Trabalho 1: Siga o dinheiro

Eduardo Monteiro Tavares*, Murilo Machado da Silva†
Faculdade de Engenharia de Software — PUCRS

10 de abril de 2024

Resumo

Este artigo descreve alternativas de solução para o primeiro problema proposto na disciplina de Algoritmos e Estruturas de Dados II no semestre 2024/1, que trata de solucionar um problema de mapeamento. É apresentado uma solução ao problema, junto da explicação da estrutura de dados adotada e o motivo de sua escolha. Em seguida os resultados obtidos e as conclusões tiradas.

Introdução

No infortúnio evento de um grande assalto ao banco, bandidos realizaram uma fuga pela cidade, deixando notas de dinheiro caírem pelo caminho, a polícia precisa seguir a quadrilha e recolher as notas deixadas para trás. Para isso, a equipe de perícia tracejou múltiplos mapas, que infelizmente acabaram por ser longos e confusos. O presente trabalho consiste em desenvolver um algoritmo capaz de percorrer o caminho descrito nos mapas, contabilizando quanto dinheiro foi recolhido no final da perseguição.

Dentro do escopo da disciplina de Algoritmos e Estruturas de Dados II, o primeiro passo a ser discutido nesse problema seria a estrutura de dados que será utilizado na sua solução.

O enunciado informa da existência de mapas, que indicam uma trilha que deve ser seguida e números no caminho que devem ser contabilizados, seguindo então 7 regras:

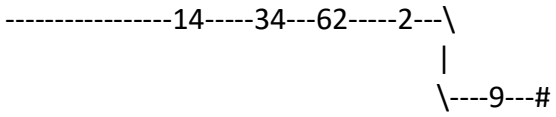
1. A primeira informação exibida em cada mapa é seu tamanho, em linhas e colunas, respectivamente.
2. A trilha sempre começa em algum ponto do lado esquerdo, com o próximo passo sendo à direita.

* eduardo.tavares002@edu.pucrs.br

† murilo.012@edu.pucrs.br

3. O mapeamento é linear, sendo interrompido caso encontre os símbolos “/” ou “\”, o qual indicam uma curva.
4. Números podem ser contabilizados duas vezes caso estejam em cruzamentos, indicado pelo símbolo “|”.
5. Quando o símbolo “#” for atingido, isso indica o fim do percurso.
6. Os números encontrados no caminho devem ser armazenados na ordem em que foram encontrados.
7. Deve-se contabilizar os números na ordem em que estão representados. Isso é, se o caminho percorrido está sendo da esquerda à direita, e no percurso há o número “82”, deve-se ler 82, caso o caminho tenha sido no sentido inverso, deve-se ler 28.

Por exemplo, seguindo estas regras temos o seguinte mapa de demonstração.



Nesse exemplo, o mapeamento começa a esquerda, segue contabilizando os números 14, 34, 62 e 2, encontra o símbolo “\” indicando uma curva para a direita abaixo, seguindo a reta “|”, e se curvando a direita com o símbolo “\”, seguindo o caminho, onde encontra o 9 e finalmente encerra seu percurso ao encontrar “#”. No final, todos os números são somados, no respectivo exemplo, a soma seria 121.

O problema a ser resolvido é a implementação de um algoritmo que resolva esse caminhamento, seguindo as regras de caminhamento e somando os números no caminho encontrados, indicando ao final o resultado dessa soma.

Para resolver a questão proposta, será analisado uma alternativa de solução com a implementação de um algoritmo que resolva esse caminhamento utilizando de uma estrutura de dados ideal. Em sequência será apresentado os resultados obtidos, bem como as conclusões tiradas desse trabalho.

Solução

Após uma análise detalhada do problema em questão, constatamos que o primeiro passo seria o desenvolvimento de um algoritmo capaz de transferir o conteúdo de arquivos de texto para uma estrutura de matriz. Essa abordagem viabilizaria a leitura eficiente do conteúdo, caractere a caractere, facilitando futuras manipulações e caminhamentos. Para atender a essa demanda, foi concebida a classe *Leitor*, encarregada de efetuar a leitura do arquivo e armazenar seu conteúdo em uma estrutura de dados apropriada.

