UNIVERSIDADE DE ITAÚNA

DAVI VENTURA CARDOSO PERDIGÃO
EDMILSON LINO CORDEIRO
ERIC HENRIQUE DE CASTRO CHAVES

APLICAÇÃO BACKTRACKING - TENTATIVA E ERRO

Questão 3 - Prova Final (Projeto e Análise de Algoritmo)

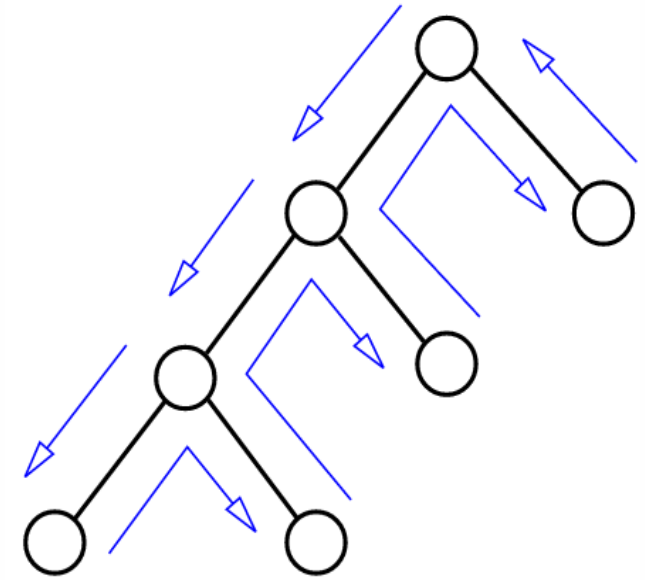
ITAÚNA
2022

SUMÁRIO

1. INTRODUÇÃO

Backtracking é um algoritmo genérico que busca, por força bruta, soluções possíveis para problemas computacionais, em que múltiplas soluções podem ser eliminadas sem serem explicitamente examinadas. O termo foi criado pelo matemático estadunidense D. H. Lehmer na década de 1950.

Uma busca inicial em um programa nessa linguagem segue o padrão busca em profundidade, ou seja, a árvore é percorrida sistematicamente de cima para baixo e da esquerda para direita. Quando essa pesquisa falha, ou é encontrado um nodo terminal da árvore, entra em funcionamento o mecanismo de ***backtracking***. Esse procedimento faz com que o sistema retorne pelo mesmo caminho percorrido com a finalidade de encontrar soluções alternativas.



Fonte: Fórum StackOverflow. (2015)

2. CARACTERÍSTICAS DO BACKTRACKING

Características Básicas:

- Para cada chamada recursiva existem diversas opções que podem ser seguidas. Ex: Muitos vértices podem ser o próximo;
- Não seguem regra fixa de computação:
- Constrói a solução, um componente por vez, tentando terminar o processo tão logo quanto for possível identificar que uma solução não poderá ser feita obtida em razão das escolhas feitas.
- O algoritmo pode escolher um ou poucos dados segundo um critério qualquer.

- O processo de busca cria uma árvore de chamadas recursivas. Ex: Todos os caminhos parciais do caixeiro viajante.
- Folhas dessa árvore são de dois tipos:
    1. Representam uma possível solução para o problema.
    2. Representam um ponto onde o algoritmo não pôde mais ir adiante (failure) sem ferir alguma pré-condição para que a solução gerada até então seja válida.

Pontos Positivos:

- Forma bastante fácil de implementar um problema que de outra forma seria muito mais complexo de se resolver.
- Esta técnica torna possível a resolução de muitos problemas NP-difícil com instâncias grandes em um tempo aceitável.

Pontos Negativos:

- Programas de backtracking, a não ser que se programe restrições (constraints) executam sempre uma busca exaustiva e tenderão à explosão combinatória.
- A quantidade de variáveis locais transportada por cada chamada recursiva é diretamente proporcional ao tamanho do problema. Isto significa que a quantidade de memória requerida para um programa de backtracking pode crescer exponencialmente com o tamanho do problema.

