TP3 de Algoritmos e Estruturas de Dados III

Lucas O. Silvestre[1]

[1]Sistemas de Informação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, MG – Brazil

lsilvs@dcc.ufmg.br

1. Objetivo inicial

O trabalho prático em questão visa consolidar os conhecimentos do aluno no que diz respeito a virtualização de memória. Para tal, é proposto a implementação de um Sistema de Memória Virtual. O SMV serve para que os programas que rodam em determinada arquitetura enxerguem um espaço contínuo de memória, facilitando sua execução. A cada acesso solicitado pelo programa, o SMV deve verificar se o dado já encontra-se na memória. Caso já esteja lá, o programa acessa normalmente. Caso não esteja (situação que chamamos de page miss, page fault ou falha), o SMV busca o dado no disco. Se ainda houver espaça em memória, o dado é armazenado. Caso não haja mais espaço, é retirado algum dado para substituição. O dado que será removido dependerá da política de remoção utilizada.

Aqui, iremos simular, de forma simplificada (apenas um nível de paginação, sem caches ou otimizações), utilizando três métodos de reposição de páginas. São eles: FIFO (a primeira página a entrar será a primeira a sair), LRU (a página acessada a mais tempo dará lugar a uma nova) e LFU (a página menos acessada dará lugar à nova).

Além disso, para agilizar futuros acessos, o SMV pode armazenar na memória primária os dados vizinhos ao que foi solicitado. Isso porque há uma probabilidade grande desses dados serem requisitados em acessos próximos.

2. Modelagem e Solução Proposta

A solução proposta foi implementada utilizado uma estrutura de fila com Lista Duplamente Encadeada. Dessa forma, as novas páginas são inseridas no início da fila e as páginas escolhidas para remoção são retiradas do fim da fila. A estrutura Lista possui quantidade (páginas armazenadas), numero de páginas livres (páginas ainda disponíveis na memória), número de falhas (contabiliza quantas falhas determinada sequência de

acessos gerou) e um ponteiro para o primeiro e o último elemento da lista. O elemento citado acima é uma estrutura que possui valor (endereço da página), acessos (número de vezes que a página já foi acessada), próximo (ponteiro para o próximo elemento) e anterior (ponteiro para o elemento anterior).

Além disso, o código implementa as funções inicializa (que seta os valores iniciais da lista criada), pesquisa (que retorna o ponteiro do elemento pesquisado pelo seu valor ou null caso não encontre), elimina (que retira da lista o elemento passado por parâmetro), insere (que insere um elemento no início da fila com o valor passado por parâmetro), ordena_by_acessos (que ordena a lista em ordem crescente de acessos) e desaloca_lista (que libera o espaço em memória da lista alocada).

A cada acesso lido do arquivo de entrada, os três métodos são evocados e salvam o estado atual de falhas no atributo de sua lista específica. Ao final da leitura, o valor total de falhas de cada método de substituição é gravado no arquivo de saída. Abaixo será apresentado cada método de substituição com mais detalhes.

2.1. FIFO

O método de substituição de páginas FIFO consiste em, caso a página já esteja cheia, retirar a primeira página inserida. Para sua implementação, primeiramente fazemos uma pesquisa para saber se a página desejada já se encontra na memória. Caso positivo, nada acontece (em uma situação real, o dado já estaria lá disponível e seria acessado pelo programa). Caso a página ainda não esteja na memória, o SMV verifica se ainda há espacó em memória para salvar a nova página. Se sim, simplesmente a insere no início da lista. Se não, remove o último elemento da lista (que na verdade foi o primeiro elemento a entrar) e então insere a nova página no início da lista. Sempre que a página não é encontrada na memória, gera uma falha e o contador é incrementado.

2.2. LRU

Para o LRU (Least Recently Used), utilizamos uma lógica similar à do FIFO, porém, caso a página já esteja em memória, ela é retirada e inserida novamente no início da fila. Isso porque, como ela acabou de ser requisitada, precisamos atualizar sua posição para que ela passe a ser a última na escolha de remoção (de acordo com a definição do LRU).

2.3. LFU

Para implementar o LFU (Least Frequently Used), precisamos manter e atualizar um contador de acesso a cada página da memória, pois esse método privilegia as páginas mais acessadas e remove as menos requisitadas. Ou seja, o princípio de funcionamento continua bem similar aos dois descritos acima, porém, a casa inserção realizamos a ordenação das páginas por número de acesso e por índice como critério de desempate.

3. Análise de Complexidade

Como a implementação utiliza uma lista encadeada, isso reduz significativamente a complexidade do algoritmo. As funções *elimina* e *insere* possuem complexidade O(1). Isso porque *insere* adiciona sempre no início da fila e *elimina* retira o elemento passado por parâmetro. Já a função *pesquisa* é, no pior caso O(n), sendo n o tamanho da lista no momento da pesquisa. Isso porque ela percorre toda a lista até encontrar o valor desejado. Já a função *ordena_by_acessos* é sempre O(n), sendo n o tamanho da lista no momento da ordenação. Isso porque ela percorre toda a lista rearranjando-a.

