Centro Universitário FEI

PÓS GRADUAÇÃO – ENGENHARIA ELÉTRICA

MESTRADO – IAAA

INTELIGÊNCIA ARTIFICIAL APLICADA

À AUTOMAÇÃO E ROBÓTICA

Programação Científica– PEL 216

Prof. Dr. Reinaldo Augusto da Costa Bianchi

ALGORTIMOS DE BUSCA

"BUSCA CEGA E BUSCA INFORMADA"

FLÁVIO INFANTI

Matrícula: 118.310-2

AULA 03 – 25/06/2019

1 OBJETIVO

Avançar no aprendizado da disciplina de Programação Científica e se aprofundar nas técnicas referentes à Programação Orientada a Objeto C++, aplicando conceitos como herança, alocação dinâmica de memória, tratamento de exceções, conforme apresentado no livro "Inteligência Artificial" – Russel & Norvig 3ª Edição, consolidando desta forma, os conhecimentos adquiridos durante aula realizada em 25/06/2019, com a implementação de um programa que realize os processos de busca sem informação (largura e profundidade), bem como busca informada (subida de encosta e A estrela).

2 INTRODUÇÃO

Um algoritmo de busca é um algoritmo que percorre um grafo andando pelos arcos de um vértice a outro. Depois de visitar a ponta inicial de um arco, o algoritmo percorre o arco e visita sua ponta final. Cada arco é percorrido no máximo uma vez.

Atualmente existem muitas maneiras de organizar uma busca. Cada estratégia de busca é caracterizada pela ordem em que os vértices são visitados e como estão intimamente relacionadas com os conceitos de distância e caminho mínimo.

Dos algoritmos de busca cega se destacam Breadth-First Search (Busca em Largura) e Depth-First Search (Busca em Profundidade) e dos de busca informada, Hill Climb (Subida de Encosta) e algoritmo A*(Estrela).

2.1. Busca em Largura:

O algoritmo de Busca em Largura é provavelmente o mais simples de todos eles, suas operações consistem em gerar um grafo de espaço de estados pela aplicação de todas as ações/operações à partir do vértice inicial, encontrar as arestas, novos vértices sucessores, gerados pelas ações, ou operações realizadas, e repetir este procedimento sucessivamente aos seus sucessores, até que seja apresentada, caso exista, uma solução para o problema, ou seja aberto todos os elementos do espaço de estados do problema.

Na Busca em Largura ordem de exploração do problema se assemelha à interação de dados da estrutura de dados em Fila, na qual o conjunto de elementos inseridos primeiro é verificado pelas aplicações em primeiro lugar (FIFO – first in first out).

Seus métodos principais são:

- Problema (Inicial): Recebe o problema 'Inicial' e inicializa o estado de espaços.

- Explora (Vértice, ação): explora cada ‘ação’ do estado ‘Vértice’, abre suas Arestas e adiciona no estado de espaços.
- P(Vértice, Ação): função de transição de estados recebe o ‘Vértice’ atual, a ‘Ação’ e retorna o estado futuro Goal (Aresta): Verifica se a Aresta do ‘Vértice’ explorado é a solução do problema.
- PrintPath(): retorna o caminho para o resultado da busca

A Busca em Largura é completa, se existir uma solução, e é ótima, encontra o caminho de menor custo para a solução, se os custos de cada passo são iguais.

Utiliza muita memória por abrir todos os elementos até encontrar o objetivo, e tem sua complexidade de tempo, no pior caso, quando tem que abrir todos os elementos para encontrar a solução, em O (arestas + vértices)

2.2. Busca em Profundidade:

O algoritmo de Busca em Profundidade, assim como no de Busca em Largura, suas operações consistem em gerar um grafo de espaço de estados pela aplicação das ações/operações iniciadas à partir do vértice inicial, a principal diferença essas estratégias de buscas é que o algoritmo de Busca em Profundidade gera as arrestas sucessoras de um vértice somente 1 a cada momento, aplicando as ações/operações individualmente, abrindo ramos, arestas, de uma arvore de sucessores, e repete este procedimento sucessivamente aos seus sucessores, até que seja apresentada, caso exista, uma solução para o problema, ou seja alcançando o ponto mais profundo do ramo que está sendo explorado; nesses casos a busca é retomada pela próxima ação do ramo raiz gerador desse ramo que alcançou o ponto mais fundo do grafo, até que sejam abertos todos os elementos do espaço de estados do problema.

