ALESSANDRO SOARES DA SILVA

MATRICULA: 20231023705

11° LISTA DE ALGORITMO

1. Mostre experimentalmente que a taxa de competitividade do algoritmo de mover para a frente se aproxima de 4 estatisticamente.

O algortimo de mover para a frente está relacionado aos algoritmos online por conta de sua abordagem iterativa e reativa. Algoritmos online são aqueles que tomam decisão com base em informações disponívies no momento da tomada de decisão, sem acesso ao conjunto completo de dados. Eles precisam tomar decisões adaptativas à medida que novos dados chegam.

O algoritmo de mover para a frente é uma técnica que opera de forma similar. Ele percorre sequencialmente uma lista (ou qualquer outra estrutura de dados) e toma decisões com base nos elementos encontrados até aquele ponto. A medida que novos elementos são encontrados, ele ajusta seu estado interno sem ter acesso prévio a lista de valores.

Essa relação com algoritmos online se estabelece por que, em contextos online, muitas vezes você precisa tomar decisões com base em dados progressivamente relevaldos. No caso desse algoritmos, eke só vê um elemento por vez e decide se vai atualizar ou não o máximo, assim como um algoritmo online tomaria decisões baseados em dados progressivos.

Implementação do algoritmo.

```python
import random
import matplotlib.pyplot as plt
import numpy as np

def foresee(lst, searches):
    search_cost = 0
    swap_cost = 0
    for x in searches:
        if x in lst:
            search_cost += 1
            pos = lst.index(x)
            prev_swap_cost = pos  # Custo previsto da troca
            # Calcula o custo real da troca se x for movido para a frente
            lst.remove(x)
            lst.insert(0, x)
            real_swap_cost = lst.index(x)

            if real_swap_cost != prev_swap_cost:
                swap_cost += 1
    return search_cost, swap_cost

def move_to_front(lst, searches):
    search_cost = 0
    swap_cost = 0
    for x in searches:
        if x in lst:
            search_cost += 1
            pos = lst.index(x)
            swap_cost += pos
            lst.insert(0, lst.pop(pos))
    return search_cost, swap_cost
```

```python
# Exemplo
competitive_ratios = []
for i in range(1, 80):
    input_size = 10
    lst = [random.randint(1, 10) for _ in range(input_size)]
    #lst = [j for j in range (1, 11)]
    print("\n",lst)

    input_searches = 4
    searches = [random.randint(1, 10) for _ in range(input_searches)]
    print(searches)
```

```python
    foresee_search_cost, foresee_swap_cost = foresee(lst[:], searches)
    move_to_front_search_cost, move_to_front_swap_cost = move_to_front(lst[:], searches)

