Técnicas de programação concorrente aplicadas ao algotitmo de Best Matching

Gilney de Azevedo Alves Junior

5 de abril de 2019

1 Introdução

Suponha um conjunto de palavras D ao qual chamaremos de dicionário e uma palavra de entrada p. O algoritmo de Best Matching consiste em calcular a palavra (ou a lista de palavras) mais semelhante a p dentro do dicionário D. Para isso, utilizaremos a distância de Levenshtein como medida de similaridade entre duas palavras.

A distância de Levenshtein mede a quantidade de alterações (adição, remoção ou substituição de um caractere) necessárias para transformar uma palavra em outra. Esse cálculo possui a seguinte fórmula recursiva:

$$\text{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{, se } \min(i,j) = 0 \\ \min \begin{cases} \text{lev}_{a,b}(i-1,j) + 1 \\ \text{lev}_{a,b}(i,j-1) + 1 \\ \text{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_i)} \end{cases} & \text{, caso contrário} \end{cases}$$

, onde a e b são duas palavras, i e j são os tamanhos de a e b respectivamente e $1_{(a_i \neq b_i)}$ é uma condição que soma 1 à fórmula quando é verdadeira e soma 0 quando é falsa.

Tendo feito o cálculo da distância de Levenshtein de p para cada uma das palavras em D, basta tomar a palavra (ou lista de palavras) com menor distância para se obter o resultado das mais similares.

2 Algoritmos

O algoritmo usado neste experimento, em vez de calcular as palavras de um dicionário mais semelhantes a uma única palavra de entrada, calculará para uma lista de palavras de entrada. As duas subseções seguintes mostrarão as abordagens usadas para um algoritmo sequencial e um algoritmo concorrente. As implementações dos algoritmos utilizados neste experimento estão disponibilizados em `https://github.com/gilneyjr/BestMatching`.

2.1 Algoritmo Sequencial

Como mostrado na Figura 1, o algoritmo sequencial primeiramente lê as palavras do dicionário D, depois lê as palavras de entrada P. Em Seguida, para cada palavra $p_i \in P$, calcula a distância de Levenshtein para cada $d_j \in D$ e salva o par $(\text{lev}_{p_i,d_j}(|p_i|, |d_j|), d_j)$ em uma lista R, e, por fim, ordena e salva R em um arquivo.