Na Busca em Largura a ordem de exploração do problema se assemelha à interação de dados da estrutura de dados em Pilha, na qual o conjunto de elementos que são inseridos por último é verificado pelas aplicações em primeiro lugar (LIFO - Last in First out).

Seus métodos principais são:

- Problema (Inicial): Recebe o problema ‘Inicial’ e inicializa o estado de espaços.
- Explora (Vértice, ação): explora recursivamente cada ‘Vértice’ e ‘Ação’ do estado ‘Vértice’, abre suas Arestas e adiciona no estado de espaços.
- P(Vértice, Ação): função de transição de estados recebe o ‘Vértice’ atual, a ação e retorna o estado futuro Goal (Aresta): Verifica se a Aresta do ‘Vértice’ explorado é a solução do problema.
- PrintPath(): retorna o caminho para o resultado da busca

A busca em profundidade é completa, se existir uma solução, não é ótima.

Utiliza muita memória por abrir todos os elementos até encontrar o objetivo, e tem sua complexidade de tempo, no pior caso, quando tem que abrir todos os elementos para encontrar a solução, em O (arestas + vértices)

2.3. Subida de encosta:

Há problemas que o caminho percorrido para encontrar a solução é irrelevante e o estado final em si é a solução. A esses problemas é possível utilizar soluções de busca informada, com métodos de aproximação, sem a necessidade de guardar o caminho até a solução, utilizando pouca memória para encontrar soluções razoáveis.

O algoritmo de Subida de Encosta movimenta continuamente no espaço de estados na direção de aproximação do objetivo final, "o pico da subida de uma encosta" (crescimento do valor – uphill), e interrompe sua busca somente quando não existem vizinhos com valores heurísticos melhores (que mais se aproximam do objetivo final) - f (n) = h(n). Esse algoritmo não necessita armazenas na memória a arvore de busca, armazena somente o estado atual, a função de transferência e o objetivo final, o que faz esse algoritmo ter um baixo consumo de memória. O de Encosta é classificado como um algoritmo de busca local gulosa por seguir o melhor estado vizinho sem observar o estado à frente ou anteriores.

Os principais desafios da busca de Subida de Encosta estão relacionados à paralização prematura quando alcança máximos locais, valores de pico local que não são o objetivo final do problema e que tem somente estados inferiores ao redor, e platôs, casos em que os ganhos laterais são inexistentes. Para esses casos de paralização é possível realizar a operação de "têmpera simulada", que consiste na melhora temporária dos estados vizinhos de forma a alcançar novamente o ponto mais baixo e retomar a subida por outro ponto.

No problema do quebra cabeça de 8 peças utilizaremos a função heurística h (n) da distância de Manhattan, que é a soma das distancias das peças do estado atual até a sua posição objetivo. Como as peças não podem mover-se na diagonal, a distância será a contagem de blocos na horizontal e vertical.

Seus métodos principais são:

- Problema (Inicial): Recebe o problema 'Inicial' e inicializa o estado de espaços.
- Explora (Vértice, ação): explora recursivamente cada Vértice e 'Ação' do estado 'Vértice', abre suas Arestas e adiciona no estado de espaços.

- P(Vértice, Ação): função de transição de estados recebe o ‘Vértice’ atual, a ação e retorna o estado futuro Heurístico (vértice): recebe o Vértice adjacente e retorna a estimativa heurística para se alcançar o objetivo do problema.
- Goal (Aresta): Verifica se a Aresta do ‘Vértice’ explorado é a solução do problema.

2.4. Algoritmo A*:

Em casos em que é necessário conhecer o caminho da solução do problema e também há algum conhecimento sobre o problema, que possibilite a criação de uma função heurística h(n) para indicar o melhor caminho a ser seguido, uma estratégica de busca que se destaca é o algoritmo A* que além de buscar a solução, busca o melhor caminho até esta solução.

