De A até B já é bem longe

Enzo Russo Rodrigues

Engenharia de Software – PUCRS

15 de junho de 2020

**Resumo**

Este artigo descreve alternativas de solução para o segundo problema proposto na disciplina de Algoritmos e Estruturas de Dados II do curso de Engenharia de Software. Como somos um dos três especialistas mundiais em labirintos para filmes, fomos contratados por um estúdio que produz filmes de franquia de super-herói que deseja, para seu próximo filme, um gigantesco labirinto onde os heróis deverão procurar, até achar, os vilões para que comecem uma batalha épica. O roteiro é bem simples, como podemos ver, mas nosso trabalho nem tanto. O objetivo é fazer com que o herói ache o vilão o mais rápido possível e com o menor número de movimentos, ou seja, fazer com que percorra o caminho mais curto para isso.

**Introdução**

Um estúdio que produz filmes de super-heróis deseja inovar no seu próximo filme. A ideia é ter um labirinto gigante que funciona como o mapa fictício do filme e os produtores nos procuraram para que possamos ajudar o herói a chegar no vilão, que estará parado criando um plano maligno para pôr em prática quando chegar a hora. Como o filme é dividido em duas etapas, perseguição e batalha, o tempo é algo importante. Para o herói não ficar percorrendo caminhos sem fim, devemos descobrir o comprimento do menor caminho dele até o vilão, para equilibrar o tempo das etapas.

Após analisar o problema e o nosso trabalho, o próximo passo é receber o projeto dos labirintos dos estudos preliminares e apresentar o menor comprimento do herói até o vilão.

Conheceremos o método de funcionamento dos labirintos no próximo tópico, assim, devemos nos concentrar em concluir o serviço. Mesmo o filme sendo uma história inverídica (afinal, estamos falando de super-heróis), os produtores cobram autenticidade nos nossos meios de percorrer labirintos, logo querem saber como as respostas foram obtidas e o tempo que custou.

**Entendendo os labirintos**

O labirinto contém os símbolos explicitados abaixo, cujo cada um segue regras, seja para as direções ou restrições e permissões dos movimentos dos personagens:

**A** representa um herói genérico que se move para cima, para baixo, esquerda e direita, mas não nas diagonais;

**B** representa um vilão genérico que está parado fazendo seu plano maligno;

**#** representa uma parede impenetrável;

**.** representa um espaço livre por onde o herói poderá se mover.

Sendo assim, o labirinto que usaremos para testar nossa solução assume este formato:

#########

#A..#...#

#.#...#.#

#.#.....#

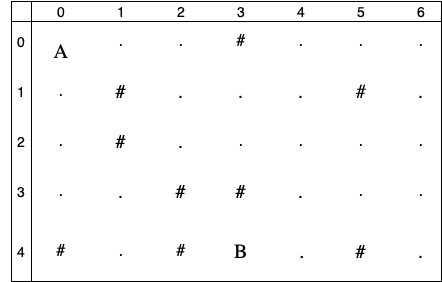
#..##...#

##.#B.#.#

#########

**Transição dos labirintos para estrutura**

Para representar o labirinto em uma linguagem de programação (neste caso, Java), foi criado uma matriz de tipo *char* que guarda a informação de qual caractere está na posição identificada pela linha e coluna. Criamos uma matriz (linha e coluna) de booleanos para informar a posição visitada. Quando lemos um A ou B, significa que achamos nossos personagens, então salvamos as posições deles em variáveis do objeto Coordenadas, uma classe que conta com os métodos getX(), getY() e getPai() para acessar a informação armazenada em cada posição. Podemos ver a transição do labirinto do exemplo para uma matriz na imagem abaixo (tiramos apenas o entorno do labirinto já que era somente parede e começamos no A):



Podemos perceber que, nessa imagem, o herói está na posição (0, 0), enquanto o vilão está na (4, 3).

Para explicar como fizemos a transformação da primeira imagem, que é o arquivo do labirinto que recebemos, para a segunda imagem, a matriz, segue abaixo um pseudocódigo que traduz o que é feito pelo código fonte em Java:

char[][] Labirinto

boolean[][] visitado

Coordenadas heroiPos

Coordenadas vilaoPos

String quebraLinha = arquivo\_de\_entrada.split(“\n”)

labirinto = new char[linhas.tamanho][linhas[pos 0].tamanho]

visitado = new boolean[linhas.tamanho][linhas[pos 0].tamanho]

para linha = 0; linha < altura; linha++:

para coluna = 0; coluna < largura; coluna++:

se quebraLinha[linha].charAt(coluna) == ‘#’:

labirinto[linha][coluna] = ‘#’

senão se quebraLinha[linha].charAt(coluna) == ‘A’:

labirinto[linha][coluna] = ‘A’

heroiPos = new Coordenadas(linha, coluna)

senão se se quebraLinha[linha].charAt(coluna) == ‘B’:

labirinto[linha][coluna] = ‘B’

vilaoPos = new Coordenadas(linha, coluna)

senão labirinto[linha][coluna] = ‘.’