De posse dessa análise inicial, concluímos que as funções FIFO e LRU são, no pior caso, $O(n^2)$, onde n é o número de acessos. Com n sendo o número acessos realizados pelo programa que utiliza o SMV. Para o LFU, temos $O(n^2) + O(n)$ que se resume em $O(n^2)$.

4. Experimentos e Análises

Foi disponibilizado um arquivo para realizar os testes e análises para esse trabalho. O arquivo contem quatro instâncias com mais ou menos 200 entradas cada uma. Abaixo serão apresentados os gráficos e tabelas exigidos pela especificação do trabalho.

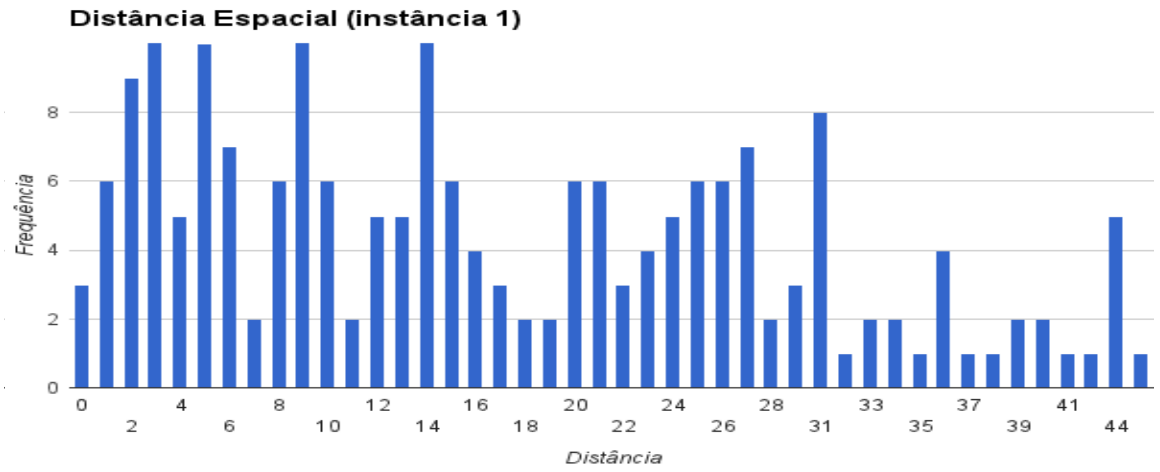
4.1. Tabela de Localidade de referência temporal e espacial

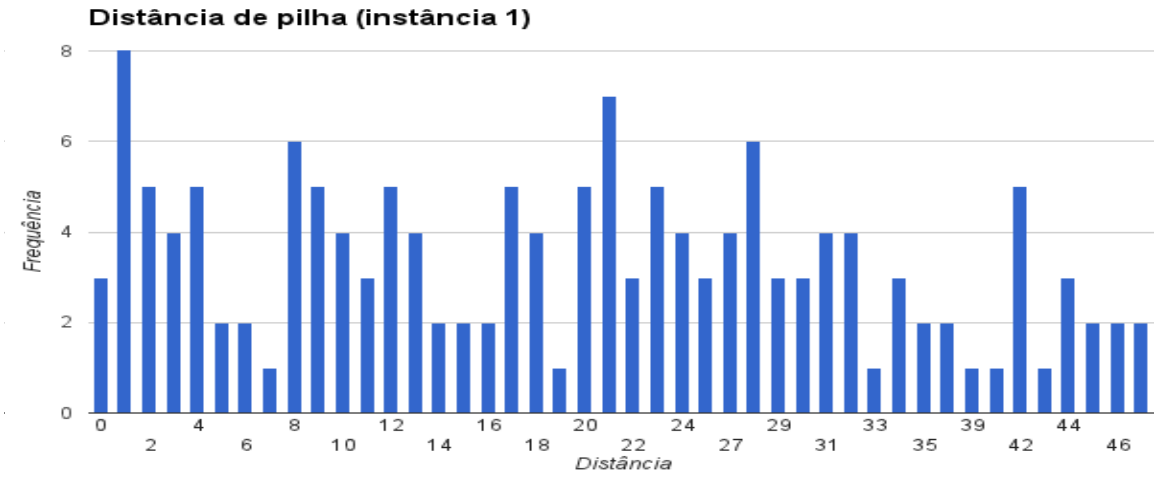
Segue abaixo a tabela da média dos valores de localidade de referência temporal e espacial:

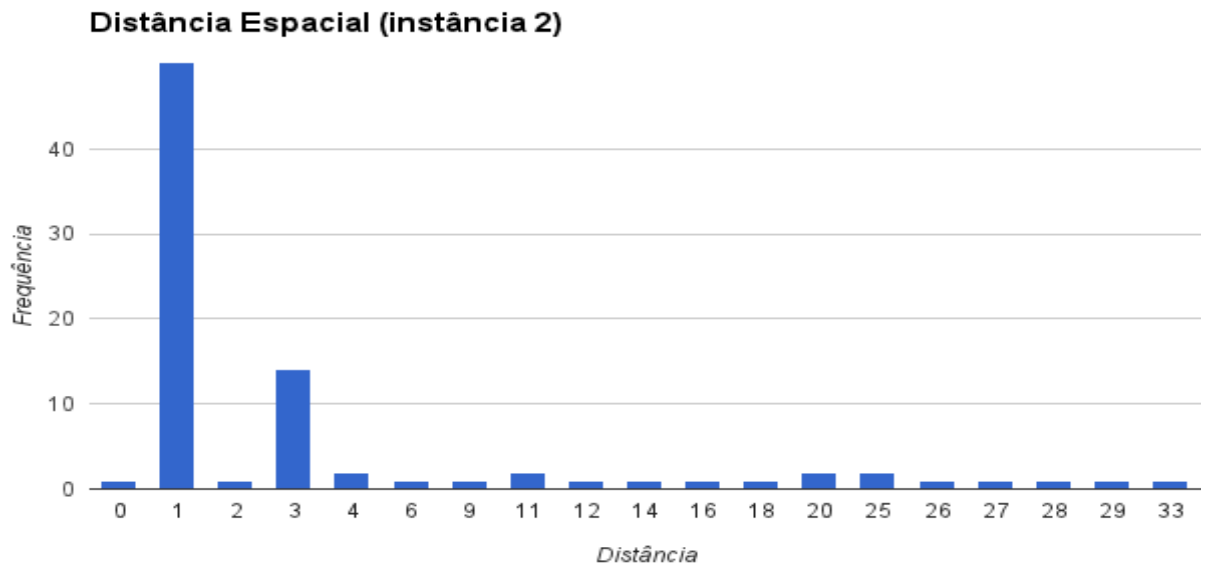
Instância	Espacial	Temporal
1	16.69	19.65
2	2.56	13.95
3	10.47	20.98
4	19.71	10.66

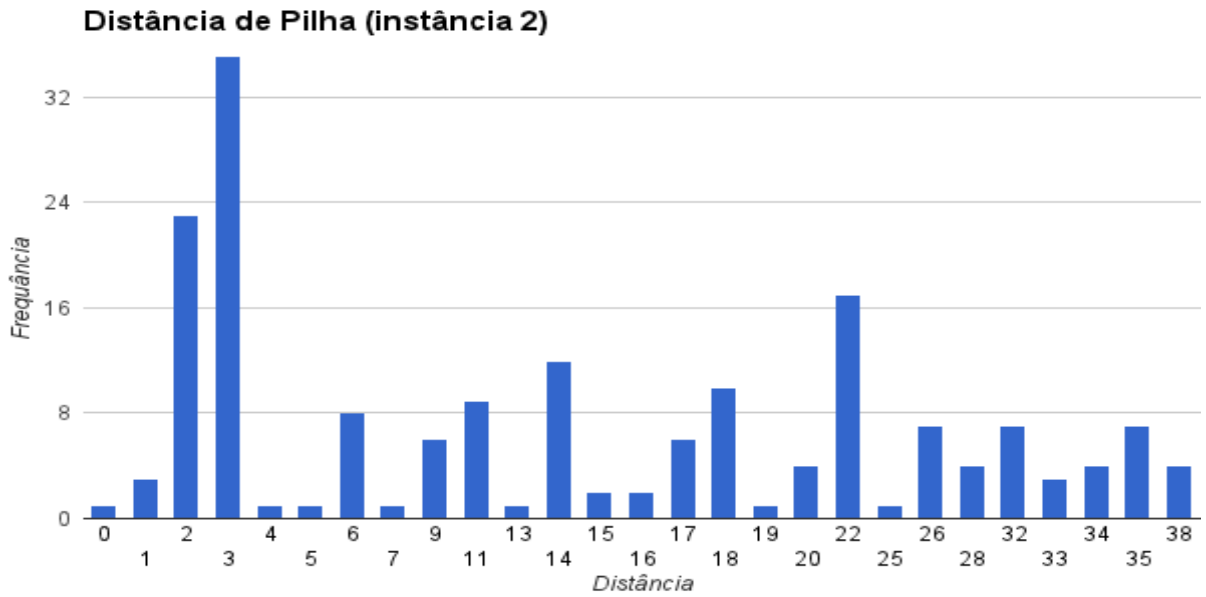
4.2. Histogramas das distâancias de acessos e de pilha