O Algoritmo A* tem semelhanças com a metodologia de busca do Busca em Largura e com a metodologia de Subida de Encosta, diferenciando-se pela ordem de abertura de ‘Vértices’ sucessores que ao lugar de explorar estes vértices pela ordem de descobrimento “abertura”, o A* ordena a abertura dos sucessores pela ordem da função de custo heurístico f(n), que diferente da metodologia de Subida de Encosta, utiliza em sua função, além da heurística h(n), o custo acumulado g(n) até alcançar o vértice aberto f (n) = g(n) + h(n). Esta estratégia necessita armazenar os vértices abertos, aumentando o uso de memória em comparação com a Subida de Encosta, porém “direciona” para o melhor e mais curto caminho até o resultado final, e uma menor quantidade de vértices. Para heurísticas admissíveis, que não superestima o custo para o objetivo final, A* é completa, encontra o objetivo caso exista, e ótima, encontra o caminho de menor custo.

No problema do quebra cabeça de 8 peças utilizaremos a função heurística h (n) da distância de Manhattan, que é a soma das distancias de peças do estado atual até a sua posição objetivo. Como as peças não podem mover-se na diagonal, a distância será a contagem de blocos na horizontal e vertical.

Seus métodos principais são:

- Problema (Inicial): Recebe o problema ‘Inicial’ e inicializa o estado de espaços.
- Explora (Vértice, Ação): explora recursivamente cada vértice ‘Ação’ do estado ‘Vértice’, abre suas Arestas e adiciona no estado de espaços.
- P(Vértice, Ação): função de transição de estados recebe o ‘Vértice’ atual, a ação e retorna o estado futuro.
- Heurística (vértice, Custo Acumulado): recebe o vértice adjacente e o custo acumulado e retorna a estimativa heurística para se alcançar o objetivo do problema.
- Goal (Aresta): Verifica se a Aresta do ‘Vértice’ explorado é a solução do problema.

3 PROPOSTA DE IMPLEMENTAÇÃO.

O problema das oito peças (8 – puzzle) consiste em um quebra cabeça de oito peças, em uma matriz 3 x 3, e bloco vazio que pode mover-se em suas adjacências. A tarefa é encontrar a sequência de movimentos que transformam o quebra cabeça na posição ordenada.

Estados: a descrição do estado especifica a localização de cada uma das 8 peças e da peça vazia no tabuleiro de 9 posições;

Estado inicial: Qualquer estado pode ser designado como inicial. Para este estudo iremos utilizar como Estado Inicial informado em aula é: 4,1,6,3,2,8,7,x,5.

Ações: é a formulação da movimentação do espaço vazio, pode ser esquerda (0), acima (1), direita (2), abaixo (3).

Modo de transição: dado um estado e uma ação ele retorna o estado sucessor Teste de resultado: verifica se alcanço o resultado final 1,2,3,4,5,6,7,8,x.

Custo do passo: cada passo custa 1 movimento

4	1	6
3	2	8
7	X	5

Estado Inicial

1	2	3
4	5	6
7	8	X

Estado Final

Portanto, partiu-se da ideia de gerar um programa com um “menu” contendo opções que demostrem a utilização da resolução de busca sem informação e informada, tento o seguinte formato:

Figura 1 - Tela do Programa 8 PUZZLE

```
======================================================================

         8 PUZZLE - Resolucao de exercicios por Meio de Busca

======================================================================


   ==> Programa Setado para Rodar com Estado Inicial - 416328795:


    Busca Sem informacao (Cega)
          1 - Busca em Largura (BFS)
          2 - Busca em Profundidade (DFS)

    Busca informacao (Heuristica)
          3 - Busca com Subida de Encosta
          4 - Busca com A*

   ==>    0 - Sair do Programa


   ==> Digite uma opcao desejada? _
```

4 RESULTADOS

Abaixo apresentamos os resultados obtidos da execução do programa com o intuito de realizar os algoritmos de Busca cega e Busca Informmada.