Resolvemos representar as possibilidades de movimento do herói na forma de um grafo (em: <https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bfs.html#sec:queuevsstack>), com os personagens sendo os nodos e o espaço livre (‘.’) suas arestas. Foi criado uma fila para que pudéssemos implementar o algoritmo de busca em largura (do inglês, *BFS* – *breadth-first search*), que está relacionada com os conceitos de distância e caminho mínimo. Como o herói pode se mover para baixo, para cima, esquerda e direita, usamos uma matriz de inteiros para armazenar os valores das direções. Os critérios escolhidos foram estes abaixo:

{0, 1} – personagem se move apenas no eixo x e, como é 1, se move para direita;

{0, -1} – personagem se move apenas no eixo x e, como é -1, se move para esquerda;

{1, 0} – personagem se move apenas no eixo y e, como é 1, se move para baixo;

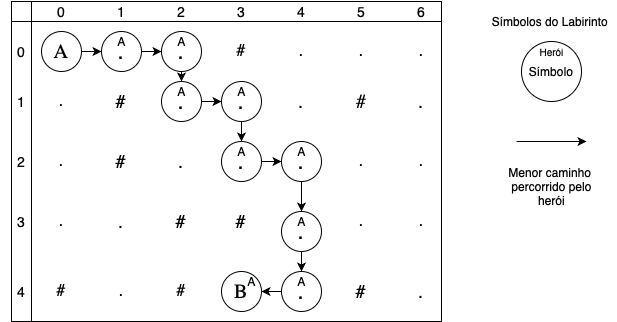
{-1, 0} – personagem se move apenas no eixo y e, como é -1, se move para cima.

A busca começa a partir da posição do herói, que já foi reconhecida na classe Labirinto (com heroiPos e vilaoPos). O algoritmo todos os vizinhos do herói, depois todos os vizinhos dos vizinhos, e assim por diante. Sendo assim, para o caso do exemplo temos um desenho de caminhamento em grafo parecido com este:

Teclado de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Ao lado, temos as posições adicionadas na fila começando pela (0, 0), que é a posição do herói e paramos na do vilão (4, 3), que é a última armazenada. Para adicionarmos as próximas posições, visitamos os vizinhos adjacentes que ainda não foram visitados. Continuamos o processo, evitando as paredes, até acharmos o vilão e paramos a busca imediatamente (perceba que a posição (4, 6) não foi visitada e assim seria caso tivéssemos mais posições). Afinal, não nos importa mais se ainda podemos chegar no vilão por algum outro caminho já que este é o método que percorre o menor caminho e, logo, o menor número de movimentos. (Abaixo, uma imagem que representa melhor o caminho percorrido pelo herói. Todas as posições são nodos, mas para poupar desenho representamos apenas os símbolos que fazem parte do caminho como nodos).



Para exemplificar melhor as imagens e a explicação sobre a estratégia *BFS*, podemos acompanhar o pseudocódigo do método caminhaLabirinto (que recebe um objeto Labirinto como parâmetro) da classe GrafoBFS a seguir:

int[][] DIRECOES = {{0,1}, {1,0}, {0,-1}, {-1,0}}

int contMovimentos;

Queue<Coordenadas> proxAVisitar = new Queue<>()

Coordenadas heroi = getHeroiPos() //inicio da busca

proxAVisitar.enqueue(herói) //adiciona o herói na fila

enquanto proxAVisitar não está vazio:

Coordenadas atual = proxAVisitar.dequeue //retira um vértice da fila

ehLocalValido(atual.getX, atual.getY)

se não é um ehLocalValido ou se ehExplorado:

continua

se ehParede(atual.getX, atual.getY):

foiVisitado(atual.getX, atual.getY, true)

continua

se ehVilaoPos(atual.getX, atual.getY):

return backTrackPath(atual)

para cada int[] vizinho em DIRECOES:

Coordenadas c = new Coordenadas(atual.getX + vizinho [pos 0], atual.getY + vizinho [pos 0], atual) //numera o vizinho

proxAVisitar.enqueue(c) //adiciona vizinho na fila

foiVisitado(atual.getX, atual.getY, true)

return Queue.<Coordenadas>emptyQueue()

Os métodos getHeroiPos, ehLocalValido, ehExplorado, ehParede, ehVilaoPos e foiVisitado são provenientes da classe Labirinto, então deveremos referenciá-los. Eles são instanciados recebendo a posição atual por parâmetro, sendo que o método foiVisitado conta um valor extra booleano para que fosse possível sinalizar como uma *flag* que aquela posição foi visitada. O retorno do método se dá por duas maneiras diferentes. A primeira delas é no terceiro laço *if*, na qual checamos se a posição em que nos encontramos é a do vilão. Caso isso seja verdade, o método retorna direto para um outro método, o backTrackPath (que recebe um objeto Coordenadas como parâmetro), que funciona como um processo iterativo para voltar atrás de um determinado nodo até sua raiz (pai). É utilizado para rastrear o caminho quando o vilão é encontrado e segue a estrutura em Java do pseudocódigo abaixo:

Queue<Coordenadas> caminho = new Queue<>()

Coordenadas iterador = atual //atual é a variável passada por parâmetro

enquanto iterador for diferente de nulo:

caminho.enqueue(iterador)

iterador = iterador.getPai()

contMov++

retorna caminho

Usamos uma fila neste método acima por conta do retorno do método caminhaLabirinto. Como o retorno do caminhaLabirinto é uma fila, temos que receber uma variável de mesmo tipo do método que é chamado.

