A Fantástica Máquina de Tubos e Bolinhas

Fernando Elger
Engenharia de Software — PUCRS

29 de setembro de 2019

Resumo

Este artigo descreve uma alternativa de solução para o segundo problema proposto na disciplina de Algoritmos e Estruturas de Dados II no semestre 3, que trata de um mecanismo de conexões entre tubos, no qual devemos descobrir o caminho mais suscetível partindo de vários pontos de início. É apresentado a lógica por trás da máquina e o funcionamento da solução que realiza o caminhamento entre as conexões. Em seguida é exposto os resultados para oito casos analisados e é avaliado a eficiência do algoritmo.

Introdução

Dentro do escopo da disciplina de Algoritmos e Estruturas de Dados II, o segundo problema pode ser resumido assim: a Fantástica Máquina de Tubos e Bolinhas do enigmático senhor Sibério é a grande atração deste ano na feira anual da vila de Fragoletto. A máquina é construída de longos tubos de metal por onde escorregam bolinhas e de vez em quando há um desvio para que a bolinha vá para outro tubo. Assim, a bolinha pode ir de um tubo para o outro várias vezes e sair em um tubo inesperado.

O senhor Sibério ganha dinheiro através de apostas com o público, onde as pessoas colocam uma bolinha em um tubo e depois tentam adivinhar por qual tubo ela sairá. Sibério reconecta sua máquina todas as noites alterando os desvios entre os tubos, para que cada novo dia traga novas oportunidades para ganhar dinheiro.

Sua namorada trabalha na barraca ao lado do senhor Sibério e ela descobre que pode enxergar pela janela o projeto com a programação da máquina, que consiste em uma longa lista de números (*Imagem 1*) que usaremos como exemplo.

Para interpretarmos a lista devemos seguir três princípios:

1. O primeiro número indica o número de tubos existentes;
2. O segundo número informa o comprimento dos tubos;
3. Em seguida, vem uma lista de grupos de 4 números para representar as conexões, que como demonstração usaremos: **a b c d**.

   Isso significa que o tubo **a** tem uma ligação no ponto **b** que leva até o tubo **c**, colocando a bolinha no ponto **d** daquele tubo.

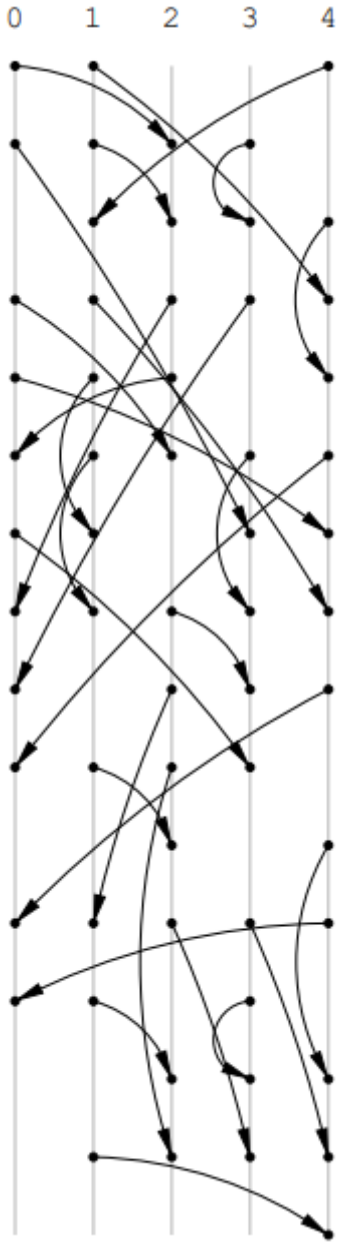
Você vê a lista e desenha os tubos com todas suas conexões (*Imagem 2*). Agora você quer ganhar dinheiro vendendo para os apostadores os seus conhecimentos sobre a máquina, mas constata que é complicado descobrir por onde a bolinha vai sair quando é colocada em um tubo qualquer.

Imagem 1

```
5  15
2  11  3  14
3  11  4  14
4  2  4  4
1  14  4  15
3  1  3  2
0  0  2  1
2  8  1  11
2  7  3  8
1  12  2  13
0  3  2  5
3  3  0  8
4  5  0  9
2  3  0  7
2  4  0  5
4  10  4  13
0  6  3  9
2  9  2  14
1  0  4  3
0  4  4  6
4  8  0  11
1  1  2  2
1  5  1  7
3  12  3  13
4  11  0  12
1  3  4  7
1  9  2  10
1  4  1  6
4  0  1  2
3  5  3  7
0  1  3  6
```

Representação da lista de números do exemplo

Imagem 2



Representação do desenho das conexões do exemplo

Conclui-se que o desenho é complicado de interpretar, logo você decide elaborar um programa para resolver o problema. O programa deve ser capaz de ler a longa lista de números que consistem no esquema da máquina e descobrir:

1. Qual o tubo por onde saem mais bolinhas? (considerando que jogaremos uma bolinha em cada tubo)
2. Quantas bolinhas saem por esse tubo?

