O Labirinto do Horror II

Maria Rita Rodrigues[*], Murilo dos Santos[†]

Escola Politécnica — PUCRS

25 de novembro de 2024

Resumo

Este artigo descreve o desenvolvimento de duas soluções alternativas para resolver o problema proposto no segundo trabalho da disciplina de Algoritmos e Estruturas de Dados II, no semestre de 2024/2. O problema consiste em percorrer um labirinto, identificar suas regiões isoladas e determinar as criaturas que aparecem com maior frequência em cada uma dessas regiões.

Introdução

O segundo trabalho da disciplina de Algoritmos e Estruturas de Dados II propõe a resolução de um problema que envolve a análise e exploração de labirintos. O enigma é baseado em uma narrativa fictícia, onde a escavação arqueológica em Creta revela os mistérios de um labirinto ancestral que possui diversas regiões isoladas, dificultando a interação entre seus habitantes — Anões, Bruxas, Cavaleiros, Duendes, Elfos e Feijões. A proposta consiste em identificar essas regiões e analisar quais criaturas são mais frequentes em cada uma.

O labirinto é representado por uma matriz de dimensões $m \times n$, em que cada célula é codificada em hexadecimal com um valor de 4 bits. Esses bits indicam a presença ou ausência de paredes nas quatro direções (superior, direita, inferior e esquerda) e determinam a conectividade entre as células. Assim, a matriz pode conter regiões completamente isoladas, formadas por células interligadas que não têm acesso a outras áreas do labirinto.

[1] maria.rita04@edu.pucrs.br
[2] m.kasperbauer@edu.pucrs.br

```
4 4
9 A a 6
5 9 A c
5 5 F 3
7 3 A e
```

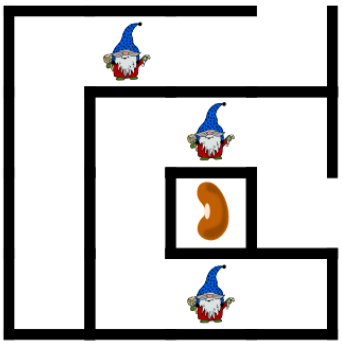

Figura 1 - Exemplo de labirinto com três regiões isoladas. Neste labirinto, o ser que mais aparece numa região é o Anão (representado pela letra 'A').

As criaturas presentes no labirinto são representadas por caracteres maiúsculos no código hexadecimal de cada célula, como 'A' para Anões, 'B' para Bruxas e 'C' para Cavaleiros. Cada criatura ocupa uma posição na matriz e pode estar presente em uma ou mais regiões. A análise dessas criaturas inclui identificar sua distribuição e apontar o ser mais frequente em cada região isolada, contribuindo para uma compreensão detalhada da dinâmica do labirinto.

```
6 6
9 E b A a E
3 a E b 8 8
D d 9 a 4 7
5 5 3 e 5 D
1 0 C 9 2 6
7 7 5 3 a e
```

Figura 2 - Labirinto com quatro regiões isoladas por cor.
O Elfo ('E') é o personagem mais frequente na matriz exibida à esquerda.

Os labirintos utilizados no trabalho são fornecidos em arquivos no formato ***.txt***, onde cada célula é descrita por um caractere hexadecimal. A conversão desses valores para binário resulta em 4 bits, que indicam a presença de paredes em cada uma das direções:

- **Bit 1 (superior):** parede superior;
- **Bit 2 (direita):** parede direita;
- **Bit 3 (inferior):** parede inferior;
- **Bit 4 (esquerda):** parede à esquerda.

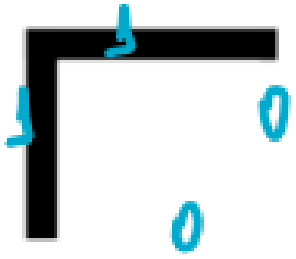

Figura 3 - Exemplo visual sobre a codificação binária das paredes em 4 bits.

As células conectadas formam regiões, que podem ser exploradas para identificar sua estrutura e ocupação. Cada região isolada é composta por células interligadas, sem acesso a outras regiões. O desafio é, portanto, implementar um algoritmo capaz de:

- Determinar o número total de regiões isoladas;
- Identificar o tipo de criatura mais frequente em cada região.