Figura 2 - Opção 1 - Busca em Largura (BFS)

```
==> Digite uma opcao desejada? 1
Resultado para Busca em Largura(BFS)
==> Quantidade de Elementos Explorados = 2092
==> Quantidade de Movimentos para alcancar a solucao = 14
```

Figura 3 - Busca em Profundidade (DFS)

```
==> Digite uma opcao desejada? 2
Resultado para Busca em Profundidade (DFS) : Aguarde um Instante! Programa em Execucao.
==> Quantidade de Elementos Explorados = 107567
==> Quantidade de Movimentos para alcancar a solucao = 96284
```

Figura 4 - Busca de Subida de Encosta (Hill Climbing)

```
==> Digite uma opcao desejada? 3
Resultado para Busca em Subida de Encosta (Hill Climbing)
==> Quantidade de Elementos Explorados = 21716
==> Quantidade de Movimentos para alcancar a solucao = 21716
```

Figura 5 - Busca A* (A Star)

```
==> Digite uma opcao desejada? 4
Resultado para A* ( A Estrela)
==> Quantidade de Elementos Explorados = 45
==> Quantidade de Movimentos para alcancar a solucao = 14
```

A fim de realizarmos comparações entre os principais meios de busca, apresentamos a seguir a Tabela 1 gerada através de simulações feitas pelo programa desenvolvido.

RESOLUÇÃO DE PROBLEMAS POR MEIO DE BUSCA

Estado Inicial (Simulação no Programa)	Largura		Profundidade		Subida de Encosta		A * (star)	
	Qtde Elementos Explorados	Qtde Movimentos para Solução	Qtde Elementos Explorados	Qtde Movimentos para Solução	Qtde Elementos Explorados	Qtde Movimentos para Solução	Qtde Elementos Explorados	Qtde Movimentos para Solução
4163287x5	2.092	14	107.567	96.284	21.716	21.716	45	14
41632875x	1.558	13	110.549	98.023	18.753	18.753	43	13
x23145786	24	5	414	407	5	5	9	5
5623x1478	10.475	17	7.275	7.035	32.201	32.201	68	17
56231x478	8.245	16	127.146	106.876	27.323	27.323	36	16
57x231648	92.762	23	30.063	28.911	22.199	22.199	545	23
38x571426	160.790	27	30.880	29.677	14.131	14.131	3.068	27
1483562x7	144.562	26	132.766	107.882	8.744	8.744	3.231	26

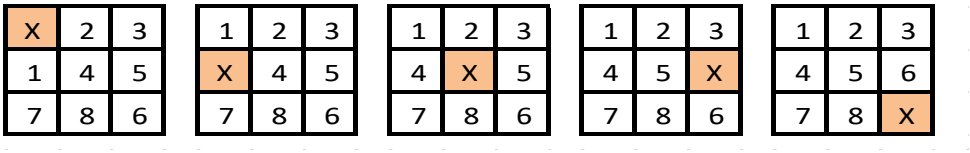
Exercício proposto em aula Verificação de Resultado

Tabela 1 - Tabela Comparativa entre os tipos de Busca

5 CONCLUSÕES

Primeiramente, destacamos que a Busca A* apresentou os melhores desempenhos na solução do 8 Puzzle, acompanhada em seguida pela Busca em Largura que só apresentou um tempo maior devido ao fato de ter um número maior de elementos explorados, porém observamos que para a simulação destacada na cor laranja, a quantidade de elementos e movimentos para Busca em Subida de Encosta teve o mesmo desempenho aos descritos anteriormente, o qual colocamos a sequencia de resolução para o estado inicial: x23145786.

Movimentos entre Estado Inicial e Estado Final



Como principal conclusão podemos afirmar que, embora desenvolver algoritmos utilizando ponteiros seja mais difícil e moroso, ao aplicarmos o conceito de lista encadeadas nos programas esta prática seja a melhor opção quanto à necessidade de alocar memória dinâmica, ou seja, conforme vai usando a memória vai estendendo sua capacidade.

6 REFERÊNCIAS BIBLIOGRAFIA

- RUSSELL, S.; NORVIG, P. – Inteligência Artificial. Elsevier – 3ª edição, 2013.
- CORMEN, T.H.; LEISERSON, C.E.; RIVEST, R.L. e STEIN, C. Introduction to Algorithms, 3ª edição, MIT Press, 2009.
- ZIVIANI, N. Projeto de Algoritmos com Implementações em Java e C++. Thomson, 2007.