    print("Custo de busca FORESEE:", foresee_search_cost)
    print("Custo de troca FORESEE:", foresee_swap_cost)
    print("Custo de busca MOVE-TO-FRONT:", move_to_front_search_cost)
    print("Custo de troca MOVE-TO-FRONT:", move_to_front_swap_cost)
    total_foresee = (foresee_search_cost + foresee_swap_cost)
    total_move_to_front = (move_to_front_search_cost + move_to_front_swap_cost)
    print("Custo Total FORESEE:", total_foresee)
    print("Custo Total MOVE-TO-FRONT:", total_move_to_front)
    if total_foresee != 0:
        competitive_ratio = total_move_to_front / total_foresee
        competitive_ratios.append(competitive_ratio)

# Calculando o desvio padrão das amostras
std_deviation = np.std(competitive_ratios)

# Criando o primeiro gráfico estatístico
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.plot(competitive_ratios, marker='o', linestyle='-', color='b')
plt.title('Custos Competitivos')
plt.xlabel('Iteração')
plt.ylabel('Custo Competitivo')
plt.grid(True)
```

```python
# Criando o segundo gráfico com desvio padrão centralizado em 3
plt.subplot(1, 2, 2)
plt.errorbar(range(len(competitive_ratios)), competitive_ratios, yerr=std_deviation, fmt='o',
plt.axhline(y=2.5, color='red', linestyle='--', label='Centro em 2.5')
plt.title('Desvio Padrão das Amostras entre 0 e 5')
plt.xlabel('Iteração')
plt.ylabel('Custo Competitivo')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

Plotamos doi gráficos para representar a distribuição dos dados encontrados, fizemos testes com vetores que variavam de tamanho 10 a 50, e fizemos 100 iterações para acumular os valores e gerar o gráfico visto na figura 1.

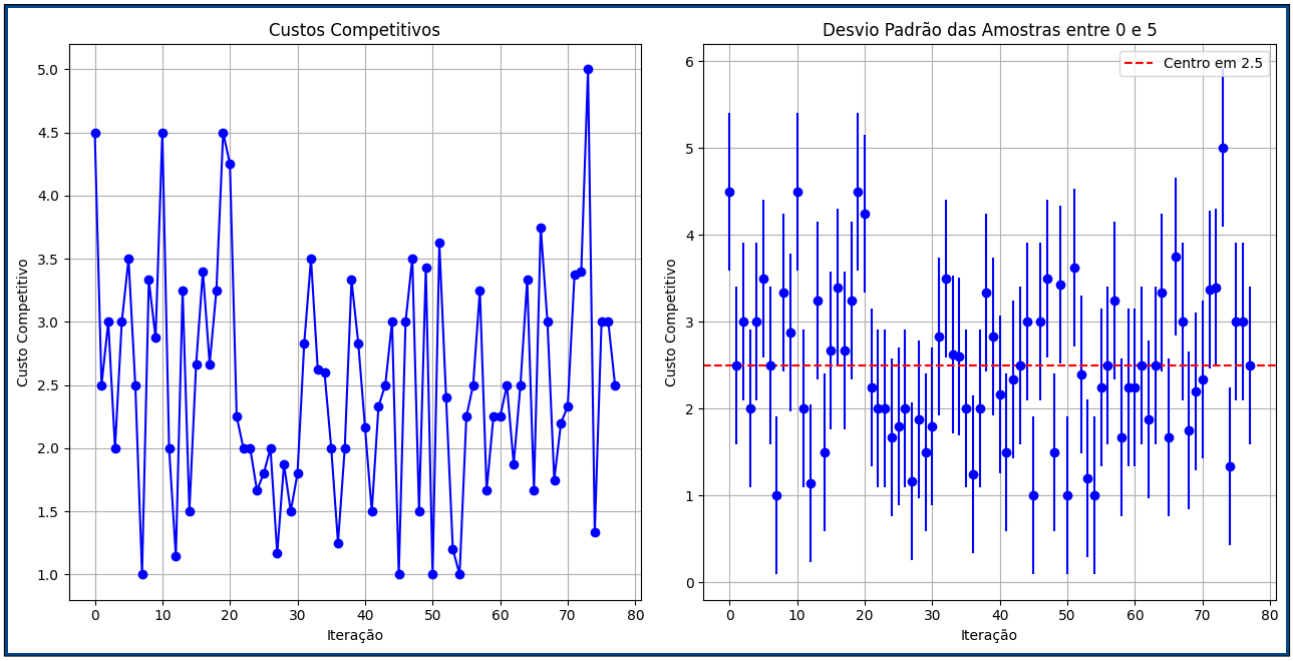


Figura 1

Apesar de termos alguns valores na escala de 1 e poucos na escala de acima de 4, todos os outros estão em uma faixa que se encontram entre 2 e 3.5.

2. Mostre experimentalmente que a taxa de competitividade do algoritmo de uso menos recente tem taxa de competitividade independente do tamanho da entrada para o mecanismo de memória cache.

FIFO e LRU têm uma relação competitiva igual a Θ(k). O tamanho da cache k é independente da sequência de entrada e não cresce corforme mais solicitações que chegam com o tempo. Uma relação competitiva que depende de n, por outro lado cresce com o tamanho da sequência de entrada e portanto, pode ficar bem grande. É preferível usar um algoritmo com porporção competitiva que não cresce com o tamanho da sequência de entrada, quando possível.

De modo que o teorema 27.4 do livro determina que: qualquer algoritmo determinístico online para armazenamento em cache com tamanho de cache com tamanho de cache "k" tem razão competitiva Θ(k).

IMPLEMENTAÇÃO DO ALGORITMO