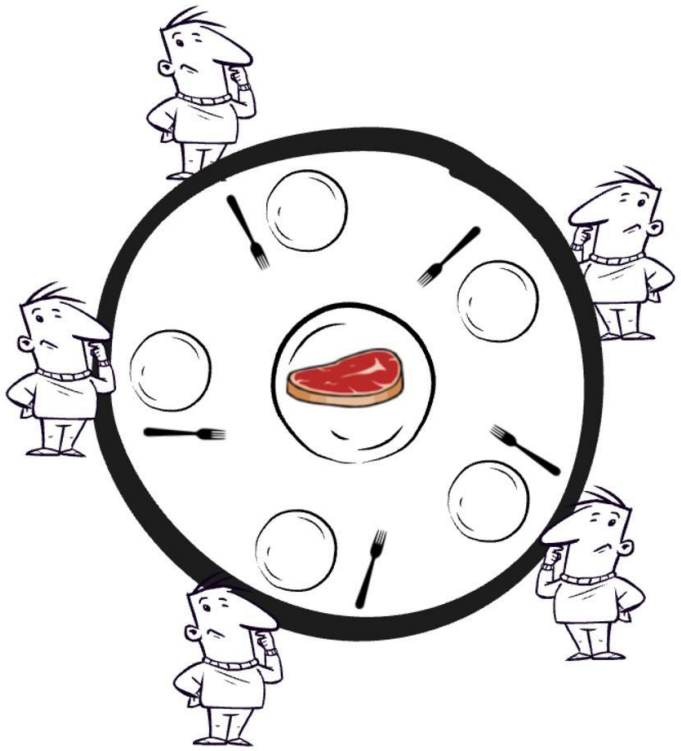
3. O PROBLEMA - JANTAR DOS FILÓSOFOS

O ***Problema dos Filósofos*** foi proposto pelo cientista da computação Edsger W. Dijkstra em 1965, sendo classificado como um problema de sincronização.

Este problema consiste em 5 filósofos que passam a vida pensando e comendo, eles partilham de uma mesa redonda com 5 lugares pertencendo a cada um deles No centro da mesa encontra-se uma bandeja contendo um alimento e 5 garfos na mesa, sendo um para cada filósofo.

Quando um filósofo pensa, ele não interage com os outros. De tempos em tempos, cada filósofo fica com fome e tenta apanhar os dois garfos que estão mais próximos (os garfos que estão ou à esquerda ou à direita). O filósofo apenas pode apanhar um garfo de cada vez, além de não pode apanhar um garfo se este estiver na mão do vizinho. Quando um filósofo esfomeado tem 2 garfos ao mesmo tempo ele come sem largar os garfos. Apenas quando acaba de comer, o filósofo pousa os garfos na mesa, libertando-os e começa a pensar de novo.

➔ 5 Filósofos
➔ 5 Garfos
➔ Come quando tiver 2 garfos
➔ Após comer devolve os garfos



Fonte: CHAGAS, Gustavo. (2021)

4. O ALGORÍTMO E SUA COMPLEXIDADE

O conceito de processamento concorrente de tarefas integra a linguagem Java desde sua criação, através de multithreading. Os threads em Java são gerenciados pela JVM (Máquina Virtual Java), e só suporta um processo, sendo necessário um novo JVM para cada processo novo. Para o desenvolvedor, o programa começa um thread com a chamada de main thread, e este é responsável por criar novas threads. Na JVM, os threads são escalonados de forma preemptiva seguindo o algoritmo escalonador “round-robin”. Isso quer dizer que o escalonador pode pará-las, dar espaço e tempo para outra thread ser executada. Tratando-se da complexidade, teremos um **“for”** na classe principal que iniciará um thread onde há um **“While”** na classe Filosofo.java, com isso podemos concluir que sua complexidade é **N^2.**

- Filósofo é um thread filha da classe Thread
- Semáforo representa o garfo
- Wait e Signal contra o deadlock