ANEXO – ALGORITMO DO PROGRAMA – “BUSCAS”

```
//=== PEL216 - PROGRAMACAO CIENTIFICA
//=== Prof. Dr. Reinaldo Augusto da Costa Bianchi
//=== Aluno: Flavio Infanti nº 118.310-2
//=== AULA 3A e 3B - 25/06/2019
//===
//=== Busca Cega (DFS e BFS)e Busca Informada HC  e A*)
```

```
//=== BIBLIOTECA DE ENTRADA E SAIDA

#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
#include <time.h>
#include <iostream>
#include "AG_Puzzle_lib.h"

using namespace std;


int main(){
   char opcaoTrocaEstadoInicial;
            int exercicio, valor, maxEstados, opcao, TotalAmostras, i;
            clock_t t;
   //time.h para utilizar na medida de tempo do acesso a estrutura de dados
            double time_taken;

   // Exercicio Proposto em Aula
   //exercicio=416328795;
   //exercicio=416328759;
   //exercicio=923145786;
   //exercicio=562319478;
   //exercicio=579231648;
    exercicio=148356297;


   AGENTE *puzzle8;
   AG_DFS *DFS;
   AG_BFS *BFS;
   AG_HC *HC;
   AG_A *A;
   float tempos[4][40];

   //base_amostras = new unitMem<int>;
   unitMem<int> *base_amostras;
   AG_ESTADOS *ESTADOS; // possiveis estados para uso em estatistica

   printf("\n================================================================================\n\n");
   printf( "          8 PUZZLE - Resolucao de exercicios por Meio de Busca            \n" );
   printf("\n================================================================================\n\n");
   printf("");

   printf("\n\n    ==> Programa Setado para Rodar com Estado Inicial - %d: \n",exercicio);
   //printf("\n\n    ==> Deseja apresentar outro Estado Inicial (S/N):");

   //scanf("%s", &opcaoTrocaEstadoInicial);
   //opcaoTrocaEstadoInicial = toupper(opcaoTrocaEstadoInicial);

   //if (opcaoTrocaEstadoInicial=='N')
      //exercicio =416328795;
   //else
      //scanf("Digite o novo Estado Inicial : %d",&exercicio);
      //exercicio =416328795;


while( 1 )
   {
      printf("\n\n     Busca Sem informacao (Cega)\n");
                  printf("          1 - Busca em Largura (BFS)\n");
                  printf("          2 - Busca em Profundidade (DFS)\n\n");
                  printf("     Busca informacao (Heuristica)\n");
                  printf("          3 - Busca com Subida de Encosta\n");
                  printf("          4 - Busca com A*\n");

                  printf("\n    ==>    0 - Sair do Programa\n");

                  printf("\n\n    ==> Digite uma opcao desejada? ");
                  scanf("%d", &opcao);

            switch (opcao){
                              case 1: // Resolucao com BFS - Busca em Largura
                                 printf("\n    Resultado para Busca em Largura(BFS)" );
                                 t = clock();
           BFS=new AG_BFS(1);
           // Polimorfismo com Metodo Virtual puzzle8 obtem metodos do BFS
```

```
                                    puzzle8=BFS;
                                    try{
    puzzle8->expanding_node(exercicio);
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
t = clock() -t;
time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
//printf("|%f|seconds to execute \n",  time_taken);
try{
    puzzle8->PrintPath();
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
puzzle8->~AGENTE();
break;
                                case 2: // Resolucao com DFS - Busca em Largura
printf("\n    Resultado para Busca em Profundidade (DFS)" );
t = clock();
DFS=new AG_DFS(1);
  //    Polimorfismo com Metodo Virtual puzzle8 obtem metodos do DFS
puzzle8=DFS;
try{
    puzzle8->expanding_node(exercicio);
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
t = clock() -t;
time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
//printf("|%f|seconds to execute \n",  time_taken);
try{
    puzzle8->PrintPath();
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
puzzle8->~AGENTE();
                                        break;
                                case 3: // Resolucao com Subida de Encosta
printf("\n    Resultado para Busca em Subida de Encosta (Hill Climbing)" );
                                    t = clock();
                                    HC=new AG_HC(1);
  //    Polimorfismo com Metodo Virtual puzzle8 obtem metodos do HC
puzzle8=HC;
try{
    puzzle8->expanding_node(exercicio);
