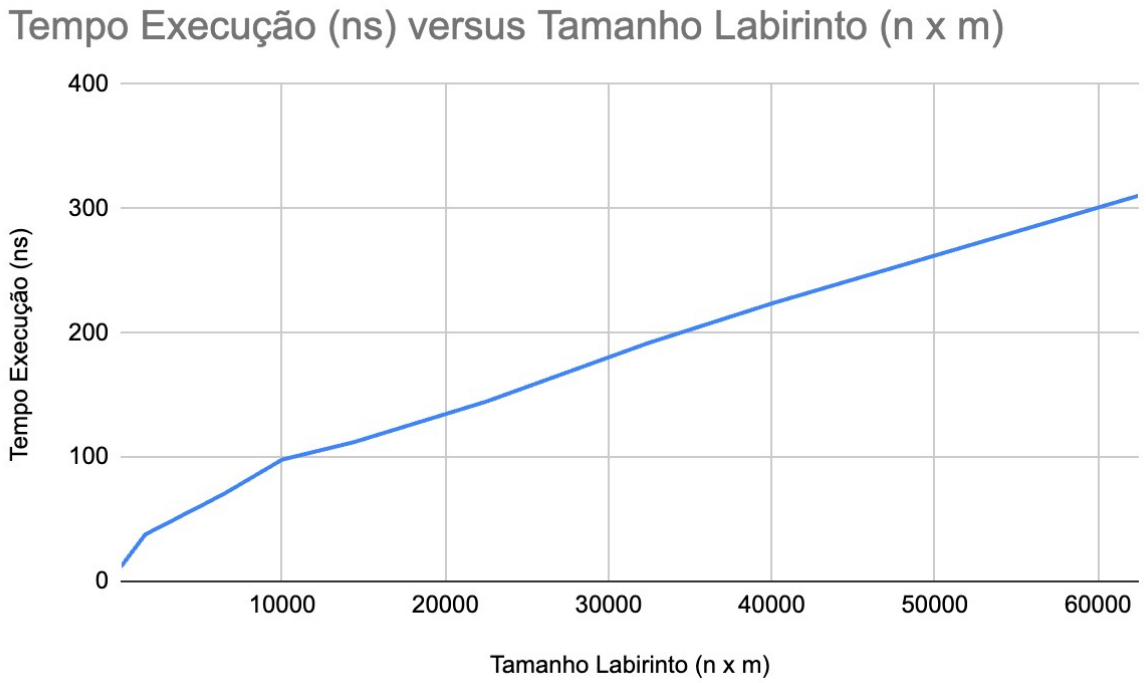
Caso de Teste	Quantidade de regiões	Ser mais frequente
caso10.txt	3	Anão (1 vez)
caso40.txt	18	Bruxa (3 vezes)
caso80.txt	19	Elfo (14 vezes)
caso100.txt	61	Cavaleiro (10 vezes)
caso120.txt	89	Anão (14 vezes)
caso150.txt	98	Anão (16 vezes)
caso180.txt	140	Cavaleiro (19 vezes)
caso200.txt	9	Anão (203 vezes)
caso250.txt	224	Cavaleiro (20 vezes)

Análise do algoritmo

Abaixo, são apresentados os tempos de execução em nanosegundos para cada um dos casos de teste executados. Esses resultados permitem avaliar a eficiência do algoritmo em termos de tempo, considerando diferentes tamanhos de labirinto. Os dados foram coletados para labirintos com tamanhos variados, e o gráfico subsequente ilustra a relação entre o tamanho da matriz ($n \times m$) e o tempo de execução.

Caso de teste	Tempo de execução em nanosegundos
caso10.txt	11565333 nanosegundos
caso40.txt	37673208 nanosegundos
caso80.txt	70194625 nanosegundos
caso100.txt	97979875 nanosegundos
caso120.txt	112058709 nanosegundos
caso150.txt	144654833 nanosegundos

caso180.txt	191634542 nanosegundos
caso200.txt	223575542 nanosegundos
caso250.txt	310405208 nanosegundos



Com a análise do gráfico, é possível perceber que o algoritmo desenvolvido para solucionar o problema fictício possui complexidade linear entre o tamanho do labirinto ($n \times m$) e o tempo de execução. Essa complexidade é eficiente para um problema que envolve exploração de labirinto, pois, à medida que o tamanho do labirinto aumenta, o tempo de execução cresce de maneira proporcional.

Como executar o programa:

Os casos de teste estão organizados na pasta do código-fonte. Para executar o programa, basta informar o nome do arquivo correspondente ao labirinto que deseja analisar. O programa processará os dados, retornando a quantidade de regiões isoladas identificadas, o ser mais frequente em cada uma delas e o tempo de execução do caso de teste.

Conclusão

O desenvolvimento do trabalho foi fundamental para aprofundar a compreensão sobre grafos, em especial o algoritmo de busca em profundidade para resolver um problema de exploração de labirinto. A solução proposta ao longo do relatório permitiu aplicar conceitos na prática para identificar regiões isoladas e determinar o ser mais frequente em cada uma delas.

A solução consistiu em representar o labirinto como uma matriz, onde cada célula possui informações sobre suas conexões, com base nas paredes descritas por valores hexadecimais. O algoritmo de busca em profundidade foi utilizado para explorar cada região conectada do labirinto, mapeando as células visitadas e agrupando-as em regiões isoladas. Durante essa exploração, o algoritmo também contou a ocorrência dos seres representados por caracteres nas células, possibilitando identificar o ser mais frequente em cada região.

Os resultados dos testes realizados mostraram que o algoritmo é capaz de identificar corretamente o número de regiões isoladas e os seres mais frequentes em cada uma delas, conforme esperado. O retorno de informações para cada caso de teste validou a eficácia da solução e sua capacidade de lidar com diferentes tamanhos de labirintos.

Além disso, a análise dos tempos de execução destacou a eficiência do algoritmo, que apresentou complexidade linear em relação ao tamanho do labirinto ($n \times m$). Esse comportamento evidencia que cada célula foi processada uma única vez, resultando em um desempenho adequado para labirintos de grandes dimensões.