Primeira Solução

Nesta seção, iremos descrever a primeira solução adotada para o problema proposto através de pseudocódigos. Antes de iniciar a implementação do algoritmo, foi essencial realizar a leitura do arquivo .txt, o que foi feito seguindo a seguinte lógica:

```
1  classe Main:
2      método main(args):
3          tenta:
4              cria um objeto Labirinto com o arquivo "casos4/caso40_4.txt"
5              chama o método imprimirGrafo() do objeto Labirinto
6          captura exceção IOException:
7              imprime a mensagem de erro
```

Nesta solução, tentamos usar a estrutura *Union-Find* e grafos para representar o labirinto, o que parecia ser uma boa ideia para identificar as regiões isoladas e fazer as operações necessárias. A proposta era transformar cada célula do labirinto em um vértice de um grafo e, com base nas paredes (definidas pelos valores hexadecimais), criar conexões entre as células. A estratégia de usar o *Union-Find* para agrupar as células conectadas parecia eficiente, e também queríamos contar os tipos de seres dentro dessas regiões.

classe Labirinto:

Atributos:

linhas: número de linhas do labirinto

colunas: número de colunas do labirinto

labirinto: matriz representando o labirinto

Construtor:

***Recebe** o caminho do arquivo*

***Chama** o método carregarLabirinto passando o caminho do arquivo*

Método carregarLabirinto(arquivo):

***Abrir** o arquivo para leitura*

***Ler** as dimensões do labirinto (linhas e colunas)*

***Inicializar** a matriz 'labirinto' com as dimensões lidas*

***Para** cada linha do labirinto:*

    ***Ler** a linha do arquivo*

    ***Converter** cada valor hexadecimal e armazenar na matriz*

Método indice(linha, coluna):

***Retorna** o índice linear correspondente à posição (linha, coluna) na matriz*

Método limites(x, y):

***Verifica** se as coordenadas (x, y) estão dentro dos limites da matriz*

Método construirGrafo():

***Inicializa** uma lista de listas chamada 'grafo'*

Para cada célula do labirinto (representada por suas coordenadas i, j):

Obter o valor da célula

Verificar as direções (direita, baixo, esquerda, cima):

    Se a célula permite ir nessa direção e o destino está dentro dos limites:

        Adicionar uma aresta entre a célula atual e o destino no grafo

Retornar o grafo construído

Dificuldades

Apesar de termos idealizado como o *Union-Find* poderia ser utilizado para resolver o problema, nos perdemos um pouco durante a implementação. A principal falha foi termos implementado as operações essenciais como *find* e *union* de forma incorreta, o que impediu que as células fossem unidas corretamente e as regiões isoladas fossem identificadas.

Percebemos que as dificuldades encontradas durante a implementação tem origem no fato de que não havíamos compreendido o problema completamente, pelo menos não além da teoria. No começo, a ideia era usar todos os códigos que vimos em aula, mas depois vimos que isso não era necessário — apenas algumas classes da Interface de Programação de Aplicações (API) de grafos precisam, de fato, ser utilizadas. Então, fizemos uma reavaliação do código, refatoramos tudo e utilizamos as matrizes do próprio enunciado (imagens 1 e 2) como base para validar o funcionamento do algoritmo apresentado na segunda solução.

Segunda Solução

Nesta segunda implementação, as classes *CaminhamentoPorProfundidade* e *Graph* nos auxiliam a explorar e entender a organização do labirinto. A classe *Graph* organiza o labirinto como um grafo, onde cada célula é representada como um ponto (vértice) e as conexões entre células sem parede são representadas por linhas (arestas). Isso facilita a identificação de células conectadas. A classe *CaminhamentoPorProfundidade* realiza a busca no labirinto, deslocando-se de uma célula para as outras conectadas. Por ser uma busca por profundidade, a caminhada é feita até atingirmos o "nível" mais profundo possível a partir de uma célula até outra, identificando assim as regiões interligadas.

Juntas, as classes *CaminhamentoPorProfundidade* e *Graph* nos permitem caminhar pelo labirinto e identificar as regiões conectadas. Como ambas fazem parte da API de grafos, não as incluímos nas apresentações dos pseudocódigos, mas essa é a explicação de como foram utilizadas nesta implementação.

Sobre a implementação: a classe ***Labirinto*** representa o labirinto e sua lógica de funcionamento. Ela tem alguns atributos importantes: a interface gráfica (**gui**), a matriz que mostra o labirinto (**labirinto**), suas dimensões (**m** e **n**), e outros atributos como o vetor de deslocamento para as direções do labirinto (**dx** e **dy**), um mapa para armazenar as regiões e seus habitantes (**regioesSeres**), e o grafo (**graph**) que organiza as conexões entre as células.

```
1   classe Labirinto:
2       atributos:
3           gui             // interface gráfica do labirinto
4           labirinto       // matriz representando o labirinto
5           m, n            // dimensões do labirinto
6           dx, dy          // vetores de deslocamento para as 4 direções
7           regioesSeres    // mapa de regiões e seus habitantes
8           graph           // grafo representando conexões no labirinto
9   Construtor:
10      Inicializar gui, labirinto, dimensões m e n
11      Inicializar regioesSeres como mapa vazio
12      Criar grafo com tamanho baseado no número total de células do labirinto
```