JantarDosFilosofos.java -

```
import java.util.concurrent.Semaphore;

public class JantarDosFilosofos {
    private static final int N = 5;

    public static void main(String[] args) {
        // Cada garfo será um semaforo
        Semaphore[] garfo = new Semaphore[N];

        for (int i = 0; i < N; i++) {
            garfo[i] = new Semaphore(1);
        }

        // Cria os filosofos e inicia cada um executando a sua thread
        Filosofo[] filosofos = new Filosofo[5];

        for (int i = 0; i < N; i++) {
            filosofos[i] = new Filosofo(i, garfo[i], garfo[(i + 1) %
N]);
            new Thread(filosofos[i]).start();
        }
    }
}
```

Filosofo.java -

```
import java.util.Random;
import java.util.concurrent.Semaphore;

public class Filosofo implements Runnable {
    // Usado para definir por quanto tempo um filosofo pensa antes de
comer e quanto
    // tempo ele fica comendo.
    private Random number = new Random();
    private int qtdComida = 10;

    // Utilizado para finalizar a execução do thread
    private boolean threadEstado;

    private final int id;
    private final Semaphore garfo_esquerdo;
```

```
    private final Semaphore garfo_direito;

    public Filosofo(int id, Semaphore garfo_esquerdo, Semaphore
garfo_direito) {
        this.id = id;
        this.garfo_esquerdo = garfo_esquerdo;
        this.garfo_direito = garfo_direito;
        this.threadEstado = true;
    }

    // Ciclo infinito de cada filosofo, executado em threads separados
    @Override
    public void run() {
        try {
            while (threadEstado) {
                pensar();
                pegarGarfo_esquerdo();
                pegarGarfo_direito();
                comer();
                devolverGarfo();
            }
        } catch (InterruptedException e) {
            System.out.println("Filosofo " + id + "foi
interrempido.\n");
        }
    }

    // Modelo de pensamento. Define um tempo aleatorio para o filosofo
pensar
    private void pensar() throws InterruptedException {
        System.out.println("Filosofo " + id + " está pensando\n");
        // Tempo de execução
        Thread.sleep(number.nextInt(10));
    }

    // Analisa a quantidade de permissões para poder pegar o garfo
esquerdo
    private void pegarGarfo_esquerdo() throws InterruptedException {
        if (garfo_esquerdo.availablePermits() == 0) {
            System.out.println("Filosofo " + id + " está ESPERANDO o
garfo esquerdo.\n");
        }
```

```
        garfo_esquerdo.acquire();
        System.out.println("Filosofo " + id + " está SEGURANDO o garfo
esquerdo\n");
    }

    // Analisa a quantidade de permissões para poder pegar o garfo
direito
    private void pegarGarfo_direito() throws InterruptedException {
        if (garfo_direito.availablePermits() == 0) {
            System.out.println("Filosofo " + id + " está ESPERANDO o
garfo direito.\n");
        }

        garfo_direito.acquire();
        System.out.println("Filosofo " + id + " está SEGURANDO o garfo
direito\n");
    }

    // Define um tempo aleatorio para o filosofo comer
    private void comer() throws InterruptedException {
        if (this.qtdComida > 0) {
            System.out.println("Filoso " + id + " está COMENDO");
            this.qtdComida--;
            Thread.sleep(number.nextInt(10));
        } else {
            System.out.println("A comida acabou\n");
            threadEstado = false;
            System.exit(0);
        }

    }

    // Liberar os garfos para outro filosofo poder pegar
    private void devolverGarfo() {
        garfo_esquerdo.release();
        garfo_direito.release();
        System.out.println("Filosofo " + id + " SOLTOU os garfos\n");
    }
}
```