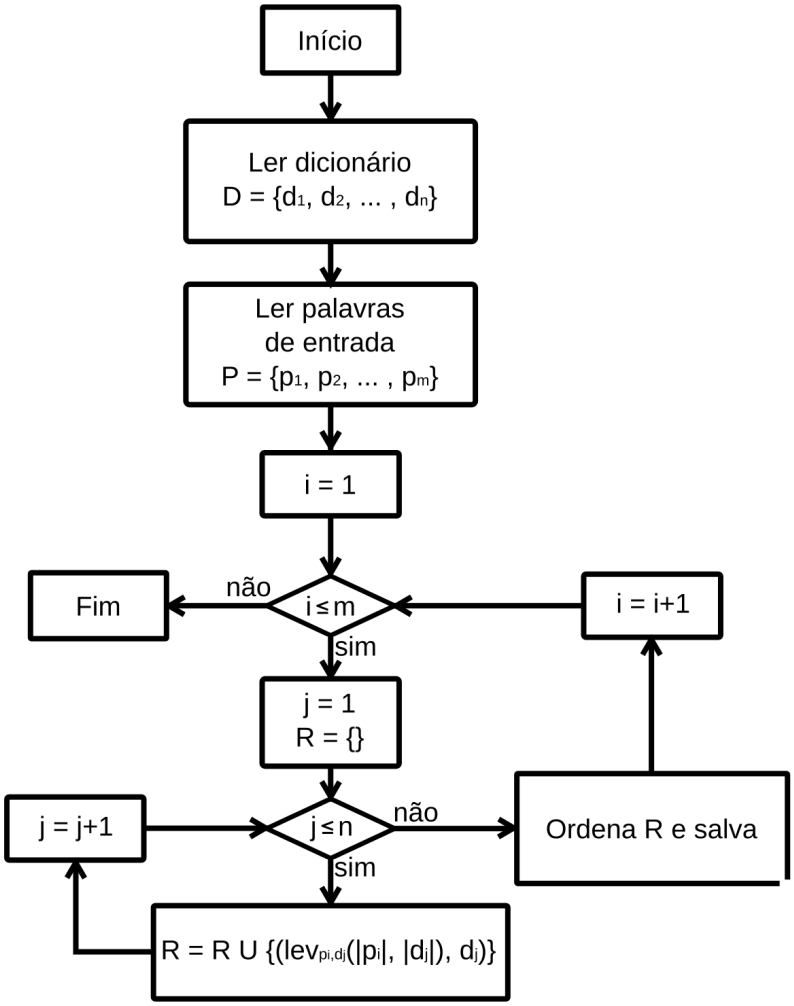


Figura 1: Fluxograma do algoritmo sequencial.

2.2 Algoritmo Concorrente

Como podemos ver nas Figuras 2, 3, 4 , o algoritmo concorrente segue a mesma ideia do algoritmo sequencial, porém divide algumas tarefas entre threads.

A Figura 2 mostra o fluxo principal do programa. A primeira divisão de tarefas ocorre na leitura, onde as threads lêem as palavras do dicionário e as palavras de entrada concorrentemente. Após isso, ocorre um join para sincronizar as threads e há uma nova divisão em várias threads T1. E, no final, ocorre outro join para finalizar a execução. Na implementação desse algoritmo, foram utilizadas 4 threads T1, e para cada thread T1, 4 threads T2 foram instanciadas.

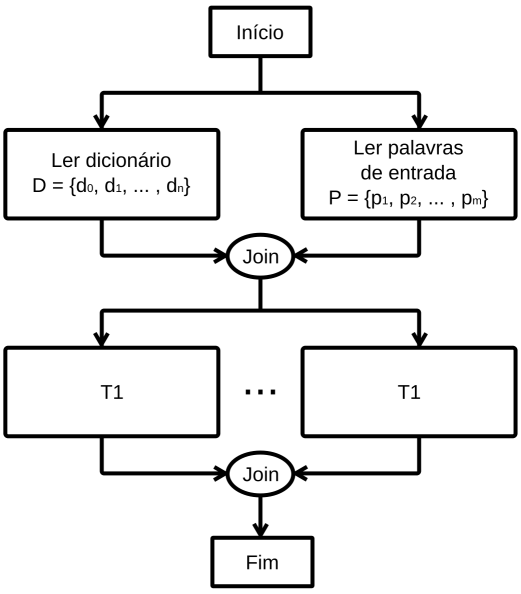


Figura 2: Fluxograma do algoritmo concorrente.

A Figura 3 mostra o fluxo de execução da thread T1. Primeiramente, cria-se uma lista R que vai guardar o resultado final e será compartilhado entre as threads. Após isso, entra-se em um laço que percorre todo $p_i \in P$. Dentro do laço, várias threads T2 são criadas para distribuir o trabalho de calcular a similiridade entre p_i e todas as palavras de D. Após esse cálculo, ocorre um join, e R é ordenado e salvo.

Na figura 4, é mostrado o fluxo de execução da thread T2. Dentro dela, o algoritmo percorrerá um laço, calculando para cada $d_j \in D$, a similaridade entre um dado $p_i \in P$ e d_j, e salvará em R no formato de par ordenado, onde o primeiro componente é a similaridade e o segundo é a palavra d_j (o par $(\text{lev}_{p_i,d_i}(|p_i|, |d_j|), d_j)$).