O método *leitorArquivo(String path)* foi elaborado para realizar essa tarefa de forma eficaz, através de uma leitura genérica de arquivos, processando cada linha e construindo uma string, no caso, uma sequência de caracteres, contendo o conteúdo completo.

Em seguida, a classe “Matriz” desempenha o papel da transformação da string gerada em uma matriz. Para isso, utiliza-se o algoritmo *transformarEmMatriz(String conteudo)*, responsável pela conversão. Esse algoritmo divide o conteúdo em um arranjo bidimensional, no caso, em uma matriz, identificando assim as dimensões partir da primeira linha. Durante a iteração pelas linhas do conteúdo, o método preenche a matriz com os caracteres correspondentes, garantindo que cada posição no arranjo seja totalmente preenchida. Caso uma linha do conteúdo seja mais curta do que a largura especificada pela matriz, as posições restantes são preenchidas com espaços em branco, permitindo uma conversão mais precisa e completa.

```
função transformarEmMatriz(conteudo)
    String[] linhas = dividir conteudo em linhas usando a quebra de linha
    String[] dimensoes = dividir primeira linha de linhas usando espaço
    int largura = converter dimensoes[0] para inteiro
    int altura = converter dimensoes[1] para inteiro
    matriz = criar matriz de caracteres com altura x largura
    para cada i de 0 até altura - 1
        linha = linhas[i + 1]
        para cada j de 0 até largura - 1
            se j < comprimento da linha
                matriz[i][j] = caractere na posição j da linha
```

```
            senão
                matriz[i][j] = espaço em branco
        retornar matriz
```

Inicialmente, a implementação do algoritmo acima enfrentou um desafio relacionado à iteração sobre as linhas do arquivo. A tentativa de atribuir todos os valores de uma linha diretamente à matriz resultou em um erro de *IndexOutOfBoundsException* neste trecho:

```
for (int j = 0; j < largura; j++) {
            matriz[i][j] = linha.charAt(j);
        }
```

Com a finalidade de solucionarmos esse problema, adotamos uma abordagem mais robusta. Uma validação foi implementada para garantir que cada caractere da linha fosse atribuído à matriz de maneira adequada. Isso assegurou que todas as linhas fossem processadas corretamente, mesmo que o comprimento delas variasse.

```
matriz[i][j] = j < linha.length() ? linha.charAt(j) : ' ';
```

Após a resolução deste impasse, implementamos o algoritmo *encontrarHifenEsquerda(),* o qual identifica a linha onde o hífen é encontrado na primeira coluna da matriz, indicando assim o ponto de partida do caminho. O método percorre todas as linhas da matriz e verifica se o primeiro caractere seria o hífen. Caso sim, retornará o índice da linha onde está localizado; caso contrário, retornará -1, indicando assim que nenhum hífen foi encontrado.

```
    função encontrarHifenEsquerda():
        para cada linha i na matriz:
            se o primeiro caractere da linha i for um hífen:
                retornar i
```

```
    retornar -1
```

É importante ressaltarmos que este método foi de extrema utilidade para estabelecermos uma referência inicial na leitura do arquivo, conforme especificado, representando o ponto de partida fundamental para o subsequente caminhamento necessário.

Com a implementação bem-sucedida da conversão do conteúdo da string para matriz e a criação do algoritmo para identificar o início do caminho, avançamos para a elaboração do algoritmo principal da nossa aplicação: *percorrerCaminho(int xInicial)*. Este algoritmo desempenha um papel central em nosso sistema, sendo responsável por realizar o caminhamento necessário de acordo com a especificação dentro da matriz gerada a partir do arquivo de texto.

O parâmetro *xInicial* indica o índice da linha de início do percurso, previamente determinado pelo algoritmo de identificação do início do caminho. A seguir, vamos detalhar minuciosamente o funcionamento deste método, demonstrando sua importância e contribuição para a solução do problema proposto.

```
função percorrerCaminho(xInicial):
    soma = 0
    x = xInicial
    y = 0
    novoNum = 0
    direction = Direction.DIREITA
    isEnd = falso
    numStr = novo StringBuilder()
...
```

Apresentaremos abaixo primeiramente os atributos do algoritmo em questão, detalhados em tópicos para proporcionar uma compreensão clara de suas funcionalidades:

- *soma*: inicialmente configurada como zero, esta variável é usada para acumular a soma dos números encontrados durante o percurso;

