

Disciplina: Algoritmos e Estrutura de Dados II

Professor: Diego Vrague Noble

Aluno: Luís Gabriel Pacheco Marquetti

2019/2

O problema sendo resolvido é: dado um arquivo que descreve uma máquina de tubos e bolinhas, determine quantas bolinhas cairão em cada tubo após uma bolinha ser jogada no início de cada tubo.

Inicialmente eu tentei modelar um tubo com um array de links, que teria o mesmo tamanho do tubo. Essa approach não daria certo, pois não há um link em cada posição dos tubos, o que resultaria em muitas posições vazias no array, e em muita memória reservada e não usada. Eu acabei desenvolvendo dois algoritmos para resolver o problema, cada um modelando os tubos e links de maneiras diferentes. No primeiro algoritmo eu criei uma classe Link, que modelava um link entre dois tubos, e cada link continha os 4 números presentes nos arquivos. A classe Tube, que modela um tubo, contem um ArrayList de Links, ordenado de acordo com a posição inicial no tubo em que o link começa. E para juntar os tubos eu criei uma classe Machine, que modela a máquina inteira, e que possui um array de tubos para modelar o conjunto de tubos.

No segundo algoritmo eu criei apenas uma classe Machine, que modela a maquina inteira. Essa classe possui um array de filas de prioridade de links, onde cada fila de prioridade modela um tubo.

Eu não consegui criar um algoritmo que removesse links redundantes, pois acabei gastando muito tempo para resolver o problema mais básico T\_T

No primeiro algoritmo eu usei uma espécie de recursão, em que o método goThroughLink chama ele mesmo, porém em uma outra instância de Tube. Cada chamada desse método percorre um link, saindo de um tubo e chegando em outro. O método nextLink é chamado no tubo em que o link chega, e recebe a posição de chegada e remove todos os links do tubo de chegada que foram ‘pulados’, e que não seriam utilizados. Como a lista de links está ordenada, o próximo link a ser percorrido no tubo de chegada será o primeiro da lista. Esse método auxiliar também serve para determinar se a bolinha caiu no tubo onde o método foi chamado.

public void dropIn(){

        if(links.size()>0)

            goThroughLink(0);

    }

    public void goThroughLink(int linkPos){

        Link link = links.remove(linkPos);

        Tube destTube = link.getDestination();

        int nextLinkPos = destTube.nextLink(link.getEndPoint());

        if(destTube.finishedHere())

            return;

        destTube.goThroughLink(nextLinkPos);

    }

private int nextLink(int posChegada){

        if(links.isEmpty()){

            finishedHere = true;

            return -1;

        }

        while(links.get(0).getStartingPoint() <= posChegada){

            links.remove(0);

            if(links.isEmpty()){

                this.finishedHere = true;

                return -1;

            }

        }

        return 0;

    }

Eu demorei mais tempo do que eu gostaria de admitir para perceber que tinha que, a cada vez que uma bolinha era jogada em um tubo, reiniciar as estruturas de dados modelando os tubos e links para o seu estado inicial, já que eu removia elementos a cada jogada de bolinha. Durante esse tempo, eu me frustrei bastante , pois ao jogar apenas uma bolinha em apenas um cano a resposta estava correta, mas ao tentar jogar uma em cada cano em uma mesma execução do programa a resposta estava completamente errada.

No desenvolvimento do segundo algoritmo, o que usa um array de filas de prioridade, eu aprendi com os erros do desenvolvimento do primeiro e consegui acabá-lo muito mais rapidamente. A ideia básica do algoritmo é a mesma, ir de um tubo ao outro percorrendo os links, e removendo os links que foram ‘pulados’, e assim que forem removidos todos os links de um cano, significa que a bolinha caiu nele. O método skippedLinksRemover é análogo ao método nextLink do primeiro algoritmo.

//metodo principal que implementa o algoritmo para percorrer os tubos e

links

    public void dropBallInTube(int tube){

       PriorityQueue<Link> tuboSaida = machine[tube];

       int posChegada;

       int tuboDestino;

       Link link;

       while(tuboSaida.size()>0){

           link = tuboSaida.poll();

           tuboDestino = link.getDestination();

           posChegada = link.getEndPoint();

           if(skippedLinksRemover(tuboDestino, posChegada)){

               count[tuboDestino]++;

               break;

           }

           tuboSaida = machine[tuboDestino];

       }

    }

    //quando uma bolinha é direcionada a um tubo, esse metodo remove todos os links que foram 'pulados' e retorna true quando a fila termina, ou seja

    //quando o tubo em questão é o tubo no qual a bolinha caiu

    private boolean skippedLinksRemover(int tuboChegada, int posChegada){

        PriorityQueue<Link> tube = machine[tuboChegada];

        if(tube.isEmpty())

            return true;

        while(tube.peek().getStartingPoint() <= posChegada){

            tube.poll();

            if(tube.isEmpty()){

                return true;

            }

        }

        return false;