O método **identificarSer** determina o tipo de ser em uma célula do labirinto, como "Anão", "Bruxa" ou "Cavaleiro", com base no caractere presente nela. Já o **coordToVertex** converte as coordenadas de uma célula (linha e coluna) para um índice linear, posteriormente usado no grafo, enquanto **vertexToCoord** faz a conversão inversa, retornando as coordenadas correspondentes a um índice do grafo.

```
13  método identificarSer(ser):
14      Se ser for 'A' Retornar "Anão"
15      Se ser for 'B' Retornar "Bruxa"
16      Se ser for 'C' Retornar "Cavaleiro"
17      Se ser for 'D'  Retornar "Duende"
18      Se ser for 'E'  Retornar "Elfo"
19      Se ser for 'F'  Retornar "Feijão"
20      Senão  Retornar "Desconhecido"
21  método coordVertex(i, j):
22       Retornar índice linear baseado em i, j
23  método vertexCoord(v):
```

```
24          Retornar coordenadas i, j correspondentes ao índice linear v
```

O método **construirGrafo** cria as conexões entre as células do labirinto, verificando a presença de paredes em cada direção e adicionando arestas no grafo para as células vizinhas. Em seguida, **processarLabirinto** utiliza esse grafo para mapear as regiões conectadas por meio de uma busca por profundidade, agrupando as células e atualizando a interface gráfica. **contarRegioes** calcula o número de regiões isoladas com seres, enquanto resultados exibe o número total de regiões e detalhes sobre cada uma, como o ser mais frequente e suas ocorrências.

```
25  método construirGrafo():
26          Para cada célula do labirinto Faça
27                  Se a célula contiver um caractere válido
28                          Converter a célula para um número de paredes
29                          Para cada direção (d) Faça
30                                  Se não houver parede na direção d
31                                          Calcular nova posição (ni, nj)
32                                          Se (ni, nj) for válida
33                                                  Adicionar uma aresta entre os vértices das
                                                    posições
34                                  fim Se
35                          fim Se
36                  fim Para
37          fim Se
38  fim Para
39  Criar matriz de regiões para a interface
40  Atualizar a interface gráfica com as regiões
41  método contarRegioes():
42          Inicializar conjunto regioesUnicas
43          Para cada célula do labirinto Faça
44                  Se a célula conter um ser
```

```
45                          Adicionar a região correspondente ao conjunto
46                      fim se
47                  fim para
48          Retorna o tamanho de regioesUnicas
49          método resultados():
50                  Exibir o número de regiões encontradas:
51                  Exibir os detalhes de cada região, incluindo:
52                   - Região, ser mais frequente, ocorrências do ser e o símbolo do ser
```

A classe **SeresRegiao** tem como principal objetivo organizar e contar as criaturas encontradas nas diferentes regiões do labirinto. Ela mantém um **mapa** (atributo contagem) que armazena a quantidade de cada tipo de ser encontrado em uma região específica. O método **adicionarSer** é responsável por adicionar uma nova criatura ao mapa ou atualizar a contagem de uma criatura já registrada.

Se o ser já existir no mapa, seu valor é incrementado; caso contrário, ele é inserido com o valor inicial de 1. O método **serMaisFrequente** percorre o mapa e retorna o ser que aparece mais vezes, ou um espaço em branco caso não haja nenhum ser registrado.

Por fim, o método **getContagem** simplesmente retorna o mapa com as contagens de todas as criaturas armazenadas, permitindo o acesso aos dados de contagem.

```
1   classe SeresRegiao:
2           atributos:
3                   contagem
4   construtor:
5           Inicializar contagem como um mapa vazio
6   método adicionarSer(ser):
7           Se ser já existe em contagem
8                   Incrementar o valor associado a ser
9           Senão
10                  Adicionar ser ao mapa com valor 1
```

```
método serMaisFrequente():
        Encontrar o ser com maior contagem no mapa contagem
        Retornar  o ser mais frequente ou ' ' se não houver
método getContagem():
        Retorna o mapa contagem
```

Usamos a classe **UnionFind** para identificar as regiões conectadas no labirinto porque ela é eficiente para agrupar células vizinhas. Ela organiza esses grupos com dois atributos principais: parent, que define o "líder" de cada conjunto, e size, que guarda o tamanho de cada conjunto, otimizando o processo de união entre regiões. Quando duas células estão conectadas, podemos uni-las em uma única região. O UnionFind facilita a verificação rápida de quais células pertencem a uma mesma região e permite unir regiões vizinhas de maneira eficiente, tornando o processo de identificação e agrupamento das áreas conectadas no labirinto mais simples e rápido.

```
classe UnionFind:
        atributos:
                parent   // vetor que armazena o "pai" de cada elemento
                size     // vetor que armazena o tamanho do conjunto de cada elemento
construtor(n):
        Para i = 0 até n - 1 Faça
                parent[i] = i       // cada elemento é seu próprio pai
                size[i] = 1         // cada conjunto tem tamanho 1 inicialmente
```

