Siga o Dinheiro - Algoritmos e Estruturas de Dados II

Ashiley Bianca Silva de Oliveira[*], Bernardo Santos Nilson[†]
Escola Politécnica — PUCRS

14 de abril de 2024

Resumo

O presente artigo detalha o desenvolvimento de uma alternativa para resolução do primeiro trabalho proposto na disciplina de Algoritmos e Estruturas de Dados II do terceiro semestre, intitulado "Siga o dinheiro". Este projeto desafia os alunos a conceber um algoritmo capaz de calcular a quantidade de dinheiro recolhido ao longo de uma trilha fictícia deixada por bandidos em fuga. Nesse trabalho será abordada a descrição inicial da proposta, os métodos algorítmicos e as estruturas de dados utilizados para resolver o problema, os resultados alcançados e a análise da complexidade da solução desenvolvida.

Introdução

O desafio proposto pelo trabalho I da disciplina de Algoritmos e Estrutura de Dados II envolve a resolução de um problema prático inspirado em uma situação fictícia de perseguição policial. A proposta é descrita da seguinte forma:

Considerando que um grande assalto a banco ocorreu, os criminosos deixaram uma trilha de notas de dinheiro enquanto tentavam fugir da polícia.

O problema a ser resolvido é auxiliar a equipe de perícia a calcular a quantidade total de dinheiro recolhido ao longo dessa trilha. Para a realização desse cálculo, é necessário levar alguns detalhes em consideração:

- Os mapas de teste fornecem representações de possíveis rotas de fuga dos criminosos, esses mapas estão em formato txt;
- O mapa deve ser percorrido imaginando que um carro está dirigindo nas estradas, coletando o dinheiro e contabilizando o total;
- Na primeira linha dos mapas é fornecida a informação sobre seu tamanho, sendo o tamanho definido pela quantidade de linhas e colunas do mapa;
- Símbolos como barras e contrabarras indicam mudanças de direção no mapa (para cima, para baixo, esquerda, direita);
- Números inseridos entre os itens representam a presença de dinheiro a ser recolhido;
- O "carro" deve andar sempre em linha reta, a não ser que encontre uma barra ou contrabarra;

[*]ashiley.bianca@edu.pucrs.br
[†]bernardo.nilson@edu.pucrs.br

- O caractere "#" (*hashtag* ou cerquilha) denota o fim do mapa, ou melhor, o local onde os criminosos foram capturados pela polícia.

Dessa forma, este artigo descreverá como realizou-se o planejamento e a construção de uma abordagem para a resolução do problema, descrevendo percepções específicas para a implementação prática de um algoritmo na linguagem *Python*. Ademais, haverá a apresentação dos resultados obtidos, a análise de complexidade da solução e as conclusões alcançadas através do trabalho.

Solução

Nesta seção, será realizada a descrição da solução para o trabalho proposto.

Antes de qualquer implementação algorítmica, fez-se necessária a leitura do arquivo txt, que obteve sua implementação através da seguinte lógica:

```
1 função ler_arquivo_de_texto (caminho_do_arquivo_txt)
2     abre o caminho do arquivo como arquivo
3         texto = todas as linhas do arquivo , em formato de lista
4
5     texto = limpar_barra_n_do_texto(texto)
6     retorna texto
7 fim
```

É importante salientar que foram implementados tratamentos de exceções, para que caso o usuário do programa tenha algum comportamento diferente do esperado, ele seja comunicado do erro.