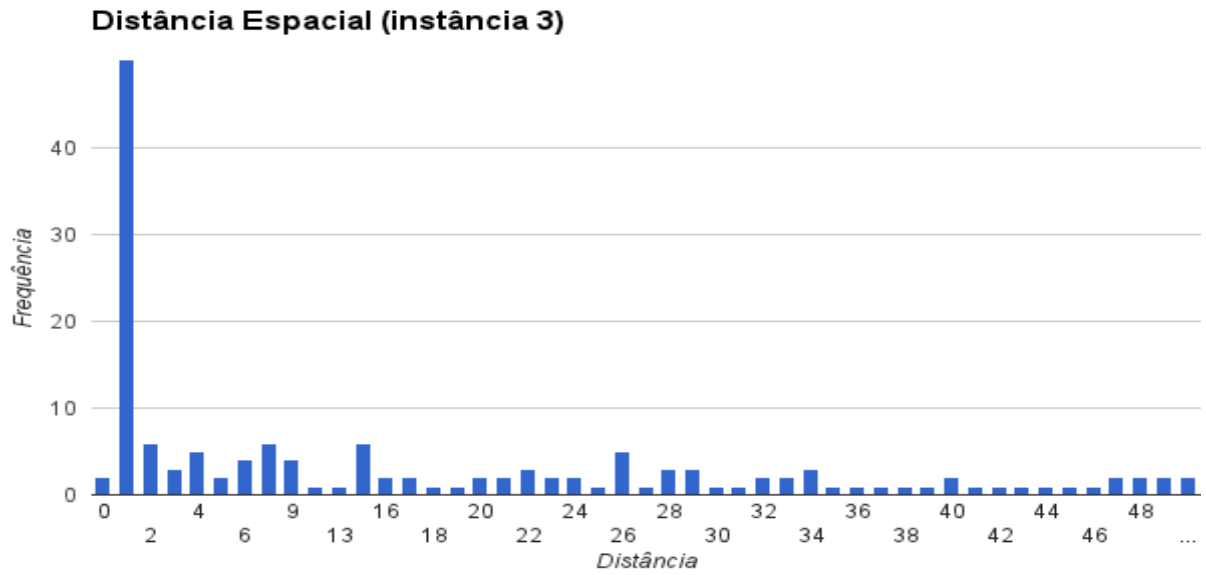
Apresentaremos os histogramas para localidade de referência temporal e espacial.