O método **find** encontra o líder (ou raiz) do conjunto ao qual um elemento pertence. Se o elemento não for seu próprio líder, ele segue a cadeia de pais até chegar à raiz. Durante esse processo, é realizada a compressão de caminho, onde os elementos visitados são diretamente conectados à raiz, o que posteriormente torna as buscas mais rápidas.

```
método find(x):
        Se parent[x] != x
                parent[x] = find(parent[x])
                // caminho comprimido: faz o pai de x apontar para o avô
```

```
12          Retorna parent[x]
```

Já o método **union** une dois conjuntos ao encontrar suas raízes e, se forem diferentes, conecta o conjunto menor ao maior, mantendo a árvore balanceada e melhorando a performance das operações seguintes.

```
13  método union(x, y):
14          raizX = find(x)                 // encontra a raiz do conjunto
15          raizX = find(x)                 // encontra a raiz do conjunto
16          Se raizX != raizY               // se x e y estão em conjuntos diferentes
17  Se size[raizX] < size[raizY]
18          parent[raizX] = raizY           // une o conjunto de x ao de y
19          size[raizY] += size[raizX]      // atualiza o tamanho do conjunto
20  Senão
21          parent[raizY] = raizX           // une o conjunto de y ao de x
22          size[raizX] += size[raizY]      // atualiza o tamanho do conjunto
```

Dificuldades

A lógica inicial do código funcionava, mas era desorganizada e pouco clara, dificultando sua compreensão e explicação. A exibição dos resultados também estava confusa, e os identificadores das regiões, baseados em índices não sequenciais do Union-Find, precisaram ser ajustados para usar índices em ordem crescente, tornando a apresentação mais intuitiva.

Também faltavam métodos claros, como uma função para imprimir o labirinto antes do processamento, o que dificultava a depuração. A lógica de conversão de caracteres hexadecimais em binários também gerou confusões, prejudicando a identificação de paredes. Além disso, enfrentamos dificuldades com estruturas como HashMap, integração do Union-Find e aplicação de conceitos teóricos, levando à revisão do código para corrigir problemas.

Além disso, a ausência de validações adequadas na leitura de arquivos e nos limites da matriz expôs o código a falhas. A interface gráfica não apresentou corretamente o

progresso durante a coloração das regiões isoladas, comprometendo a experiência. Por fim, houve desafios na contagem de elementos do texto, como identificar criaturas (letras maiúsculas) e números hexadecimais (letras minúsculas).

Terceira Solução

Na última solução, focamos em corrigir os erros encontrados na solução 2 e fizemos algumas melhorias no código, buscando otimizar o Union-Find, a leitura dos arquivos de entrada e a validação. A classe principal continua responsável por iniciar a execução, criar o objeto labirinto e chamar o método para imprimir o resultado, conforme ilustrado no exemplo de código:

```
1  classe Main:
2      Iniciar Scanner para leitura do caminho do arquivo
3      Exibir mensagem pedindo caminho do arquivo
4      Ler o caminho do arquivo
```

Depois de obter o caminho, o código tenta abrir o arquivo para leitura. A primeira verificação que o programa realiza é se o arquivo contém o número de linhas (m). Se esse dado estiver faltando ou estiver no formato errado, um erro é gerado. Em seguida, o código verifica o número de colunas (n) e também valida esse dado.

```
5      Tentar:
6          Abrir o arquivo especificado para leitura
7          Verificar se o arquivo contém número de linhas (m)
8          Se não houver número de linhas, lançar erro "Formato inválido"
9              Ler o número de linhas (m)
```

Só então criamos a matriz *labirinto* com o número de *linhas (m)* e *colunas (n)* lidas do arquivo. Ele preenche essa matriz com os caracteres correspondentes, realizando uma nova verificação para garantir que o arquivo contém todos os dados necessários para completar o labirinto.

```
10         Criar matriz labirinto com m linhas e n colunas
```

```
11          Preencher a matriz labirinto com os caracteres do arquivo
12          Para cada linha e coluna no arquivo:
13              Se houver próximo caractere, colocar na matriz
14              Se não houver caractere, lançar erro "Labirinto com formato inválido"
```

Após o carregamento da matriz, o programa exibe uma mensagem confirmando o sucesso do carregamento do arquivo. Em seguida, ele inicializa a interface gráfica, que será responsável por mostrar o labirinto de maneira visual para o usuário.