Vale notar que é necessário um controle de concorrência para que as threads não repitam o trabalho já feito. Nos fluxogramas, esse controle é representado

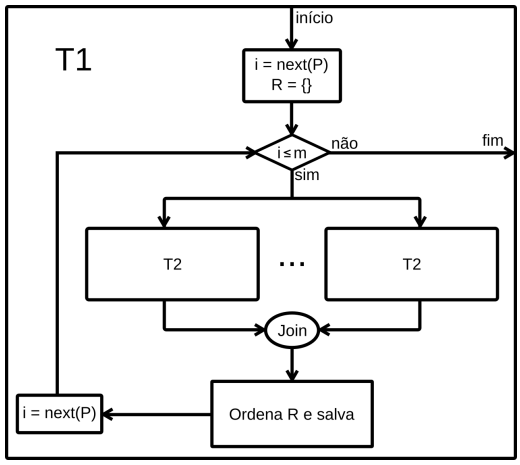


Figura 3: Fluxograma da Thread T1, mostrada na Figura 2.

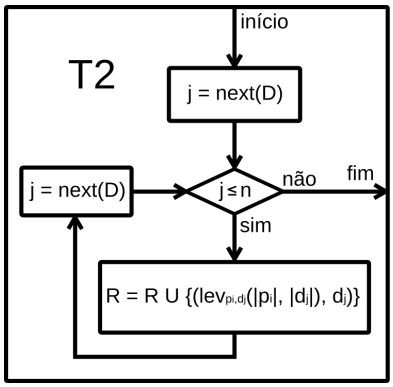


Figura 4: Fluxograma da Thread T2, mostrada na Figura 3.

por uma função *next*, que retorna a menor posição cujo o elemento ainda não foi computado. Na seção seguinte, serão apresentadas as diferentes maneiras que esse controle foi feito e comparadas todas essas abordagens.

3 Testes

Os testes realizados neste experimento foram executados em um notebook Dell Inspiron 14 Série 3000 (I14-3443-A40) com o sistema operacional *elementary OS 0.4.1 Loki (64-bit)*.

Foram utilizadas algumas ferramentas para auxiliar nos testes, sendo elas:

- **JCStress:** Responsável por realizar testes de estresse a fim de encontrar condições de corrida em algoritmos concorrentes.

- **JMH:** Responsável por realizar testes de microbenchmark, ou seja, medir a eficiência de certos partes de um programa.

Nas seções seguintes, compararemos a abordagem sequencial e as abordagens concorrentes utilizando mutex ou semáforo para controles de concorrência.

3.1 JCStress

Como o objetivo do JCStress é procurar condições de corrida em certas estruturas, os testes serão feitos somente sobre estruturas de dados compartilhadas entre threads no algoritmo concorrente tanto para a abordagem usando mutex, quanto para a abordagem usando semáforo.

As classes testadas nesse experimento são *CounterMutex, PairMutex, StringsMutex, CounterSemaphore, PairsSemaphore e StringsSemaphore.*

3.1.1 Counter

As classes *CounterMutex* e *CounterSemaphore* implementam a interface *Counter*, que tem como comportamento principal, retornar o próximo valor de um contador toda vez que o seu método *next()* for chamado. Portanto, um possível teste de concorrência seria duas threads chamarem esse método, e ele retornar os valores 0 e 1, ou 1 e 0, mas nunca 0 e 0, ou 1 e 1.

Observed states

Observed state	TC 1	TC 2	TC 3	TC 4	Expectation
0, 1	10755981	9129972	12108265	1611803	ACCEPTABLE
1, 0	11108840	7675799	9405006	1793898	ACCEPTABLE
	OK	OK	OK	OK	

Figura 5: Resultados do teste de JCStress para a classe *CounterMutex*.

Observed states

Observed state	TC 1	TC 2	TC 3	TC 4	Expectation
0, 1	10778722	10325902	9657577	943893	ACCEPTABLE
1, 0	9125949	8902419	8523434	590708	ACCEPTABLE
	OK	OK	OK	OK	

Figura 6: Resultados do teste de JCStress para a classe *CounterSemaphore*.