Após a leitura do arquivo, definiu-se que a estrutura de dados a ser utilizada para a resolução do problema seria uma matriz. Tendo isso em vista, uma função para a transformação do arquivo txt em uma matriz foi implementada, como segue:

```
1 função construir_matriz (texto):
2         matriz = []
3         para cada linha em texto:
4             linha = []
5             para cada caractere em linha:
6                 adicione caractere àlinha
7             adicione linha àmatriz
8         retorna matriz
```

A função `construir_matriz()` é responsável por transformar um texto em uma matriz de caracteres. Começando com uma matriz vazia, ela itera sobre cada linha do texto, criando uma lista para armazenar os caracteres de cada linha. Em seguida, percorre os caracteres de cada linha, adicionando-os à lista. Ao final de cada linha, essa lista é adicionada à matriz. Assim, ao percorrer todas as linhas do texto, a função constrói uma matriz na qual cada linha corresponde a uma linha do texto e cada caractere é um elemento que estava contido nessa linha. Ao final, a função retorna a matriz resultante, para que assim, consigamos utilizá-la posteriormente nos caminhamentos.

Com intuito de compreender melhor o problema para seguir com a construção dos caminhamentos, o início da solução se fez pela exploração dos mapas de fuga. Inicialmente, o objetivo da análise era identificar um modo para encontrar o ponto inicial da trilha de fuga. Dessa forma, dada a visualização de cada um dos mapas de teste, identificou-se que:

- O início do caminho a ser percorrido sempre se apresentava do lado esquerdo do mapa, mais especificamente na primeira coluna (coluna 0) com um caractere hífen '-'.

Com base nisso, implementou-se uma função responsável por encontrar o índice de ínicio da trilha dos bandidos na matriz:

```
1 função encontrar_indice_de_comeco(matrix):
2     for indice_linha , linha in enumerate(matrix):
3         if '-' in linha:
4             return indice_linha
5     return -1
```

- O mapa deve ser percorrido em ordem, de forma que os caminhos definidos façam sentido;
- Haviam caracteres do tipo 'l' (barra vertical ou *pipe*) e '-' (hífen) que não necessariamente denotavam sentidos de direção, portanto, esse ponto era importante na implementação.

Ao observar a construção do mapa, identificou-se que a depender das direções e do caractere encontrado, o caminhamento seria realizado de diferentes formas. Como exemplo: caso o "carro" estivesse se movendo da esquerda para a direita e encontrasse o caractere "/" (barra), deveria mudar sua direção, movendo para cima.

Para uma melhor visualização e interpretação, realizou-se a organização de algumas informações úteis para a construção da solução. Tabulando as informações anteriormente mencionadas, temos:

Direção Atual	Caractere Encontrado	Direção Determinada
Direita para Esquerda	Barra	Vá para Baixo
Esquerda para Direita	Barra	Vá para Cima
Cima para Baixo	Barra	Vá para a Esquerda
Baixo para Cima	Barra	Vá para a Direita
Direita para Esquerda	Contrabarra	Vá para Cima
Esquerda para Direita	Contrabarra	Vá para Baixo
Cima para Baixo	Contrabarra	Vá para a Direita
Baixo para Cima	Contrabarra	Vá para a Esquerda

Interpretando essas informações, concluiu-se que as definições das direções determinadas implicavam na matriz subir uma posição na coluna, descer uma posição na coluna, avançar uma posição em uma linha ou regredir uma posição em uma linha. Retomando ao exemplo, caso o "carro" estivesse se movendo da esquerda para a direita e encontrasse o caractere "/" (barra), deveria mudar sua direção, movendo para cima, e ele realizaria essa movimentação na matriz permanecendo na coluna atual mas subindo uma linha, ou seja, linha-1.

De modo que houvesse a possibilidade de implementar essas interpretações no código, percorrendo a matriz, expressamos que:

Direção Determinada	Implica
Vá para Cima	linha - 1, coluna atual
Vá para Baixo	linha + 1, coluna atual
Vá para a Esquerda	linha atual, coluna - 1
Vá para a Direita	linha atual, coluna + 1

Tendo essa organização, definimos uma função de caminhamento, que recebe a linha atual, a coluna atual e direção desejada e realiza a definição de como os movimentos devem ser executados:

```
função caminha (linha , coluna , direção):
    se direção['cima'] então:
        retorna linha −1 , coluna
    senão se direção['baixo'] então:
        retorna linha + 1 , coluna
    senão se direção['direita'] então:
        retorna linha , coluna + 1
    senão:
        retorna linha , coluna −1
```

Sendo assim, iniciamos a implementação da função principal que é responsável por gerenciar os caminhamentos, coletar o dinheiro e somar o total coletado.