Após isso, o programa cria uma instância da classe **Labirinto**, que vai processar o labirinto, identificar regiões e mostrar os resultados. A interface gráfica é então atualizada para transmitir essas mudanças.

```
15          Exibir mensagem "Arquivo carregado com sucesso"
16          Exibir mensagem "Inicializando interface gráfica..."
17          Inicializar a interface gráfica (GUI)
18          Executar a interface gráfica
19          Criar instância da classe Labirinto com o labirinto e a GUI
20          Processar o labirinto e identificar as regiões
21          Exibir os resultados da análise
22          Atualizar a visualização do labirinto na GUI
23      Capturar erro de IO:
24          Exibir mensagem de erro "Erro ao tentar abrir o arquivo"
25      Capturar outros erros:
26          Exibir mensagem de erro "Erro ao processar o arquivo"
27      Fechar Scanner após o processamento
28      Exibir mensagem "ANÁLISE COMPLETA! CONFIRA OS RESULTADOS"
```

Já a implementação da classe Labirinto busca identificar regiões conectadas e as criaturas presentes. A inicialização do labirinto ocorre:

No início da classe, são definidas as direções possíveis de movimentação no labirinto (**cima**, **direita**, **baixo**, **esquerda**). A classe também declara variáveis importantes, como a

interface gráfica (**gui**), a matriz do labirinto (**labirinto**), o número de linhas e colunas (**m** e **n**), o mapa que armazena as regiões e seus seres (**regioesSeres**), e o grafo que representa as conexões entre as células (**graph**).

```
1   classe Labirinto:
2       construtor(labirinto, gui):
3           Validar entrada (labirinto e gui não podem ser nulos)
4           Armazenar labirinto e gui em variáveis locais
```

O método **validarEntrada()** verifica se o labirinto e a interface gráfica são válidos. Se algum dos parâmetros for nulo, ou se o labirinto estiver vazio, o método lança um erro. O método **validarLabirinto()** percorre cada linha do labirinto para garantir que todas tenham o mesmo tamanho e que cada célula contenha um caractere válido, seja ele uma letra, número ou espaço.

```
5   método validarEntrada():
6       Se labirinto ou gui forem nulos, lançar erro
7       Se labirinto estiver vazio, lançar erro
8   método validarLabirinto():
9       Para cada linha do labirinto:
10          Se as linhas tiverem tamanhos diferentes, lançar erro
11          Para cada célula do labirinto:
12              Se o caractere não for válido, lançar erro // não for letra, número ou espaço
```

O método **construirGrafo()** converte o labirinto em um grafo, onde cada célula se torna um vértice. Para cada célula do labirinto, o método chama **processarCelula()**, que verifica se ela é válida (se é um caractere permitido) e, em seguida, conecta as células vizinhas, caso não haja paredes entre elas.

```
13  método construirGrafo():
14      Para cada célula no labirinto:
15          Chamar o método processarCelula()
16  método processarCelula(coord):
```

```
17          Se a célula não for válida (não for letra ou número), retornar
18          Converter o caractere para um valor de parede
19          Para cada direção (cima, direita, baixo, esquerda):
20              Se não houver parede, conectar a célula ao vizinho
```

O método **processarLabirinto()** é responsável por realizar todo o processo. Ele começa com a construção do grafo e, em seguida, identifica as regiões conectadas no labirinto. Além disso, ele atualiza a interface gráfica para exibir as regiões encontradas. Caso algum erro ocorra, uma exceção é lançada.

```
21  método processarLabirinto():
22          Tentar:
23              Construir o grafo
24              Identificar as regiões conectadas no labirinto
25              Atualizar a interface gráfica com as regiões identificadas
26          Se houver erro, lançar exceção
```

Para identificar as regiões, o método **identificarRegioes()** mapeia cada célula do labirinto e utiliza o algoritmo de busca em profundidade (DFS) para agrupar as células conectadas. Quando uma nova célula é encontrada, o algoritmo processa a região e associa as criaturas a ela, quando necessário.

```
27  método identificarRegioes():
28          Inicializar mapa vazio verticeParaRegiao
29          Para cada célula do labirinto:
30              Se a célula for válida e ainda não tiver sido processada, processar a
                região
31          Retornar mapa de regiões
```

Após identificar as regiões, o método **atualizarInterface()** atualiza a interface gráfica, exibindo a região correspondente a cada célula no labirinto.

```
32  método atualizarInterface(verticeParaRegiao):
```

```
33          Para cada célula do labirinto:
34              Atualizar a região associada a cada célula na interface gráfica
```

A classe **SeresRegiao** é fundamental para a classe **Labirinto**. Ela funciona como um contador inteligente, guardando as informações sobre os seres (representados por letras maiúsculas) encontrados em uma determinada região do labirinto. Além disso, ela prioriza os seres mais frequentes, para que possamos rapidamente identificar qual deles aparece mais. Assim como podemos ver no código exemplo:

```
1   classe SeresRegiao:
2       construtor:
3           atributos
4           Inicializar mapa de contagem de seres (contagem)
5           Inicializar fila de prioridade para frequência (filaFrequencias)
6           Definir um limite máximo de seres por região (MAX_SERES = 1000)
```