```python
import matplotlib.pyplot as plt
import random
import numpy as np

def lru_cache_miss(sequence, k):
    cache = []
    cache_misses = 0
    change_page_lru = 0
    for item in sequence:               # Lê a sequencia gerada
        if item not in cache:           # Se o item nao estiver na cache segue para o confirmação de estouro de cache
            if len(cache) == k:         # Confirma estouro da cache
                cache.pop(0)            # Se estourou a tamanho retira o elemeto menos usado
                change_page_lru += 1    # Imcrementa mudança de pagina
            cache.append(item)          # Se não estourou o tamanho da cache insere o valor na cache
            cache_misses += 1           # Incrementa o cache missing
        else:
            cache.remove(item)          # Se o elemento estiver na lista so mantem o elemento na lista
            cache.append(item)          # Sem gerar incremento de pagina ou cache missing

    return cache_misses, change_page_lru
```

```python
def optimal_cache_miss(sequence, k):
    cache = []
    cache_misses = 0
    most_common = 0
    change_page_optimal = 0
    i = 0
    for item in sequence:                # Faz a varredur da lista
        i +=1
        if item not in cache:            # Se o item nao estiver na cache segue para o confirmação de estouro de cache
            if len(cache) == k:          # Confirma estouro da cache
                for count in range (i, len(sequence)):   # Neste bloco é simulado o oracle
                    if item == sequence[count]:          # Pois ele verifica se o item em questão irá se repetir na lista
                        most_common +=1
            if (most_common >= 2):             # Se o valor se repetir faz inserção do elemento, deletando o mais antigo
                cache.pop(0)
                cache.append(item)             # Adiciona apenas se houver mais de uma ocorrência
                change_page_optimal += 1
                most_common = 0                # Atualizar a contagem
                cache_misses += 1              # Incrementa o cache missing
            else:
                cache.append(item)             # insere o valor na lista caso nao tenha estourado a cache
                cache_misses += 1              # Incrementa o cache missing

    return cache_misses, change_page_optimal
```

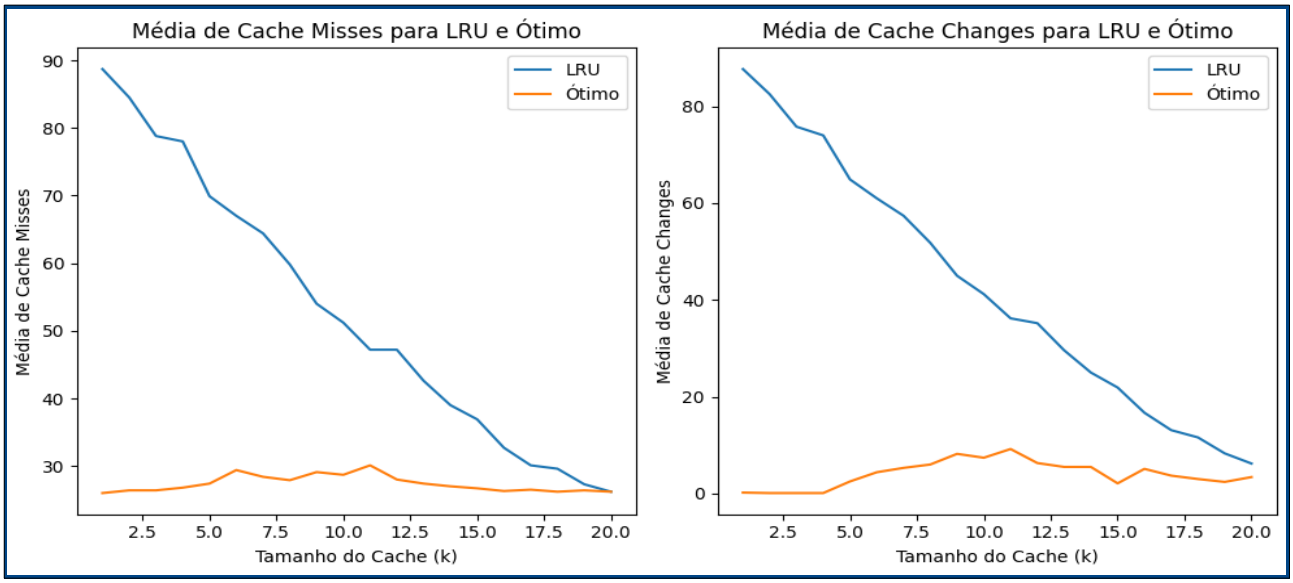
```python
# Listas para armazenar os resultados
lru_misses_avg = []
lru_changes_avg = []
optimal_misses_avg = []
optimal_changes_avg = []