A função inicia a contagem de dinheiro roubado em zero, define a direção inicial como para a direita (pois a posição inicial sempre se inicia do lado esquerdo do mapa, na primeira coluna (coluna 0), com um caractere hífen ('-') e encontra a linha e a coluna da posição inicial da matriz. Então, ele caminha através da matriz, movendo-se na direção atual até encontrar um obstáculo "/" (barra) ou "\" (contrabarra). É importante salientar que se em meio a esse processo for encontrado um número, adiciona-se esse número à quantidade roubada. Em seguida, ele ajusta a direção dependendo do caractere encontrado ('/', '\') e continua o movimento. Quando encontra um caractere '#' (*hashtag* ou cerquilha), o loop termina e a quantidade total roubada é retornada, pois esse caractere determina o fim do caminho, ou seja, a posição onde os bandidos foram capturados pela polícia.

Resultados

Realizamos a execução do algoritmo com os casos disponibilizados, e obtivemos os seguintes resultados:

Caso	Quantidade de Dinheiro Roubado (em reais)
G50	14111
G100	25411
G200	73424
G500	871897
G750	20481572
G1000	13680144
G1500	18727167
G2000	20748732

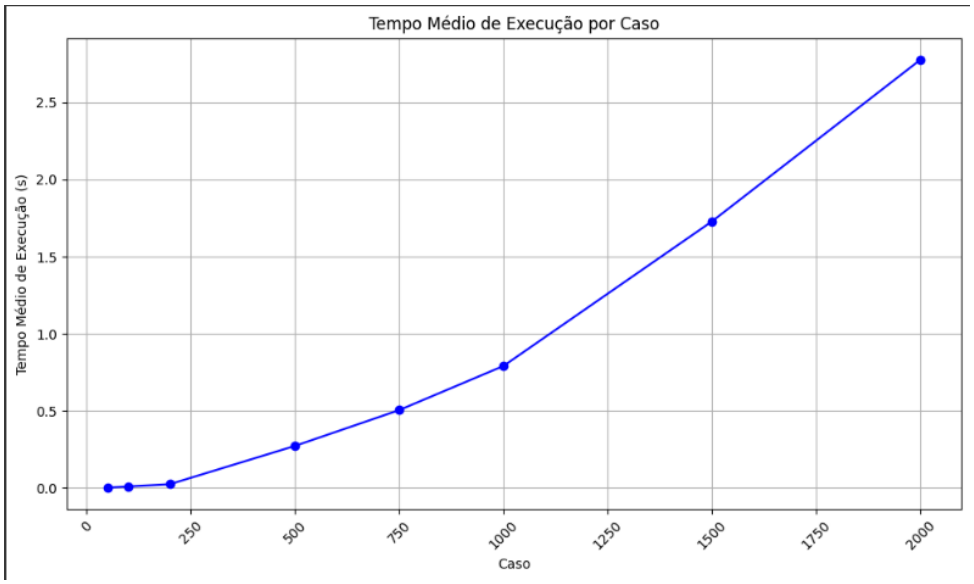
Esses resultados foram coerentes com o gabarito disponibilizado.

Depois de implementar o algoritmo em linguagem *Python*, realizamos a análise de complexidade utilizando a biblioteca *time*, atráves da função `time()` que retorna o número de segundos passados.