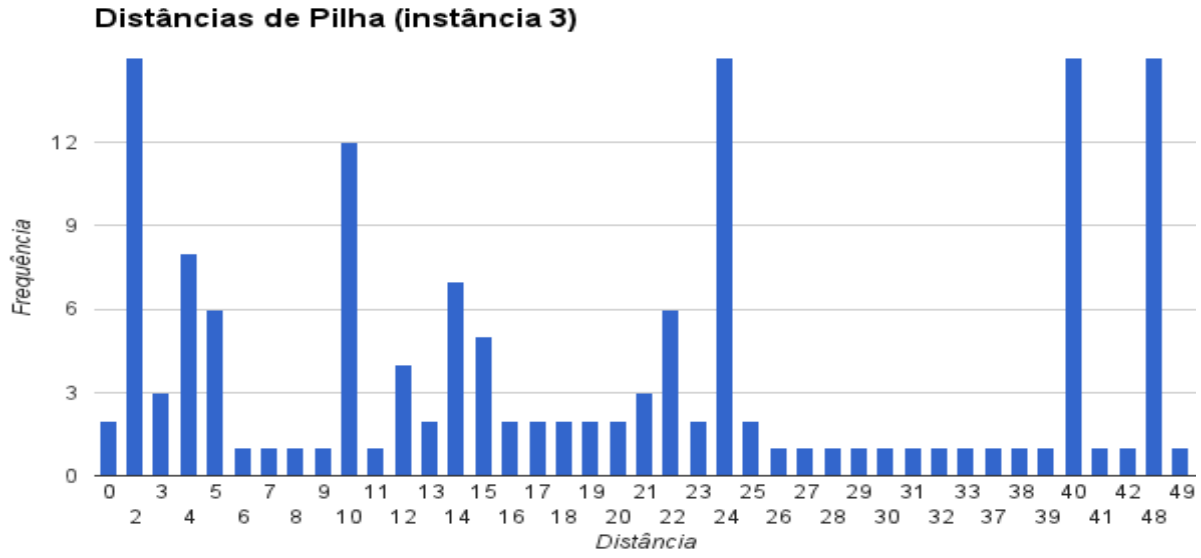


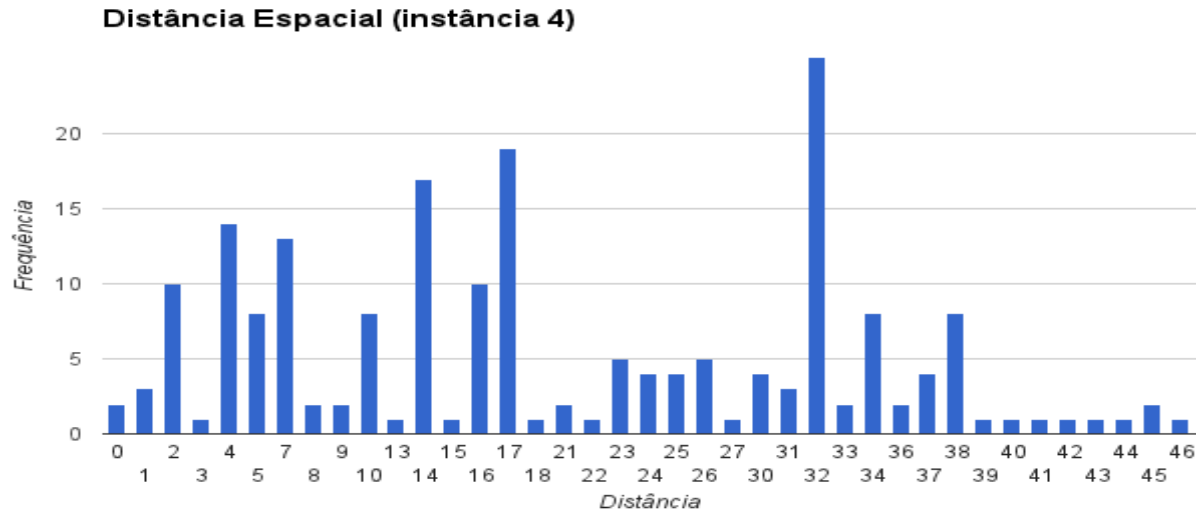


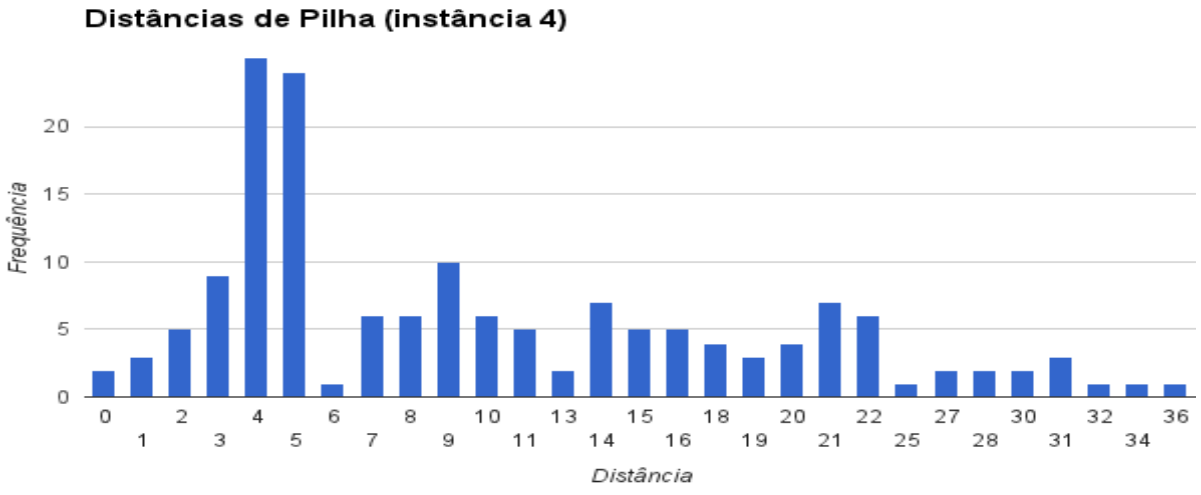

Distância Espacial (instância 2)
Frequência
40
30
20
10
0
0 1 2 3 4 6 9 11 12 14 16 18 20 25 26 27 28 29 33
Distância


Distância de Pilha (instância 2)
Frequância
32
24
16
8
0
0 1 2 3 4 5 6 7 9 11 13 14 15 16 17 18 19 20 22 25 26 28 32 33 34 35 38
Distância


Distância Espacial (instância 3)
Frequência
40
30
20
10
0
0 2 4 6 9 13 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 ...
Distância

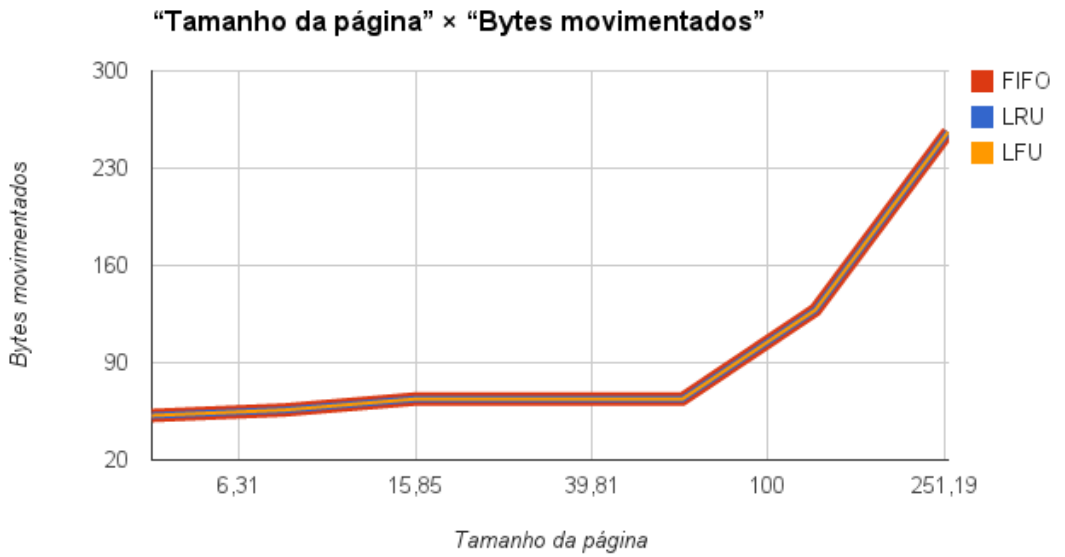

Distâncias de Pilha (instância 3)
Frequência
0
3
6
9
12
0 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 37 38 39 40 41 42 48 49
Distância