Exemplo de Saída (terminal) -

```
Filosofo 0 está pensando
Filosofo 4 está pensando
Filosofo 1 está pensando
Filosofo 3 está pensando
Filosofo 2 está pensando
Filosofo 0 está SEGURANDO o garfo esquerdo
Filosofo 3 está SEGURANDO o garfo esquerdo
Filosofo 4 está SEGURANDO o garfo esquerdo
Filosofo 4 está ESPERANDO o garfo direito.
Filosofo 0 está SEGURANDO o garfo direito
Filoso 0 está COMENDO
Filosofo 3 está ESPERANDO o garfo direito.
Filosofo 2 está SEGURANDO o garfo esquerdo
Filosofo 2 está ESPERANDO o garfo direito.
Filosofo 1 está ESPERANDO o garfo esquerdo
Filosofo 0 SOLTOU os garfos
Filosofo 4 está SEGURANDO o garfo direito
Filosofo 1 está SEGURANDO o garfo esquerdo
Filoso 4 está COMENDO
Filosofo 0 está pensando
Filosofo 4 SOLTOU os garfos
Filosofo 4 está pensando
Filosofo 3 está SEGURANDO o garfo direito
```

...

...

```
Filosofo 2 está SEGURANDO o garfo esquerdo
Filosofo 2 está ESPERANDO o garfo direito.
Filosofo 0 está SEGURANDO o garfo direito
Filoso 0 está COMENDO
Filosofo 1 está ESPERANDO o garfo esquerdo.
Filosofo 4 está SEGURANDO o garfo direito
Filoso 4 está COMENDO
Filosofo 0 SOLTOU os garfos
Filosofo 0 está pensando
Filosofo 1 está SEGURANDO o garfo esquerdo
Filosofo 1 está ESPERANDO o garfo direito.
Filosofo 4 SOLTOU os garfos
Filosofo 3 está SEGURANDO o garfo direito
Filosofo 4 está pensando
Filoso 3 está COMENDO
Filosofo 4 está ESPERANDO o garfo esquerdo.
Filosofo 2 está SEGURANDO o garfo direito
Filoso 2 está COMENDO
Filosofo 3 SOLTOU os garfos
Filosofo 3 está pensando
Filosofo 4 está SEGURANDO o garfo esquerdo
Filosofo 4 está SEGURANDO o garfo direito
A comida acabou
PS D:\Backup HD\Repositórios Git + VS\Jantar-dos-Filosofos>
```

FIM.

5. CONCLUSÃO

Como o nome diz, *retornar pelo caminho*, os algoritmos de **backtracking** constroem sempre o conjunto de solução ao retornarem das chamadas recursivas. Portanto, esses algoritmos asseguram o acerto por enumerar todas as possibilidades sem visitar nunca o mesmo estado, sendo também *eficiente*. A recursividade promove a elegância e a fácil implementação desse algoritmo, porque o vetor de novos candidatos é alocado com um procedimento recursivo.

6. REFERÊNCIAS BIBLIOGRÁFICAS

- AUTOR DESCONHECIDO. **Problemas - Jantar dos Filósofos**. Disponível em: <https://sites.google.com/site/tecprojalgoritmos/problemas/jantar-dos-filosofos>. Acesso em: 28 de junho de 2022.

- FIGUEIREDO, Jorge. **Análise e Técnicas de Algoritmos - Backtracking and Branch-and-Bound Backtracking and Branch-and-Bound**. Disponível em: <https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnx0ZWNwcm9qYWxnb3JpdG1vc3xneDo0OWIxYjdhMWRkMmMyOWM5>. Acesso em: 26 de junho de 2022.
- FIGUEIREDO, Jorge. **Análise e Técnicas de Algoritmos - Backtracking**. Disponível em: <https://sites.google.com/site/tecprojalgoritmos/tecnicas-de-projetos/backtracking>. Acesso em: 26 de junho de 2022.
- UFSC. **Backtracking**. Disponível em: <https://www.inf.ufsc.br/~aldo.vw/Analise/Backtracking.html>. Acesso em: 26 de junho de 2022.
- WIKIDOT. **Jantar dos Filósofos**. Disponível em: <http://ces33.wikidot.com/tiagoporto:jantar-dos-filosofos>. Acesso em: 28 de junho de 2022.