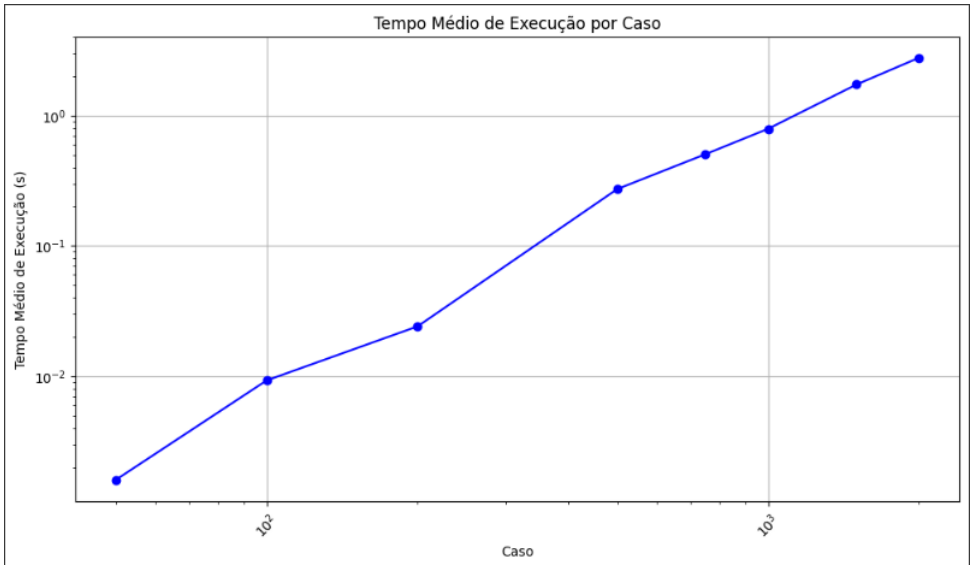
Cada um dos casos foi executado 5 vezes, os tempos de execução foram coletados e efetuou-se uma média simples. Como resultados, obtivemos:

Caso	Tempo Médio de Execução (em segundos)
G50	0.0015973091125488282
G100	0.009271192550659179
G200	0.024098587036132813
G500	0.27248268127441405
G750	0.5049667358398438
G1000	0.79122953414917
G1500	1.7292051315307617
G2000	2.778429460525513

Após obter os valores, foi utilizada a ferramenta *Matplotlib* no ambiente do *Google Colab* para gerar os gráficos usados na análise da notação O. A imagem gerada foi a seguinte:



Foram utilizadas técnicas de análise de complexidade, como a escala logarítmica nos eixos x e y, para verificar se o gráfico tem uma tendência polinomial. Se o gráfico, ao aplicar as escalas logarítmicas, for semelhante a uma reta, então a tendência é polinomial:



Para encontrar o valor aproximado do expoente b, utilizamos o seguinte cálculo:

$$b \approx \frac{\log(2.778429460525513) - \log(0.0015973091125488282)}{\log(2000) - \log(50)} \tag{1}$$

$$b \approx 2.02265234982 \tag{2}$$

Isso resulta que o algoritmo é de segunda ordem, ou seja, de complexidade quadrática.

Conclusões

O algoritmo possui complexidade quadrática, o que significa que seu tempo de execução aumenta quadraticamente com o tamanho da entrada. Apesar disso, para os tamanhos de entrada testados, o algoritmo demonstrou ser viável. No entanto, para mapas de tamanhos maiores, pode ser necessário otimizar o algoritmo para melhorar o desempenho. Para alcançarmos uma complexidade menor, ponderamos que possa ser necessária a troca da estrutura de dados utilizada (matriz) por outra que permita um caminhamento (ou pesquisa) mais rápida. Essa ideia traria ganhos de desempenho em troca de uma possível perda de legibilidade do código.

Ademais, o desenvolvimento desse projeto teve um papel importante para o aprendizado e consolidou importantes perspectivas, reunindo conteúdos apreendidos ao longo da disciplina de Algoritmos e Estruturas de Dados I e II e possibilitando que conseguimos compreender como medir as complexidades algorítmicas.