- *x*: inicialmente sendo atribuído ao valor do parâmetro *xInicial*, representa a linha inicial para percurso;
- *y*: inicialmente em zero, é usada para rastrear a posição da coluna durante o percurso;
- *novoNum*: inicialmente em zero, é usada para armazenamento do número construído a partir dos caracteres da matriz durante o percurso;
- *direction*: inicialmente atribuído com *Direction.DIREITA*, representa a direção do movimento do caminhamento após a leitura do caractere inicial. É imprescindível destacarmos que este atributo seria do tipo *Direction*, o qual foi uma enumeração (coleção de constantes nomeadas) criada para definir as direções possíveis para o percurso da matriz, representando cada uma um movimento específico ao longo das linhas ou colunas da matriz;
- *isEnd*: inicialmente atribuído como *falso*, é usada como uma sinalização para identificar se o percurso foi concluído ou não;
- *numStr*: inicialmente sendo vazio, este atributo do tipo *StringBuilder* é empregado para construirmos números a partir dos caracteres durante o percurso, caso o número seja composto por mais de um caractere.

Após a explicação dos atributos fundamentais do algoritmo principal, avançaremos com uma análise detalhada de suas operações, começando pela estratégia adotada para lidar com números que possuem múltiplos dígitos.

```
...
    enquanto não atingir o fim do caminho:
        atual = matriz[x][y]

        se atual for um dígito:
            numStr.acrescentar(atual)
            novoNum = converter numStr para inteiro

        senão:
            soma += novoNum
            novoNum = 0
```

```
            numStr.limpar()
...
```

No trecho acima, caso o caractere atual for um dígito, ele será adicionado à string *numStr*, a qual acumula os caracteres que representam um número. Caso o caractere atual não seja um dígito e a string *numStr* não esteja vazia, isso implica que a sequência de dígitos encontrados foi finalizada. Nesse caso, a string *numStr* é convertida em um número inteiro denominado *novoNum*, que é somado à variável de acumulação denominada 'soma', limpando posteriormente a variável *numStr* para acumular o próximo número encontrado durante a iteração subsequente.

Dedicando a devida atenção ao problema proposto, foi possível verificarmos que os caracteres '/' e '\' requeriam de uma lógica mais robusta ao serem encontrados. Abaixo, segue nossas validações. Explicaremos cada uma detalhadamente:

```
...
      se atual for '/':
            se direction for Direção.DIREITA:
                direction = Direção.CIMA
                x--
            senão se direction for Direção.ESQUERDA:
                direction = Direção.BAIXO
                x++
            senão se direction for Direção.CIMA:
                direction = Direção.DIREITA
            senão
                direction = Direção.ESQUERDA
...
```

Nesta parte do algoritmo, estamos lidando com a situação em que o caractere atual da matriz seria '/', indicando assim uma mudança de direção no caminho percorrido pelo algoritmo. A direção de movimento é controlada pela variável *direction*, que pode assumir valores diferentes, como para CIMA, BAIXO, DIREITA e ESQUERDA.

Essa lógica é aplicada em todos os cenários, sempre verificando a direção em que o caminhamento estava ocorrendo, para que o percurso prossiga conforme necessário. É importante destacarmos que, quando tratamos da direção horizontal, seja à direita ou à esquerda, não se faz necessário o incremento ou decremento de posição na linha X, dado que, ao mover horizontalmente a matriz, estamos mantendo a linha, apenas alterando, posteriormente, nas próximas validações de caminhamento, a linha Y.

O algoritmo para quando o caractere atual for '\' foi desenvolvido da mesma forma que o algoritmo para validação do caractere '/', apenas com a adaptação da lógica conforme necessidade. No exemplo abaixo, podemos ver que, caso o caminho venha da direita e encontre o caractere '\', ele irá para baixo, uma vez que seria esse o percurso que precisamos:

```
...
      se atual for '\':
            se direction for Direção.DIREITA:
                direction = Direção.BAIXO
                x++;
...
```

