

# MAC0438 - Programação Concorrente - 1s2012

## Relatório

Thiago de Gouveia Nunes  
Wilson Kazuo Mizutani

May 13, 2012

### Contents

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Solução desenvolvida</b>	<b>2</b>
2.1	Ideia geral . . . . .	2
2.2	Estendendo a busca em largura . . . . .	2
2.3	Aplicando concorrência . . . . .	3
<b>3</b>	<b>Implementação da solução</b>	<b>3</b>
3.1	Breve descrição das classes . . . . .	3
3.2	Alguns detalhes de implementação . . . . .	3
3.3	Sobre a barreira simétrica usada . . . . .	3
<b>4</b>	<b>Resultados</b>	<b>3</b>
4.1	Resultado para o grafo da NSFNet . . . . .	3
4.2	Comparações de eficiência . . . . .	3
<b>5</b>	<b>Conclusões</b>	<b>3</b>

# 1 Introdução

Esse relatório trata das decisões tomadas na implementação desse EP. Também fornece uma saída do programa para a entrada de exemplo fornecida no enunciado, além de explicitar a localização no código da implementação da barreira simétrica usada. Informações sobre como compilar, as dependências necessárias e o modo de uso do programa encontram-se no arquivo LEIAME.

## 2 Solução desenvolvida

### 2.1 Ideia geral

A ideia geral da nossa solução se divide em duas partes:

1. Estender o algoritmo de busca em largura (Breadth-First Search, um caso particular do algoritmo de Dijkstra no qual todas as arestas possuem custo 1) para que ele encontre não só o menor caminho, mas sim os  $n$  menores caminhos.
2. Refatorar esse algoritmo para usar programação concorrente, de tal maneira que cada thread seja responsável por tentar encontrar um novo caminho e depois sincronizar com as demais, criando assim um processo iterativo.

### 2.2 Estendendo a busca em largura

Basicamente, aproveitamos a propriedade da busca em largura na qual um novo vértice retirado da fila está sendo visitado pela primeira vez através do menor caminho.

Estendemos o algoritmo para ter uma fila de caminhos ao invés de vértices, por conveniência, e ao invés de deixarmos de visitar um vértice após passar por ele apenas uma vez, o fazemos após  $n$  vezes (o que significa que ele já tem  $n$  caminhos mínimos terminando nele).

Assim, a propriedade do nosso algoritmo (ainda no caso não-concorrente) seria que um novo caminho retirado da fila é o próximo menor caminho que termina no mesmo último vértice que ele. Mas como o enunciado do EP pedia que cada thread cuidasse de apenas um caminho, mudamos isso de forma que, na verdade, insere-se apenas candidatos a caminho na fila. A propriedade fica portanto que um novo candidato retirado da fila *pode* ser o próximo menor caminho que termina no seu último vértice, *contanto que ele não seja um ciclo*.

Uma consequência disso é que o programa vai com certeza ficar menos eficiente, pois há menos restrições sobre quem entra na fila, e portanto ela potencialmente terá mais elementos do que na maneira anterior.

## **2.3 Aplicando concorrência**

Pensamos em várias maneiras de fazer isso. O grande problema é que a ordem com que os caminhos são encontrados é indeterminada, então não podemos assumir que os  $n$  primeiros encontrados sejam, de fato, os  $n$  menores. Vimos duas maneiras de lidar com isso.

Uma seria controlar a inserção em uma fila compartilhada através de um buffer intermediário que garantisse sua ordem. Isso apresenta dois problemas: a necessidade de uma fila compartilhada, e o fato de que a inserção nela não seria verdadeiramente paralelizada, pois seria justamente preciso serializar as threads para que elas inserissem na ordem correta.

A outra maneira é deixar o trabalho todo para a lista de caminhos de cada vértice. Elas mesmas cuidariam de guardar o  $n$  menores caminhos, descartando os demais. Assim, cada thread poderia ter sua própria fila, e no total o algoritmo seria bem mais paralelizável. O problema seria que a inserção de um dos  $n$  verdadeiramente menores caminhos custaria com a melhor implementação que pudemos pensar  $O(\log(n))$  ao invés de apenas  $O(1)$ .

No final, optamos pela segunda opção, apostando que uma maior paralelização do algoritmo compensasse a inserção de novos caminhos mínimos.

## **3 Implementação da solução**

### **3.1 Breve descrição das classes**

### **3.2 Alguns detalhes de implementação**

### **3.3 Sobre a barreira simétrica usada**

## **4 Resultados**

### **4.1 Resultado para o grafo da NSFNet**

### **4.2 Comparações de eficiência**

## **5 Conclusões**