    }

Resultados:

Caso 1: Algoritmo 1: usando ArrayLists ordenados

tempo para ler arquivo e criar tubos: 0.0037302

tempo para ordenar: 2.22E-4

0->2

2->1

5->3

7->5

10->3

13->3

14->1

15->1

18->1

Tempo de execucao: 0.0068669

Maximo: cano 7, com 5 bolinhas

Algoritmo 2: usando um array de filas de prioridade

tempo de construcao dos tubos: 0.0280473s

Tempo de execucao: 5.24E-4s

0->2

2->1

5->3

7->5

10->3

13->3

14->1

15->1

18->1

Maximo: cano 7, com 5 bolinhas

Caso 2: alg 1

tempo para ler arquivo e criar tubos: 0.0080917

tempo para ordenar: 4.563E-4

4->10

8->3

9->13

10->4

14->2

17->3

20->6

35->4

38->5

Tempo de execucao: 0.0083432

Maximo: cano 9, com 13 bolinhas

Alg 2:

tempo de construcao dos tubos: 0.0317881s

Tempo de execucao: 0.0012946s

4->10

8->3

9->13

10->4

14->2

17->3

20->6

35->4

38->5

Maximo: cano 9, com 13 bolinhas

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Caso 3: alg 1

tempo para ler arquivo e criar tubos: 0.036286

tempo para ordenar: 0.0047472

52->3

55->26

86->36

99->35

Tempo de execucao: 0.0119343

Maximo: cano 86, com 36 bolinhas

Alg 2

tempo de construcao dos tubos: 0.065106s

Tempo de execucao: 0.0200674s

52->3

55->26

86->36

99->35

Maximo: cano 86, com 36 bolinhas

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Caso 4: alg 1

tempo para ler arquivo e criar tubos: 0.0986331

tempo para ordenar: 0.0096949

66->26

110->34

149->86

169->54

Tempo de execucao: 0.0590473

Maximo: cano 149, com 86 bolinhas

Alg 2:

tempo de construcao dos tubos: 0.1305716s

Tempo de execucao: 0.1190743s

66->26

110->34

149->86

169->54

Maximo: cano 149, com 86 bolinhas

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Caso 5: alg 1

tempo para ler arquivo e criar tubos: 0.224202

tempo para ordenar: 0.0146612

97->26

115->25

118->44

211->192

242->178

404->3

461->32

Tempo de execucao: 0.1874228

Maximo: cano 211, com 192 bolinhas

Alg 2:

tempo de construcao dos tubos: 0.2462582s

Tempo de execucao: 0.3770613s

97->26

115->25

118->44

211->192

242->178

404->3

461->32

Maximo: cano 211, com 192 bolinhas

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Caso 6: alg 1

tempo para ler arquivo e criar tubos: 0.3650304

tempo para ordenar: 0.0305838

37->264

45->75

130->365

149->8

185->53

220->26

581->54

726->155

Tempo de execucao: 0.5785067

Maximo: cano 130, com 365 bolinhas

Alg 2:

tempo de construcao dos tubos: 0.4503436s

Tempo de execucao: 1.1233175s

37->264

45->75

130->365

149->8

185->53

220->26

581->54

726->155

Maximo: cano 130, com 365 bolinhas

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Caso 7: alg 1

tempo para ler arquivo e criar tubos: 1.7873126

tempo para ordenar: 0.0948135

811->710

877->107

1334->684

1390->268

1659->231

Tempo de execucao: 12.9499345

Maximo: cano 811, com 710 bolinhas

Alg 2:

tempo de construcao dos tubos: 2.0909397s

Tempo de execucao: 17.6917682s

811->710

877->107

1334->684

1390->268

1659->231

Maximo: cano 811, com 710 bolinhas

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Caso 8:

Algoritmo 1: usando ArrayLists ordenados

tempo para ler arquivo e criar tubos: 9.2925613s

41->241

49->421

68->674

86->880

596->357

1416->344

1514->79

1615->846

2282->80

2987->161

4754->917

Tempo de execução: 224.6374641s

Maximo: cano 4754, com 917 bolinhas

Algoritmo 2: usando um array de filas de prioridade

tempo de construcao dos tubos: 9.0674758s

Tempo de execucao: 188.6351832s

41->241

49->421

68->674

86->880

596->357

1416->344

1514->79

1615->846

2282->80

2987->161

4754->917

Maximo: cano 4754, com 917 bolinhas

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Conclusões

Os resultados experimentais mostram que o alg 2 foi no geral mais rapido, apesar de ambos terem tempos de execução comparáveis. Acredito que para se ter uma diminuição considerável no tempo de execução eu deveria ter removido os links redundantes antes de rodar os algoritmos. Apesar de eu ter investido bastante tempo nesse trabalho, a maioria do tempo foi tentando resolver problemas no meu algoritmo, e aprendendo a usar o debugger do java, uma ferramenta útil que me ajudará bastante no futuro.