O método **adicionarSer()** verifica se o caractere é válido, atualiza a contagem e reorganiza a fila de prioridade. Caso o limite de seres seja atingido ou algum erro ocorra, ele lança uma mensagem explicando.

```
7   método adicionarSer(ser):
8       Se ser não for uma letra maiúscula:
9           Lançar erro "Ser inválido"
10      Se número de seres exceder MAX_SERES:
11          Lançar erro "Limite máximo atingido"
12      Atualizar contagem do ser no mapa (adicionar ou incrementar)
13      Se houver erro de contagem (exemplo: overflow):
14          Lançar erro "Erro na contagem"
15      Remover entrada anterior do ser na fila de prioridade (se existir)
16      Adicionar nova entrada do ser na fila de prioridade
17      Se ocorrer erro na fila de prioridade:
```

```
18                  Exibir mensagem de erro
```

O método **serMaisFrequente()** procura no mapa o ser com a maior contagem e o retorna. Se não houver nenhum ser, retorna um espaço em branco.

```
19  método serMaisFrequente():
20          Se o mapa de contagem for vazio:
21                  Retornar ' ' (nenhum ser registrado)
22          Procurar o par (ser, frequência) com a maior frequência no mapa:
23                  Para cada entrada (ser, frequência) no mapa:
24                          Comparar frequências e encontrar o maior valor
25          Se o ser encontrado não for uma letra maiúscula:
26                  Lançar erro "Ser inválido na contagem"
27          Retornar o ser encontrado
```

A classe **TipoSer** é um enum que organiza os diferentes tipos de seres encontrados no labirinto, associando a cada tipo um caractere (**simbolo**) e um nome (**nome**). A criação dessa classe eliminou a necessidade de utilizar constantes ou estruturas de controle, como o switch ou if-else, para mapear os caracteres em nomes de seres. O que, por fim, tornou o código mais modular e estruturado.

Cada constante da enum (ANAO, BRUXA, CAVALEIRO, etc.) é uma instância de **TipoSer**, inicializada com valores específicos para o símbolo e o nome. Por exemplo: ANAO('A', "Anão"), que cria uma constante para representar o anão, onde 'A' é o caractere associado, e "Anão" é o nome descritivo. Os principais métodos da classe são:

- **getSimbolo() e getNome():**
    - Esses dois métodos são bem simples e diretos. O primeiro devolve o caractere que representa o tipo de ser (por exemplo, 'A' para Anão), enquanto o segundo retorna o nome desse ser (como "Anão");

- **fromChar(char c):**
    - Pega um caractere e transforma na instância correspondente de TipoSer. Ele funciona como um filtro, percorrendo todos os tipos de seres definidos na enum (usando o values()) e comparando o caractere passado com os símbolos

disponíveis. Quando encontra um correspondente, ele devolve o tipo de ser certo; se não encontrar, retorna null.

Por fim, a classe **Coordenada** é utilizada para representar as posições no labirinto e facilitar operações, como validação e mapeamento entre matrizes e grafos.

```
1   classe Coordenada:
2       atributos:
3           i       // linha da coordenada
4           j       // coluna da coordenada
5   construtor(i, j):
6       Atribuir i à linha
7       Atribuir j à coluna
```

O método **toVertex** converte uma coordenada em um índice único de vértice, útil para representar o labirinto como um grafo. O índice é calculado multiplicando a linha pelo número de colunas e somando a coluna (ideia também implementada na solução 2 e refinada na solução final).

```
8   método toVertex(n):
9       Retornar i * n + j
```

Enquanto isso, o método **fromVertex** converte um índice de vértice de volta para uma coordenada (i, j). A linha é obtida dividindo o índice pelo número de colunas, e a coluna é o resto dessa divisão.

```
10  método fromVertex(v, n):
11      Calcular linha como v/n
12      Calcular coluna como v% n
13      Retornar nova Coordenada(linha, coluna)
```

O método isValid verifica se a coordenada está dentro dos limites do labirinto, validando que as posições estão entre 0 e as dimensões máximas (m e n).

```
10  método isValid(m, n):
```

```
11          Se i >= 0 e i < m e j >= 0 e j < n:
12              Retornar verdadeiro
13          Senão:
14              Retornar falso
```

Dificuldades encontradas

Durante o desenvolvimento dessa nova versão do código, encontramos problemas de implementação simples que foram resolvidos com laços de repetição para validação, mas também problemas mais complexos com o desenvolvimento de Union Find da forma correta, que anteriormente não fazia as conexões todas necessárias com algumas células, da mesma forma que alguns ajustes na lógica de verificação de barreiras e na manipulação de índices. O formato de saída também precisou ser revisto, que foram causados por especificadores dentro do código para que fizesse a formatação correta e o alinhamento dos valores da tabela.