# Repetir os testes 10 vezes com k variando de 1 a 10
for k_value in range(1, 21):
    lru_misses_list = []
    lru_changes_list = []
    optimal_misses_list = []
    optimal_changes_list = []

    for _ in range(10):
        # Criar a sequência
        sequence = [random.randint(1, 10) for _ in range(10)]
        sequence.extend([random.randint(11, 30) for _ in range(85)])

        # Obter os resultados para LRU e ótimo
        lru_misses, lru_changes = lru_cache_miss(sequence, k_value)
        optimal_misses, optimal_changes = optimal_cache_miss(sequence, k_value)

        # Armazenar os resultados em listas
        lru_misses_list.append(lru_misses)
        lru_changes_list.append(lru_changes)
        optimal_misses_list.append(optimal_misses)
```

```python
    # Calcular as médias dos resultados para cada k
    lru_misses_avg.append(np.mean(lru_misses_list))
    lru_changes_avg.append(np.mean(lru_changes_list))
    optimal_misses_avg.append(np.mean(optimal_misses_list))
    optimal_changes_avg.append(np.mean(optimal_changes_list))

# Plotar os gráficos das médias de cache misses e cache changes
x = list(range(1, 21))  # Valores de k de 1 a 10

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.plot(x, lru_misses_avg, label='LRU')
plt.plot(x, optimal_misses_avg, label='Ótimo')
plt.xlabel('Tamanho do Cache (k)')
plt.ylabel('Média de Cache Misses')
plt.title('Média de Cache Misses para LRU e Ótimo')
plt.legend()
```

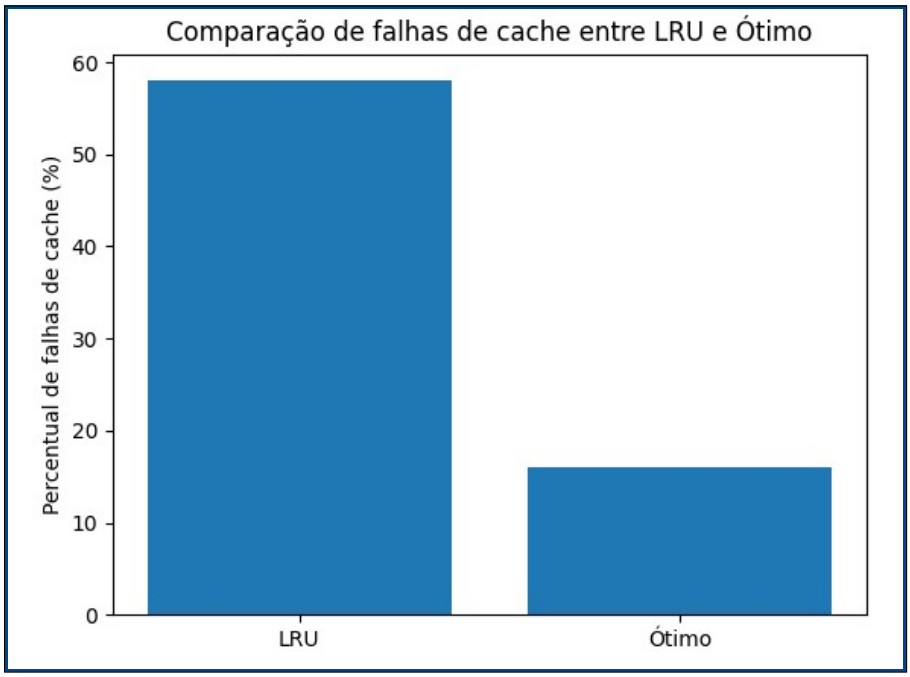
```python
plt.subplot(1, 2, 2)
plt.plot(x, lru_changes_avg, label='LRU')
plt.plot(x, optimal_changes_avg, label='Ótimo')
plt.xlabel('Tamanho do Cache (k)')
plt.ylabel('Média de Cache Changes')
plt.title('Média de Cache Changes para LRU e Ótimo')
plt.legend()