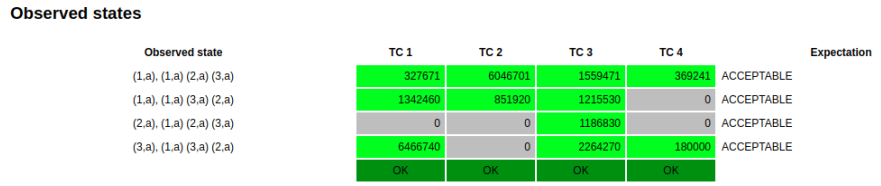
As Figuras 5 e 6 mostram os resultados dos testes estresse para as classes *CounterMutex* e *CounterSemaphore* respectivamente.

3.1.2 Pairs

As classes *PairsMutex* e *PairsSemaphore* implementam a interface *Pairs*, que tem como papel principal encapsular uma lista de objetos do tipo *Pair*, per-

mitindo somente adicionar novos elementos, recuperar o valor de um elemento, ordenar a lista e recuperar o tamanho da lista.

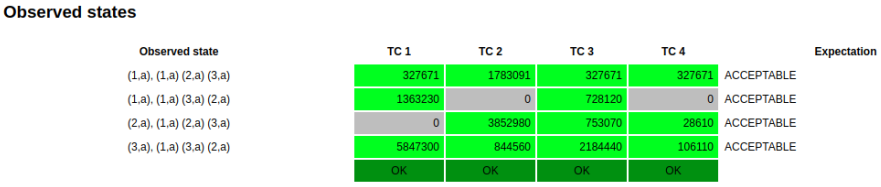
Para testar essa estrutura, precisamos executar simultaneamente as operações que modificam a lista (adicionar e ordenar) e a operação de recuperar um elemento da lista. Para uma lista L, caso essas operações sejam executadas em determinada ordem, o valor de L pode variar. Esses possíveis valores são o conjunto de valores aceitáveis pelo teste.



Observed states

Observed state	TC 1	TC 2	TC 3	TC 4	Expectation
(1,a), (1,a) (2,a) (3,a)	327671	6046701	1559471	369241	ACCEPTABLE
(1,a), (1,a) (3,a) (2,a)	1342460	851920	1215530	0	ACCEPTABLE
(2,a), (1,a) (2,a) (3,a)	0	0	1186830	0	ACCEPTABLE
(3,a), (1,a) (3,a) (2,a)	6466740	0	2264270	180000	ACCEPTABLE
	OK	OK	OK	OK	

Figura 7: Resultados do teste de JCStress para a classe *PairsMutex*.



Observed states

Observed state	TC 1	TC 2	TC 3	TC 4	Expectation
(1,a), (1,a) (2,a) (3,a)	327671	1783091	327671	327671	ACCEPTABLE
(1,a), (1,a) (3,a) (2,a)	1363230	0	728120	0	ACCEPTABLE
(2,a), (1,a) (2,a) (3,a)	0	3852980	753070	28610	ACCEPTABLE
(3,a), (1,a) (3,a) (2,a)	5847300	844560	2184440	106110	ACCEPTABLE
	OK	OK	OK	OK	

Figura 8: Resultados do teste de JCStress para a classe *PairsSemaphore*.

As Figuras 7 e 8 mostram os resultados dos testes de estresse para as classes *PairsMutex* e *PairsSemaphore* respectivamente, onde os "Observed state"são todos os possíveis resultados da lista $L = [(3, a), (1, a)]$ após passar por uma operação de ordenação, uma operação de consulta e uma operação adição de um novo elemento (elemento $(2, a)$) à lista. O primeiro elemento de um "state"é o resultado obtido pela operação *get()*, e o segundo elemento é uma lista com o resultado final de L.