Resultados

Para testar o funcionamento geral do algoritmo, utilizamos os labirintos fornecidos no enunciado da imagem como fonte. Na imagem abaixo, podemos ver os resultados encontrados para o labirinto da **figura 1**:

```
Digite o caminho do arquivo do labirinto (ex.: casos4/nomeDoCaso.txt):
src/casos4/enunciado6.txt

Arquivo carregado com sucesso!

Inicializando interface gráfica...

ANÁLISE COMPLETA! CONFIRA OS RESULTADOS:

1. Número de regiões isoladas: 4

2. Detalhes de cada região:
-------------------------------------------------------------------------
| Região     | Ser mais frequente       | Ocorrências     | Símbolo    |
-------------------------------------------------------------------------
| 1          | Elfo                     | 2               | E          |
| 2          | Anão                     | 1               | A          |
| 3          | Duende                   | 1               | D          |
| 4          | Cavaleiro                | 1               | C          |
-------------------------------------------------------------------------
```

Figura 4 - Resultado obtido após utilizar o segundo labirinto disponível no enunciado do trabalho.

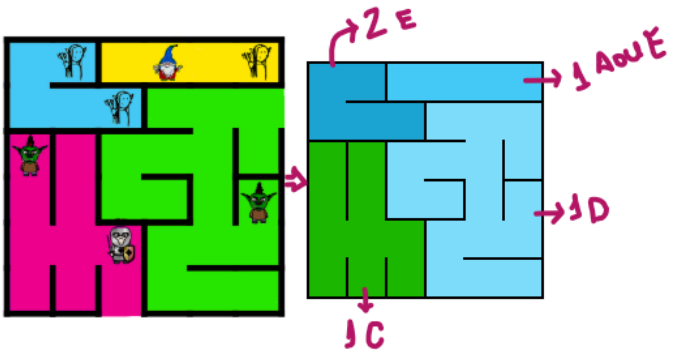

Figura 5 - Labirinto da figura 1 à esquerda e o labirinto da interface à direita. As setas indicam quais foram as saídas obtidas sobre a criatura mais frequente por região.

Após a implementação do algoritmo, executamos uma série de testes com os casos fornecidos para avaliar o seu desempenho. A tabela a seguir apresenta a quantidade de regiões isoladas encontradas em cada caso de teste:

Caso	Quantidade de Regiões Isoladas
caso 40	37
caso 80	82
caso 100	74
caso 120	31
caso 150	120
caso 180	172
caso 200	10
caso 250	73

Para analisar a complexidade, utilizamos a biblioteca ***time*** e a função ***time()***, que registra o número de segundos passados durante a execução do algoritmo. Cada caso foi executado seis vezes, e os tempos registrados foram usados para calcular uma média simples, a tabela a seguir apresenta estes resultados:

Caso	Tempo Médio (segundos)
caso 40	4.25426064 segundos
caso 80	6.19372830 segundos
caso 100	2.41920428 segundos
caso 120	5.96359935 segundos
caso 150	5.69426179 segundos
caso 180	3.71361184 segundos
caso 200	5.12571349 segundos
caso 250	6.15876770 segundos

Ao analisar o tempo de execução com base nos casos de teste, observamos que o tempo de execução varia de forma irregular entre os casos de teste. Por exemplo, o caso 150, com mais regiões isoladas, apresenta um tempo mais alto, enquanto o caso 100, apesar de ter muitas regiões, têm um tempo menor. Isso indica que o desempenho do algoritmo é influenciado por características específicas dos dados, como sua distribuição, e não apenas pela quantidade de regiões isoladas. Isso revela a necessidade de otimização em casos mais complexos.

Para ilustrar esta análise de complexidade, geramos um gráfico utilizando a biblioteca *Matplotlib* do *Google Colab*, que apresenta o comportamento do tempo de execução em relação ao número de entradas. A imagem gerada foi a seguinte:

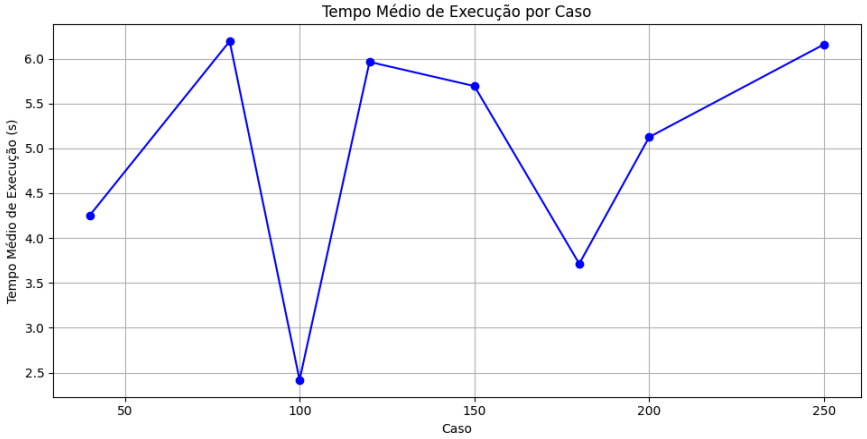

Figura 6 - O gráfico mostra como o tempo de execução varia de forma irregular conforme os casos de teste aumentam, com picos e quedas no tempo médio à medida que o número de casos muda.