plt.tight_layout()
plt.show()
```

Temos os gráficos gerados, o primeiro e quantidade de cache misses, e o segundo e quantidade de mudança de paginas em função da sequencia de entrada. O gráfico mais relevante aqui é o segundo, pois ele nos mostra que a relação Θ(k) demonstrada no teorema, a medida que aumentamos a capacidade da cache, temos a diminuição da quantidade de mudança de página.

Aqui usamos entradas que variaram de 20 a 100 valores, e uma variação de tamanho de cache que oscilou de 1 a 20, entendemos que podemos tirar uma noa conclusão com a análise do gráfico. Verificamos também que um algortimo ótimo tem uma grande vantagem por saber as características de entradas com antecedência, pois no gráfico sua variação e bem estável.



Algoritmos de caching direrem na forma de como removerão (evict) na ocorrência de um cache misses e a cache estiver cheia. O objetivo é minimizar o número de misses ao longo de toda a sequência. Na algoritmo ótimo a taxa de cache hit foi de 45% e a taxa de cache miss foi de 52%. Como no algoritmo online (LRU) é usada a estratégia de "uso menos recente", que significa que o bloco removido será aquele cujo uso está mais distante no uso. A taxa de cache hit foi de 37%, enquanto a taxa de cache misses foi de 76%. o que nos indica que a taxa c-competivo tem valores aproximados de 1.3 a 1.8, dependendo do tamanho da cache (k). No entanto, o algoritmo implementado ainda teve interferencia do tamanho da entra, embora em níveis modestos, o que significa que temos que melhorar a implementação.



3. Compare o desempenho do algoritmo de marcação aleatória com o algoritmo de uso menos recente para qualquer um dos programas que já implementou na disciplina. Dica: extraia o traço de memória do seu programa e use-o como entrada para a sua comparação.

Algoritmos "Randomized Marking". Nessa estratégia, os blocos usados são marcados e blocos não marcados são removidos numa selação randomica. Quando todos os blocos estão marcados, todos são desmarcados simultâneamente.

```
RANDOMIZED-MARKING(b)
1  if block b resides in the cache,
2       b.mark = 1
3  else
4       if all blocks b' in the cache have b'.mark = 1
5            unmark all blocks b' in the cache, setting b'.mark = 0
6       select an unmarked block u with u.mark = 0 uniformly at random
7       evict block u
8       place block b into the cache
9       b.mark = 1
```

ALGORITMO IMPLEMENTADO

```python
import matplotlib.pyplot as plt
import random
import numpy as np

# Implementação do LRU original
def lru_cache_miss(sequence, k):
    cache = []
    cache_misses = 0
    change_page_lru = 0
    for item in sequence:
        if item not in cache:
            if len(cache) == k:
                cache.pop(0)
                change_page_lru += 1
            cache.append(item)
            cache_misses += 1
        else:
            cache.remove(item)
            cache.append(item)

    return cache_misses, change_page_lru
```

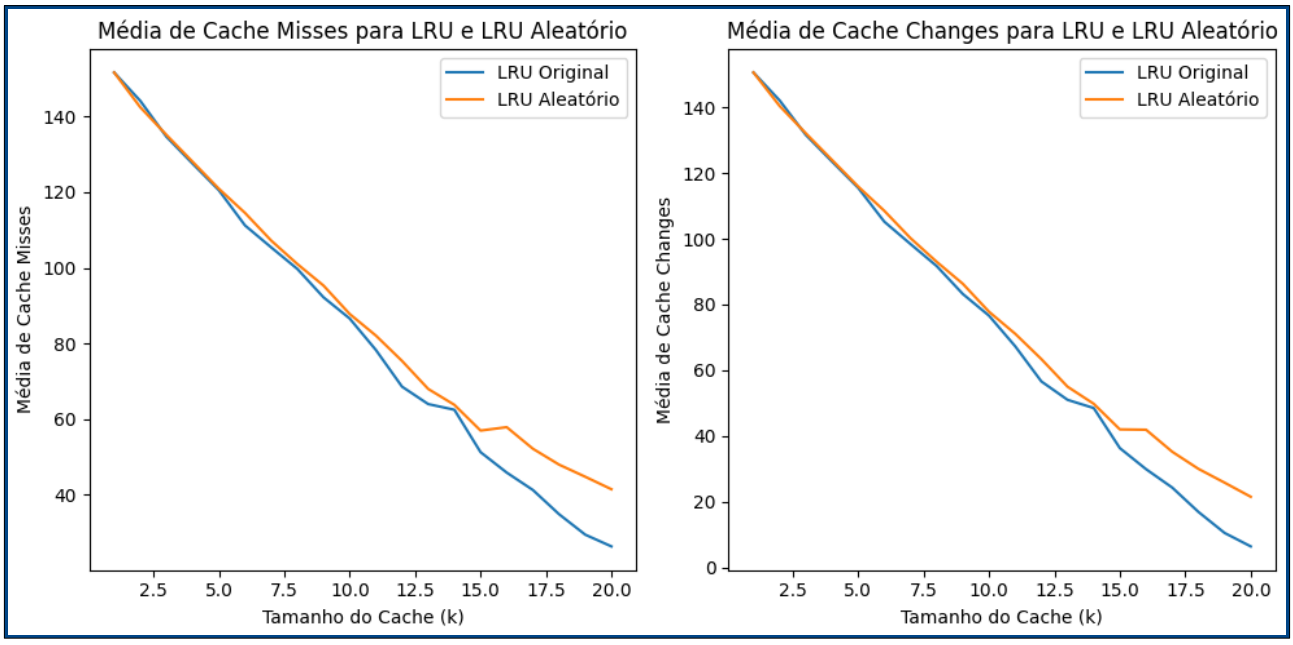
```python
# Implementação do LRU com escolha aleatória
def lru_random_cache_miss(sequence, k):
    cache = []
    cache_misses = 0
    change_page_lru = 0
    for item in sequence:
        if item not in cache:
            if len(cache) == k:
                # Escolher aleatoriamente um índice para remover da cache
                index_to_remove = random.randint(0, k - 1)
                cache.pop(index_to_remove)
                change_page_lru += 1
            cache.append(item)
            cache_misses += 1
        else:
            cache.remove(item)
            cache.append(item)