Vamos falar sobre o segundo retorno no método caminhaLabirinto. Por que é daquele jeito? A resposta é simples. Usamos a prática do *Singleton*, o que significa que o método sempre retornará a mesma instância. Sabendo que o método retorna uma fila sempre, mesmo que vazia, essa prática evita a checagem por *null*.

Na classe Queue, fizemos uma adaptação do método da classe Collections do Java que retorna listas vazias (método Collections.emptyList(), que utiliza @SuppressWarnings(“unchecked”)). Trocamos o tipo do objeto de List para Queue. Era possível utilizar a classe nativa e o método já disponível, porém optamos por trabalhar apenas com uma estrutura de dados no grafo.

**Resultados**

Como dito anteriormente, após implementarmos o algoritmo em Java, nosso trabalho era testá-lo em 8 casos diferentes. Executamos também o exemplo do enunciado, para ter uma noção se estávamos no caminho certo. Os resultados obtidos foram estes da tabela abaixo:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nome do caso | Tamanho do Labirinto (largura x altura) | Número de Movimentos | Comprimento caminho mais curto | Tempo (em segundos) |
| Caso do Enunciado | 40x30 | 49 | 50 | 0,057 |
| casoa | 100x100 | 177 | 178 | 0,065 |
| casob | 200x200 | 456 | 457 | 0,087 |
| casoc | 500x500 | 393 | 394 | 0,157 |
| casod | 1000x1000 | 764 | 765 | 0,407 |
| casoe | 2000x2000 | 2687 | 2588 | 1,844 |
| casof | 5000x5000 | 7587 | 7588 | 16,484 |
| casog | 10000x10000 | 13670 | 13671 | 140,045 |
| casoh | 20000x20000 | – | – | – |

A partir das resoluções, podemos constatar que, dos 8 casos propostos para o teste, os 5 primeiros (6, com o do enunciado) obtiveram bons tempos de execução (praticamente 0 segundos do caso do enunciado até o *casod* e menos de 2 segundos no *casoe*) e todos, com exceção ao último, tiveram resultados confiáveis. Já o *casof* teve um crescente no tempo de execução referente aos anteriores. Podemos atrelar esse motivo ao fato do labirinto mais que dobrar de tamanho, aumentando muito também o número de movimentos. No *casog*, fica evidente o comportamento do algoritmo para casos extremamente grandes. Por mais que tenha demorado bastante tempo para rodar, ainda conseguimos extrair um resultado do teste, coisa que não aconteceu para o último caso.

À medida que o tamanho do labirinto aumenta, percebemos também que o tempo de execução piora agressivamente. No *casog*, tivemos um problema inesperado. Quando executamos o algoritmo para o caso, nos deparamos com o seguinte erro impresso no terminal: “java.lang.OutOfMemoryError: Java heap space”. Após pedir ajuda para outros especialistas em labirintos para filmes e muitas pesquisas na internet sobre o erro, pensamos em todas as possíveis soluções e fomos botá-las em prática. Identificamos o que poderia ser o erro, o fato de estarmos criando duas matrizes (uma do tipo *char* e outra do tipo *boolean*) na classe Labirinto, estava ocupando muita memória. Então, tentamos abrir mais espaço na memória com a função “-Xmx512M”, “-Xmx1024M” e “-Xmx2048M” (que aumentam 512mb, 1gb e 2gb, respectivamente), usada junto com o comando de execução do programa pelo terminal, mas o erro continuou. Foi decidido criar um *ArrayList* de *ArrayList* para substituir cada matriz, na tentativa de solucionar o problema e a mudança foi retornar o erro mais rápido no terminal que anteriormente. Depois, tentamos criar um *HashMap* para usar o critério de chave, valor. Utilizamos o tipo *Integer* como sendo a chave e mantivemos os mesmos valores das matrizes, uma *boolean* e a outra *char*, mas também não nos solucionou o problema. Em matéria de tempo de execução, todos os testes resultaram tempos bem parecidos (para alguns casos até piores). Assim, optamos em manter as matrizes, já que de um jeito ou outro não conseguimos fazer com que o programa não ultrapassasse o limite da memória, sendo esse o único problema encontrado para solucionar os labirintos.

**Conclusão**

Com o desenvolvimento de uma solução para o problema proposto, acredito que conseguimos manter nosso posto entre os três especialistas na criação de labirintos para filmes. Um fator que desmotivou a resolução de um bom trabalho, foi o fato do último labirinto, aquele que mais importava para o estúdio, não ter um resultado para que o herói possa se basear.

Infelizmente, tivemos um problema justamente no labirinto que seria utilizado para as gravações do filme, mas, caso o estúdio resolva fazer um curta-metragem, terá os resultados dos outros labirintos.