No caso do exemplo acima, as informações encontradas devem ser sucessivamente:

1. Tubo 0;
2. 5 bolinhas.

Para resolver o desafio proposto, analisaremos como o problema foi modelado e em seguida, uma possível solução funcional, bem como suas características e dificuldades encontradas. Após isso, iremos comparar os resultados obtidos dos casos testados, além de elencar a complexidade do algoritmo e as conclusões obtidas no decorrer do trabalho.

Solução

Depois de considerar o desafio, podemos supor que a lista de números que organiza a estrutura da máquina poderia ser consumida e analisada ao mesmo tempo, porém como as conexões dos tubos estão em ordem aleatória na lista, esta solução é inviável já que não queremos ler o arquivo mais de uma vez. Com isso podemos concluir que devemos consumir a lista e armazenar as informações das conexões em alguma estrutura de dados. Uma opção seria usar grafos para criar os caminhos completos dos tubos a partir de cada conexão. Já outra possibilidade seria utilizar uma matriz, visto que temos o comprimento e a quantidade dos tubos, somente para guardar a referência dos desvios nos tubos.

Decidimos usar a matriz para a organização dos dados, mas como as referências das conexões dos tubos dependem de dois valores para serem representados, teremos que criar um objeto caminho, que consiste em um atributo linha e um atributo coluna, para então declarar uma matriz de caminhos.

Usamos o Scanner do Java para ler o arquivo .txt, usando os dois primeiros números para instanciar a matriz, sendo a quantidade de tubos as colunas e o comprimento dos tubos as linhas, levando em consideração que o comprimento começa em zero, logo devemos adicionar mais uma linha. Após isso, o Scanner lê os números de quatro em quatro, que constituem os dados de um único caminho, armazenando no índice das coordenadas onde começa a conexão a referência para onde ela vai.

Uma vez que temos constituídas na matriz todas as conexões dos tubos, o próximo passo é simular o jogo colocando uma bolinha em cada tubo, para isso iremos percorrer cada coluna da matriz de cima para baixo para ver em qual tubo paramos quando chegarmos no final do comprimento. Quando um índice de uma linha estiver nulo acessamos a linha de baixo pois não há nenhum desvio, já ao contrário, quando um índice não for nulo ele terá a referência para o final desvio para continuar o caminhamento. Foi criado um arranjo com o mesmo tamanho da quantidade de tubos para armazenar o incremento das bolinhas que saíram por cada tubo.

Este algoritmo que simula jogar uma bolinha em cada tubo pode funcionar da seguinte forma:

```
1       procedimento PLAYGAME()
2
3           linha ← 0
4           para i ← 0 até qtdColunas faça
5               coluna ← i
6               enquanto linha != qtdLinhas faça
7                   se matriz[linha][coluna] == null
8                       linha ← linha + 1
9                   se não
10                      linhaAux ← matriz[linha][coluna].linhaRef
11                      coluna ← matriz[linha][coluna].colunaRef
12                      linha ← linhaAux
13              fim
14              bolasTotal[coluna] ← bolasTotal[coluna] + 1
15              linha ← 0
16          fim
17      fim
```

Além do método de análise principal, foi desenvolvido também um simples menu para que o usuário escolha qual caso queira testar. Esse menu chama outro método chamado caseTest, (passando o nome do arquivo com o caso escolhido) que por sua vez, administra o teste, calculando o tempo de execução do método que consome o arquivo .txt para criar os caminhos e do outro método que simula o jogo.

Resultados

Depois de implementar o algoritmo acima em linguagem Java, os casos foram rodados cinco vezes para tirar a média do tempo de execução. Segue abaixo os resultados:

CASO	TAMANHO DO CASO (colunas * linhas)	TUBO QUE SAI MAIS BOLINHAS	QUANTIDADE DE BOLINHAS DO TUBO	TEMPO DE EXECUÇÃO (milissegundos)
caso1.txt	800	10	5	1.2
caso2.txt	5000	36	18	3.6
caso3.txt	50000	72	85	23.6
caso4.txt	200000	199	82	92.4
caso5.txt	1000000	180	139	212.8
caso6.txt	5000000	885	250	465.1
caso7.txt	20000000	680	546	2448.3
caso8.txt	100000000	2195	1565	14635.1

Conclusões

As primeiras indagações a respeito do desafio proposto, mesmo não oferecendo um resultado, contribuíram bastante para o entendimento do problema e abriu caminho para uma possível solução funcional. A partir disso, concluímos que geralmente levantar hipóteses para a solução de um problema e tentar achar equívocos nelas é muito mais inteligente do que simplesmente tentar resolver o problema na prática.

A forma apresentada de armazenar os desvios nos tubos se mostrou bastante simples, embora eu gostaria testar e discutir outras estruturas de dados para representar estes desvios, como por exemplo um dicionário. Contudo, a solução adotada possui uma notação O($n * m$), onde n é referente a quantidade de tubos e m ao comprimento dos tubos, pois no pior dos casos todos os tubos possuem desvios em todos os níveis dos seus comprimentos, logo o tempo de execução cresce proporcional a essas duas variáveis.