    return cache_misses, change_page_lru
```

```python
# Listas para armazenar os resultados
lru_misses_avg = []
lru_changes_avg = []
lru_random_misses_avg = []
lru_random_changes_avg = []

# Repetir os testes 10 vezes com k variando de 1 a 20
for k_value in range(1, 21):
    lru_misses_list = []
    lru_changes_list = []
    lru_random_misses_list = []
    lru_random_changes_list = []

    for _ in range(10):
        # Criar a sequência
        sequence = [random.randint(1, 10) for _ in range(10)]
        sequence.extend([random.randint(11, 30) for _ in range(150)])

        # Obter os resultados para LRU e LRU aleatório
        lru_misses, lru_changes = lru_cache_miss(sequence, k_value)
        lru_random_misses, lru_random_changes = lru_random_cache_miss(sequence, k_value)

        # Armazenar os resultados em listas
        lru_misses_list.append(lru_misses)
        lru_changes_list.append(lru_changes)
        lru_random_misses_list.append(lru_random_misses)
        lru_random_changes_list.append(lru_random_changes)
```

```
    # Calcular as médias dos resultados para cada k
    lru_misses_avg.append(np.mean(lru_misses_list))
    lru_changes_avg.append(np.mean(lru_changes_list))
    lru_random_misses_avg.append(np.mean(lru_random_misses_list))
    lru_random_changes_avg.append(np.mean(lru_random_changes_list))

# Plotar os gráficos das médias de cache misses e cache changes
x = list(range(1, 21))  # Valores de k de 1 a 20

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.plot(x, lru_misses_avg, label='LRU Original')
plt.plot(x, lru_random_misses_avg, label='LRU Aleatório')
plt.xlabel('Tamanho do Cache (k)')
plt.ylabel('Média de Cache Misses')
plt.title('Média de Cache Misses para LRU e LRU Aleatório')
plt.legend()
```

```
plt.subplot(1, 2, 2)
plt.plot(x, lru_changes_avg, label='LRU Original')
plt.plot(x, lru_random_changes_avg, label='LRU Aleatório')
plt.xlabel('Tamanho do Cache (k)')
plt.ylabel('Média de Cache Changes')
plt.title('Média de Cache Changes para LRU e LRU Aleatório')
plt.legend()

plt.tight_layout()
plt.show()
```

GRÁFICOS DA APLICAÇÃO



Como é possível observar o desempenho dos dois algoritmos praticamente o mesmo. A literatura afirma que o algortimo de marcação randomizada deve ter uma taxa de competitividade esperada muito menor, com limite superio Θ(log(k)). Aqui ela se comporta como a LRU, que tem limite Θ(k).

Possíveis causas, podem incluir a mesma entrada randomizada utilizada na qustão 2, uma vez que não consegui gerar valores reais, a partir do traço de memória de uma programa.