Distância Espacial (instância 4)
Frequência
0
5
10
15
20
0 1 2 3 4 5 7 8 9 10 13 14 15 16 17 18 21 22 23 24 25 26 27 30 31 32 33 34 36 37 38 39 40 41 42 43 44 45 46
Distância

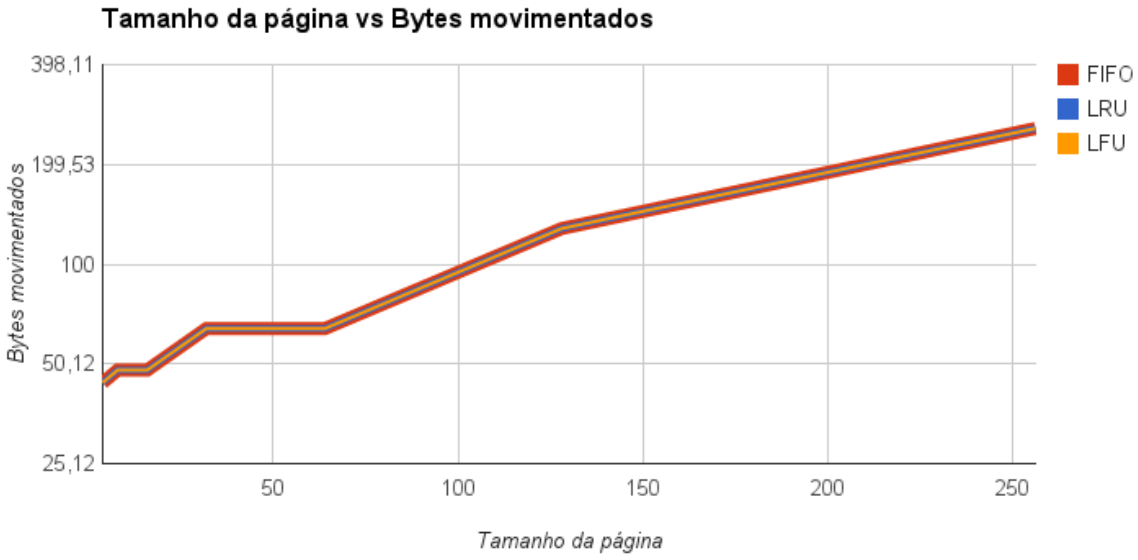

Distâncias de Pilha (instância 4)
Frequência
0
5
10
15
20
0 1 2 3 4 5 6 7 8 9 10 11 13 14 15 16 18 19 20 21 22 25 27 28 30 31 32 34 36
Distância

4.3. Gráfico “Tamanho da página” × “Bytes movimentados”

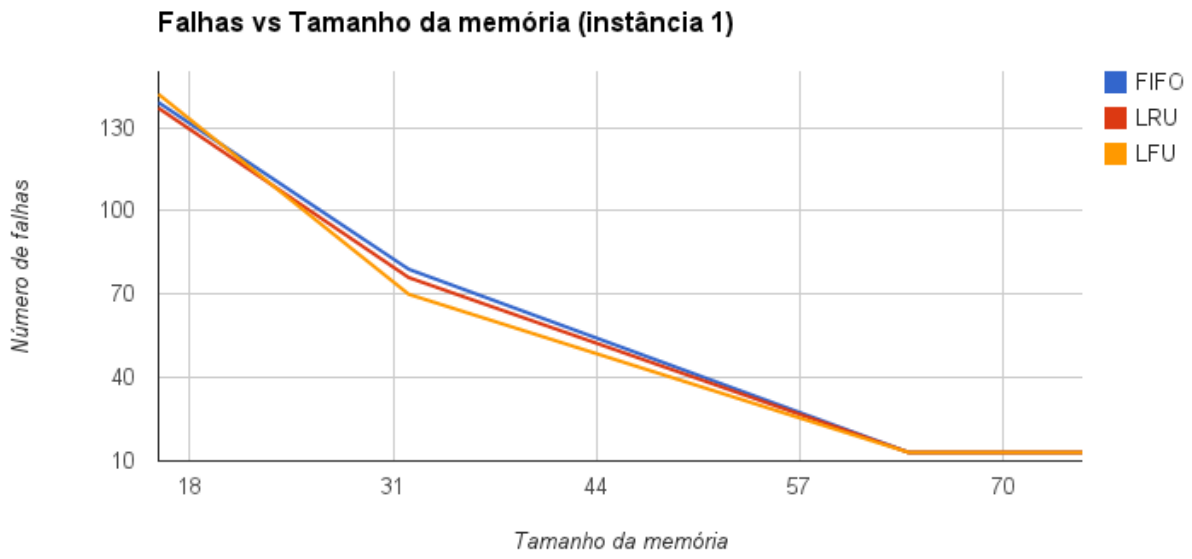
As instâncias um, três e quatro obtiveram os mesmos resultados (conforme apresentado no gráfico abaixo)