Após efetuarmos as validações dos caracteres que indicam mudança de direção, prosseguimos com as verificações essenciais para garantir o correto encaminhamento ao longo de todo o percurso.

```
...
            se a direção for DIREITA:
                  incrementar y
            se y for maior ou igual ao limite da matriz na direção y:
                  interromper
            senão se a direção for ESQUERDA:
                  decrementar y
            se y for menor que 0:
                  interromper
            senão se não for "/" ou "\" e a direção for PARA CIMA:
```

```
            decrementar x
      se x for menor que 0:
            interromper
      senão se não for "/" ou "\":
            incrementar x
      se x for maior ou igual ao limite da matriz na direção x:
            interromper
```

Este algoritmo controla o movimento ao longo da matriz de acordo com a direção atual, caso não seja o caractere ‘/’ ou ‘\’. Se a direção é para a direita, a coordenada Y é incrementada para mover-se para a direita. Caso a coordenada Y atingir ou ultrapassar o limite da matriz, o loop em questão será interrompido, indicando assim que o limite da direita foi atingido.

Da mesma forma seria quando nos referimos à próxima validação do trecho, onde conseguimos visualizar que se a direção for para a esquerda, a coordenada Y é decrementada para mover-se para esta direção. Caso ela seja 0, significa assim que o limite da esquerda foi alcançado, interrompendo o loop.

Caso seja para cima, a coordenada x será decrementada para mover-se de acordo, desde que o próximo caractere seja diferente de ‘/’ ou ‘\’. Caso a coordenada X for menor que 0, o limite superior da matriz foi alcançado e, assim, o loop é terminado.

Como última validação neste trecho, caso nenhuma das condições acima foram atendidas, a direção seria para baixo, onde a coordenada X é incrementada. Caso atinja ou ultrapasse o limite da matriz, o loop é interrompido, indicando assim que o limite inferior foi alcançado.

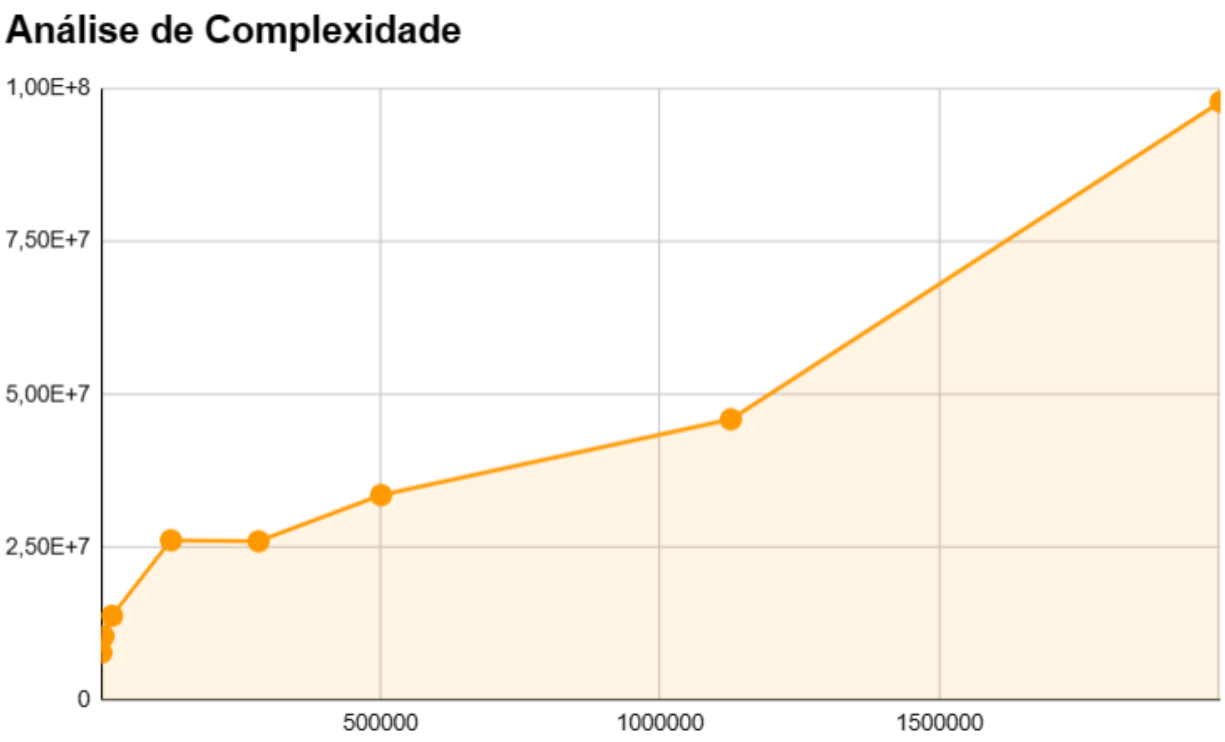
Com todo caminhamento realizado como esperado, temos agora apenas para finalizar um método a condicional que valida se o caractere atual é ‘#’, sendo o indicativo de que o caminho terminou. Assim, saímos do loop onde ocorriam todas as operações de caminhamento para retornarmos a soma do valor roubado pelos bandidos, mostrando assim a quantidade total, por fim, terminando nossa solução para o caso.