Após aplicarmos técnicas adequadas de análise de complexidade, como o uso de escala logarítmica nos eixos x e y, verificamos se o gráfico segue uma tendência polinomial. Se, ao aplicar as escalas logarítmicas, o gráfico se assemelhar a uma linha reta, isso indica uma tendência polinomial.

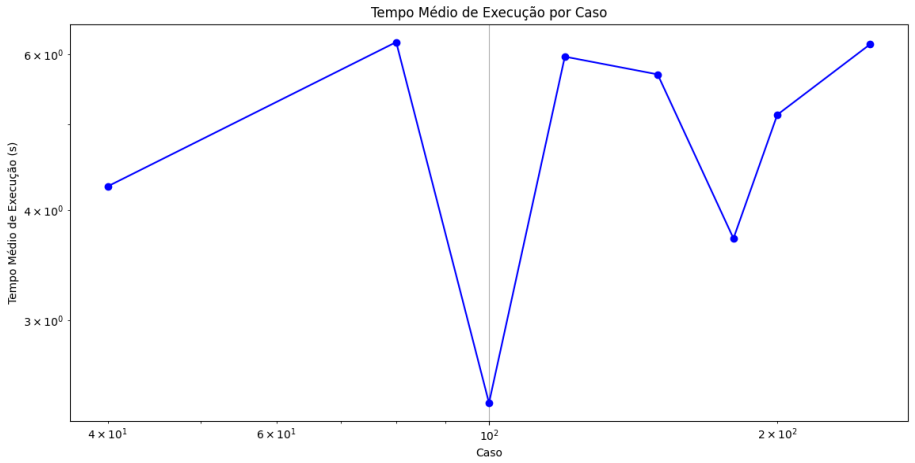

Figura 7 - O gráfico com escala logarítmica mostra variações irregulares no tempo de execução em relação aos casos de teste, sugerindo que o desempenho não segue uma relação simples com o número de casos.

No caso deste gráfico, a curva apresenta variações significativas, mas é possível identificar tendências gerais. Em regiões específicas, o comportamento se aproxima de uma relação linear no espaço logarítmico, o que nos sugere que a complexidade pode estar relacionada a uma função polinomial. No entanto, desvios da linearidade em outros trechos indicam possíveis fatores externos ou comportamentos atípicos do algoritmo.

Calculando a ordem de complexidade:

- Para $n_1 = 40$ e $T_1 = 4.25426064$ **segundos.**
- Para $n_2 = 250$ e $T_2 = 6.15876770$ **segundos.**

Utilizando a fórmula:

- $b \approx (\log(T_2) - \log(T_1)) / (\log(n_2) - \log(n_1))$

Calculando os logaritmos de T_1 e T_2:

- $\log(T_1) = \log(4.25426064) \approx 0.6288$
- $\log(T_2) = \log(6.15876770) \approx 0.7892$

Calculando os logaritmos de n_1 e n_2:

- $\log(n_1) = \log(40) \approx 1.6021$
- $\log(n_2) = \log(250) \approx 2.3979$

Diferença dos logaritmos T e n:

- $\log(T_2) - \log(T_1) = 0.7892 - 0.6288 = 0.1604$
- $\log(n_2) - \log(n_1) = 2.3979 - 1.6021 = 0.7958$

Por fim:

- $b \approx 0.1604/0.7958 \approx 0.2019$

A ordem de complexidade estimada é de b ≈ 0.202. Este valor nos confirma que o crescimento do tempo de execução em relação ao tamanho do caso é muito lento, o que nos sugere uma complexidade sublinear.

Conclusão

A solução desenvolvida para identificar regiões e criaturas em um labirinto evoluiu ao longo do processo. A primeira etapa foi focada na leitura do arquivo e na conversão da matriz, que apresentou problemas na conexão entre células.

A implementação do Union Find foi um ponto chave para identificarmos as regiões conectadas de maneira mais eficiente. Garantimos que os ajustes necessários fariam o processamento das conexões corretamente, respeitando as barreiras e tornando a leitura do labirinto mais eficaz.

Adicionamos também a introdução do paralelismo com Stream, que melhorou inicialmente o desempenho nos casos de teste do labirinto maiores, enquanto a modularização do código facilita na manutenção. Essas melhorias ajudam a manter a solução mais eficiente.

Por fim, acreditamos que esta solução tenha mostrado a importância de ajustes iterativos e um maior refinamento contínuo, mostrando que sempre vamos ter possibilidade para melhorias.