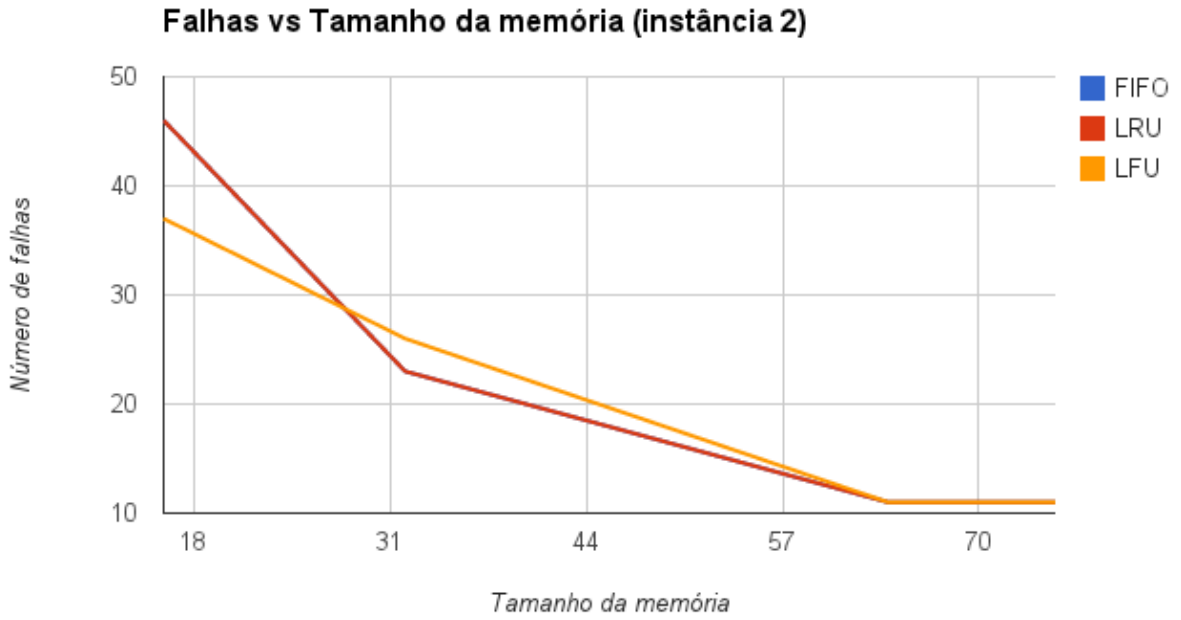


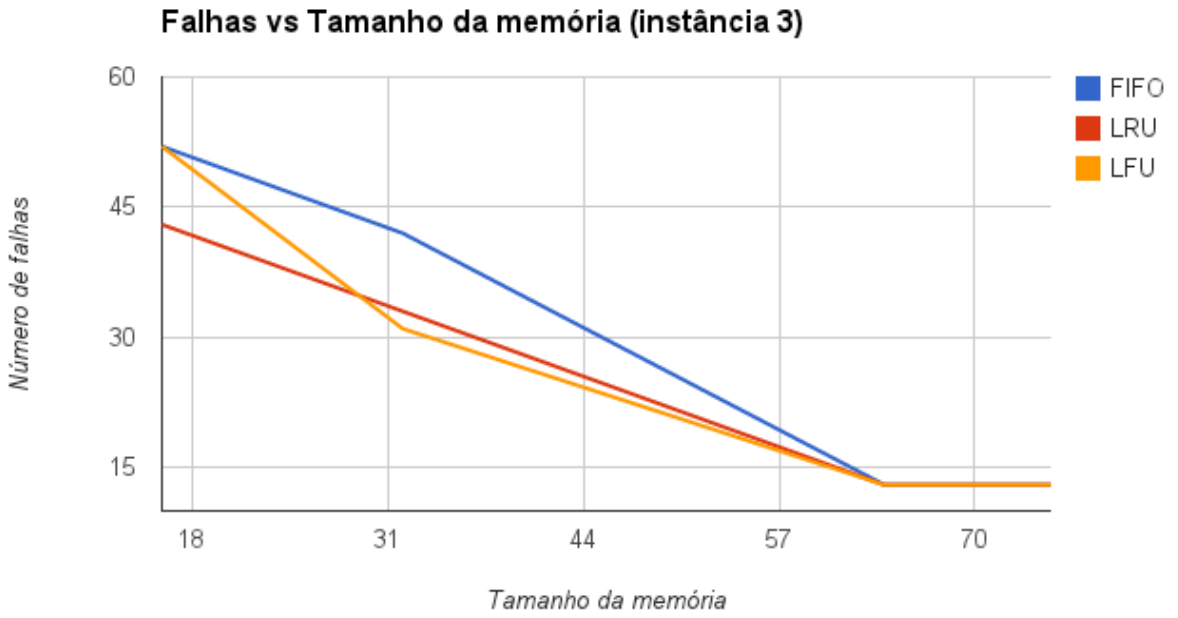
A instância dois está representada no gráfico abaixo.