3.1.3 Strings

As classes *StringsMutex* e *StringsSemaphore* implementam a interface *Strings*, que tem o mesmo papel da classes que implementam *Pairs*, porém, em vez de encapsular uma lista de *Pair*, encapsula uma lista de *String*, porém não possuem o método de ordenação. Por causa disso, os testes com as implementações de *Strings* são análogos aos testes com as implementações de *Pairs*, exceto a operação de ordenação.

As Figuras 9 e 10 mostram os resultados referentes às classes *StringsMutex* e *StringsSemaphore*. Como os testes são os mesmos das classes que implementam *Pairs*, exceto pela exclusão da operação de ordenação, a lista e o seu segundo

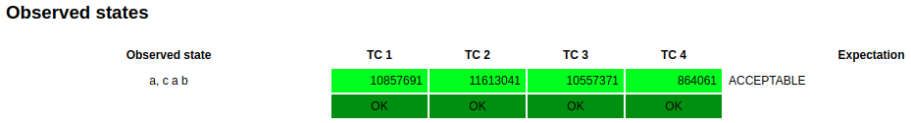


Figura 9: Resultados do teste de JCStress para a classe *StringsMutex*.

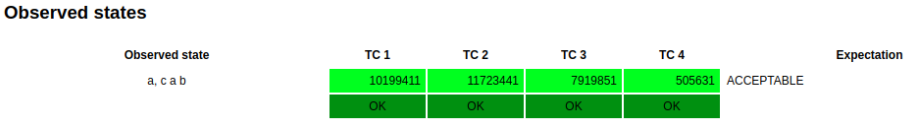


Figura 10: Resultados do teste de JCStress para a classe *StringsSemaphore*.

elemento possuem um valor fixo. Logo, o "Observed state"é único para este teste.

3.2 JMH

Nos testes do JMH foram testados os métodos principais do software, que são o método *read*, que lê o dicionário e as palavras de entrada, e o método *calculateAndWrite*, que calcula a similaridade de cada palavra de entrada para cada palavra de saída e salva os resultados em um arquivo. Os testes foram feitos para esses métodos usando a abordagem sequencial, concorrente com mutex e concorrente com semáforo.

Para a realização dos testes, foram utilizados os seguintes parâmetros: 2 Forks de warmup e 3 Forks de medição, cada Fork possuindo 5 operações de warmup e 10 iterações de medição. O modo do benchmark foi Average-Time e a unidade de tempo utilizada foi milissegundos.

Método	**Abordagem**	**média (ms/op)**	**Ganho (%)**
read	sequencial	22,254	0
	mutex	24,565	-10,38
	semáforo	27,615	-24,09
calculateAndWrite	sequencial	14129,976	0
	mutex	7531,801	46,69
	semáforo	7578,752	46,36

Tabela 1: Resultados dos testes usando o JMH.

A tabela 1 representa o resultado dos testes usando o JMH. Na primeira coluna, estão os principais métodos do algoritmo que foram testados. Na segunda coluna, qualifica qual abordagem foi usada, associando-a com o método correspondente. Na terceira coluna estão os resultados médios dos tempos de execução de cada método, para cada técnica medido em milissegundos. Já na

quarta coluna, estão as porcentagens de ganho em relação à abordagem sequencial.

Com base nesses dados, podemos perceber que as abordagens concorrentes foram quase duas vezes mais rápidas do que a abordagem sequencial no método *calculateAndWrite*, porém, no método *read*, o algoritmo se saiu um pouco melhor.

4 Conclusão

Dados os resultados dos testes acima, podemos concluir que, de modo geral, as duas abordagens do algoritmo concorrente obtiveram resultados relativamente iguais, enquanto que o algoritmo sequencial obteve um resultado pior. Porém, nos testes referentes à leitura de arquivo, o algoritmo sequencial se saiu um pouco melhor, mas com uma diferença bem pequena.