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
t = clock() -t;
time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
//printf("|%f|seconds to execute \n",  time_taken);
try{
    puzzle8->PrintPath();
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
puzzle8->~AGENTE();
                                        break;
                                case 4: // Resolucao com A*.
                                    printf("\n    Resultado para A* ( A Estrela)" );
                                    t = clock();
                                    A=new AG_A(1);
  //    Polimorfismo com Metodo Virtual puzzle8 obtem metodos do DFS
puzzle8=A;
try{
    puzzle8->expanding_node(exercicio);
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
t = clock() -t;
time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
//printf("|%f|seconds to execute \n",  time_taken);
try{
    puzzle8->PrintPath();
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
```

```
puzzle8->~AGENTE();
                                        break;

                                case 6: // testes para validação do tempo consumido pela interação com a estrutura de dados
//printf("\n\nDigite quantos elementos deseja testar: ");
scanf("%d", &TotalAmostras);
for (valor=0;valor<TotalAmostras;valor++){
   //printf( "\nInicializa Agente BFS Busca em Largura" );
   t = clock();
   BFS=new AG_BFS(1);
  //    Polimorfismo com Metodo Virtual puzzle8 obtem metodos do BFS
   puzzle8=BFS;
   try{
      puzzle8->expanding_node(exercicio);
   }
   catch (int param) { cout << "int exception"; }
   catch (char param) { cout << "char exception"; }
   t = clock() -t;
   time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
   tempos[0][valor]=time_taken;
   //printf("|%f|seconds to execute \n",  time_taken);
   puzzle8->~AGENTE();


   //printf( "\nInicializa Agente DFS Busca em Profundidade" );
   t = clock();
   DFS=new AG_DFS(1);
    //    Polimorfismo com Metodo Virtual puzzle8 obtem metodos do DFS
   puzzle8=DFS;
   try{
      puzzle8->expanding_node(exercicio);
   }
   catch (int param) { cout << "int exception"; }
   catch (char param) { cout << "char exception"; }
   t = clock() -t;
   time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
   tempos[1][valor]=time_taken;
   //printf("|%f|seconds to execute \n",  time_taken);
   puzzle8->~AGENTE();


   //printf( "\nInicializa Agente Subida de Encosta" );
   t = clock();
   HC=new AG_HC(1);
    //    Polimorfismo com Metodo Virtual puzzle8 obtem metodos do HC
   puzzle8=HC;
   try{
      puzzle8->expanding_node(exercicio);
   }
   catch (int param) { cout << "int exception"; }
   catch (char param) { cout << "char exception"; }
   t = clock() -t;
   time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
   tempos[2][valor]=time_taken;
   //printf("|%f|seconds to execute \n",  time_taken);
   puzzle8->~AGENTE();

   //printf( "\nInicializa Agente A*" );
   t = clock();
   A=new AG_A(1);
    //    Polimorfismo com Metodo Virtual puzzle8 obtem metodos do DFS
   puzzle8=A;
   try{
      puzzle8->expanding_node(exercicio);
   }
   catch (int param) { cout << "int exception"; }
   catch (char param) { cout << "char exception"; }
   t = clock() -t;
   time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
   tempos[3][valor]=time_taken;
   //printf("|%f|seconds to execute \n",  time_taken);
   puzzle8->~AGENTE();

}

tempos[0][TotalAmostras]=0;
tempos[1][TotalAmostras]=0;
```

```
            tempos[2][TotalAmostras]=0;
            tempos[3][TotalAmostras]=0;

            for (valor=0;valor<TotalAmostras;valor++){
                  tempos[0][TotalAmostras]=tempos[0][TotalAmostras]+tempos[0][valor]/TotalAmostras;
                  tempos[1][TotalAmostras]=tempos[1][TotalAmostras]+tempos[1][valor]/TotalAmostras;
                  tempos[2][TotalAmostras]=tempos[2][TotalAmostras]+tempos[2][valor]/TotalAmostras;
                  tempos[3][TotalAmostras]=tempos[3][TotalAmostras]+tempos[3][valor]/TotalAmostras;
            }

            getch();
```

```
            break;

         case 0: // desconstruir objetos e sair
            exit(0);
                                    default: printf( "\nDigite uma opção! \n" );
                        }
             }


   getch();
   exit(0);

}
```