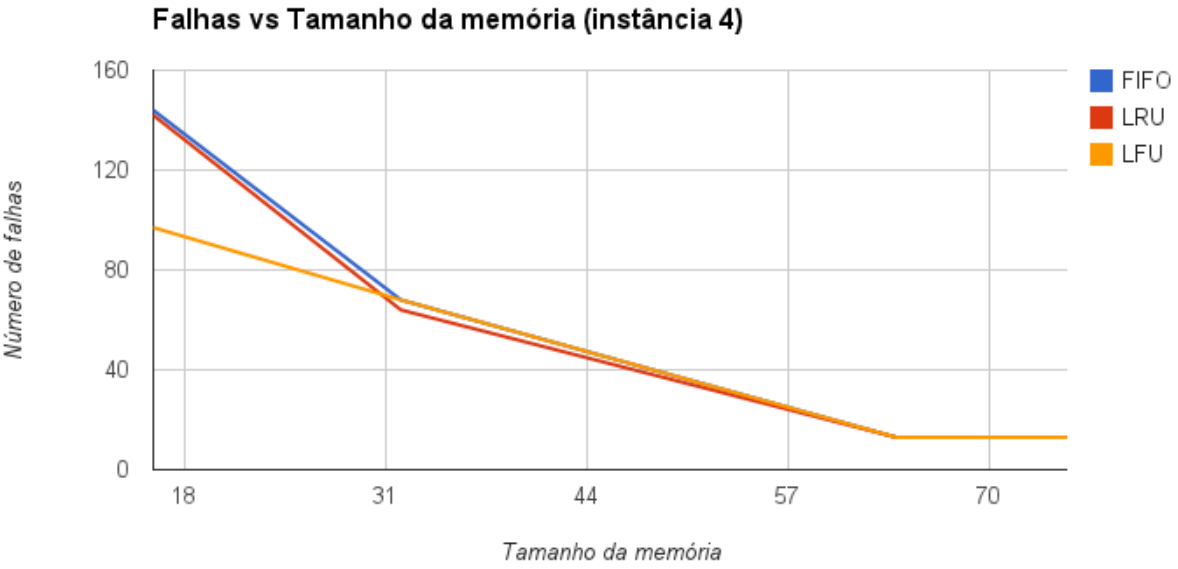


4.3. Gráfico “Tamanho da memória” × “Falhas”









5. Especificação

Todo o desenvolvimento do código foi realizado utilizando uma máquina com as seguintes especificações:

Processador: Intel Core i5 1,7 GHz

Memoria: 4 GB 1333 MHz DDR3

Sistema Operacional: OS X 10.8.2

IDE: Sublime Test 2

GCC: i686-apple-darwin11-llvm-gcc-4.2 (GCC) 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.11.00)

6. Conclusão

Concluímos com esse trabalho que o SMV é muito útil e importante para o funcionamento de um sistema. Ele permite a abstração da memória utilizado por determinado programa, ao mesmo tempo em que otimiza o acesso aos dados deste.

Percebemos também que existe um trade-off em relação ao tamanho da memória versus o tamanho da página (além, é claro, do método de reposição escolhido). Em relação à memória e à página, percebemos que, quanto maior o tamanho da memória ou o tamanho da página, menor será o número de falhas. Isso porque, no caso da memória, quanto maior ela for, mais páginas poderão ser armazenadas. No caso das páginas, quanto maior for, mais bytes caberão acarretando em menos falhas.

Porém, fica claro na análise dos gráficos que não basta sair aumentando o tamanho da memória e das páginas para se ter um algoritmo ideal. No caso da memória, há um custo financeiro adicional para se ter uma memória primária grande. E no caso do tamanho da página, há o custo de acesso ao disco.

Com base nas análises apresentadas acima, concluímos que o melhor valor de memória e tamanho de página para as instâncias utilizadas é 64 bytes (tanto para o tamanho do memória, quanto para o tamanho da página). Com esses valores, teremos apenas uma falha (que é a falha inicial onde a memória ainda está vazia) e essa falha acarreta no carregamento de todos os dados que serão utilizados para a memória.

Ou seja, para os dados analisados, uma memória maior que 64 bytes significaria um custo financeiro desnecessário e uma página maior que 64 bytes significaria em carregamento de dados desnecessários para a memória principal.

Para finalizar, percebemos que as políticas de reposição de memória são diretamente influenciadas pelo perfil da sequência de acesso aos dados requisitados pelo programa.