Resultados

Ao terminarmos a implementação do código como nos foi proposto, começamos assim a efetuar os testes com os casos de exemplo disponibilizados, sendo estes 8 casos de diferentes tamanhos de caminhos. Para termos insumos para análise, comparamos o tempo de execução com a quantidade de caracteres caminhados no mapa, no caso, com o tamanho do caminhamento de fato, onde obtivemos os seguintes dados:

Caso de Teste	Tempo de Execução (nanosegundos)	N° Caracteres	Valor roubado (R$)
G50	7.766.500	1.325	14.111,00
G100	10.462.700	5.169	25.411,00
G200	13.800.000	20.328	73.424,00
G500	26.137.500	125.795	871.897,00
G750	25.980.700	282.329	20.481.572,00
G1000	33.532.800	501.594	13.680.144,00
G1500	45.915.500	1.127.358	18.727.169,00
G2000	97.814.300	2.003.389	20.748.732,00

Para analisar a complexidade do algoritmo, utilizamos o Google Sheets para gerar um gráfico que representasse o comportamento. O eixo Y seria o tempo em nanosegundos e o eixo X seria o número de operações, no caso, os caracteres lidos. Após colocarmos os valores, obtivemos:



Observamos que em alguns momentos o gráfico apresenta "desníveis", como no caso G500, indicado pela quarta marcação no gráfico, em que o tempo de execução é maior em comparação aos outros testes. No entanto, esse comportamento pode ser atribuído ao fato de o mapa ter sido transformado em uma matriz. Em consequência disso, é possível que um caso com menos "espaço" no mapa resulte em um percurso maior na matriz. Por exemplo, a matriz pode ser totalmente preenchida com o mapa, resultando em um número igual ou próximo de caracteres caminhados de uma matriz com o triplo do tamanho, mas que não tem todo seu conteúdo preenchido.

Sendo assim, ao analisarmos o gráfico, observamos que, em grande parte do tempo, à medida que aumenta o conteúdo dos casos de teste, o gráfico assume uma postura linear. Isso sugere que a complexidade do algoritmo é, de fato, linear, sendo o *N* o número de caracteres caminhados por cada bandido. Essa linearidade torna-se mais evidente nos últimos três casos, G1000, G1500 e G2000, onde podemos observar claramente a tendência linear à medida que o tamanho da matriz aumenta.

Ou seja, após a elaboração e os testes com os resultados, essa análise confirma que o tempo de execução do algoritmo aumenta de maneira linear com o número de caracteres caminhados, indicando uma complexidade linear do algoritmo em relação ao tamanho do percurso.

Conclusões

O algoritmo apresentado ofereceu um desemprenho excelente e acima do esperado ao executar os diversos casos de mapas propostos, com tempos de execução baixos e o número de operações realizadas chegando a uma complexidade linear. Isso se deve ao algoritmo de caminhamento usar apenas algumas variáveis para manter controle do seu estado (*x*, *y*, *soma*, *isEnd*, *direction*) e nenhuma estrutura de dados que cresça de acordo com o tamanho de entrada. O pior caso do algoritmo apresentado é se haver a necessidade de visitar todas as células da matriz pelo menos uma vez, tornando um tempo de complexidade O(linhas*colunas).

Conclui-se, também, que os demais métodos apresentados, o transformar matriz e o de encontrar o hífen, expõem desempenhos igualmente satisfazíveis, com o de transformação tendo tempo de execução e complexidade de espaço ambos sendo O(número de linhas * tamanho máximo da linha), e o de encontrar hífen apresentando seu exemplo de pior caso como O(n), onde n é o número de linhas da matriz, que ainda assim possui complexidade de espaço constante.