Universidade Federal de Santa Catarina

Otimização do processo de armazenamento e recuperação de informações tridimensionais em bancos de dados

Mauricio Schoenfelder

Universidade Federal de Santa Catarina

Departamento de Informática e Estatística

Bacharelado em Ciências da Computação

OTIMIZAÇÃO DO PROCESSO DE ARMAZENAMENTO E RECUPERAÇÃO DE INFORMAÇÕES TRIDIMENSIONAIS EM BANCOS DE DADOS

Autor: Mauricio Schoenfelder
Orientador: D. Sc. A. Jorge Muniz Barreto
Banca Examinadora:
Dr. Mauro Roisenberg
Dr. Antônio Carlos Zimmermann

Florianópolis, fevereiro de 2004

À minha esposa, Rosilene

Aos meus pais, Erci e Teresita

À minha irmã, Tatiana e seu marido, John

Sumário

Resumo

A recuperação de informações em bases de dados cada vez maiores tornou-se um processo crítico que deve ser feito de maneira rápida e confiável. Tais requisitos podem ser atingidos através de técnicas como hash, árvore e indexação em sistemas onde os dados são textuais ou numéricos, porém, com o crescimento da Internet e da capacidade de processamento dos computadores, dados multimídia também estão sendo armazenados, como exemplo, arquivos de imagem, som e vídeo. O armazenamento de elementos desta classe de dados faz com que tais técnicas de estruturação devam ser redefinidas, levando em conta a semântica da informação. Atualmente o projeto SORFACE (Sistema Óptico de Reconhecimento de Faces) trata de identificar se duas faces confrontadas são ou não da mesma pessoa, caracterizando um cenário de verificação, em uma extensão, o sistema deverá ser capaz de realizar a identificação de um indivíduo, ou seja, encontrar a identidade de uma pessoa sem que esta seja declarada. A utilização do método de hashing geométrico e raciocínio baseado em memória permitiu o desenvolvimento de um protótipo de um sistema gerenciador de dados capaz de indexar as faces geradas pelo SORFACE utilizando unicamente seu conteúdo e capaz de aprender com os processos de reconhecimento realizados.

Palavras Chave: banco de dados, hashing, Raciocínio Baseado em Casos.

1 Introdução

Do ponto de vista do gerenciamento de informações, o armazenamento e a recuperação de dados complexos, denominados por alguns autores de multimídia ou não convencionais, trouxe consigo diversas conseqüências.

Sistemas gerenciadores deste tipo de informação devem ser capazes de realizar buscas tendo como chave de procura o próprio dado. Tal fato faz com que as técnicas de estruturação devam ser redefinidas devido às características singulares de cada tipo de informação. Por exemplo, as características que identificam um arquivo de som podem não ser as mesmas que identificam um arquivo de imagem. Normalmente, sistemas de banco de dados comerciais, tratam dados não usuais de modo genérico e efetuam suas consultas segundo dados textuais descritivos.

O desenvolvimento de sistemas de banco de dados que tratem dados não usuais de modo não genérico, normalmente é realizado restringindo o domínio de aplicação do sistema, como exemplo, sistemas de banco de dados geográficos. Tais sistemas restringem seu domínio à dados geográficos ou georeferenciados. Por tratarem de dados complexos, muitos sistemas deste tipo utilizam o modelo orientado a objeto para banco de dados, evitando conversões entre modelos – por exemplo, de um modelo orientado a objeto para relacional.

O projeto SORFACE (Sistema Óptico de Reconhecimento de Faces) trata do reconhecimento de faces humanas a partir de sua forma geométrica 3D. Os

modelos tridimensionais após terem sido captados serão guardados em uma base de dados para uso posterior. Dois cenários são propostos para o processo de reconhecimento, verificação e identificação.

Os dados referentes às faces podem ser vistos como uma classe de dados complexos, e, portanto possuem características próprias que não se aplicam a dados textuais ou numéricos. Tais informações possuem uma estrutura bastante simples, porém, sua semântica é rica e deve ser a principal característica explorada em uma busca. Caso fossem utilizadas técnicas de comparação exaustiva – tal como distância de Hamming - em uma base de dados com muitas faces o tempo de recuperação de uma imagem próxima à imagem dada pode tornar-se proibitivo, quando somado ao tempo de verificação.

Como solução proposta para tal problema foi a utilização de uma técnica de indexação por conteúdo com o principal requisito de não realizar comparações exaustivas. Somado a técnica de indexação, métodos de inteligência artificial (IA) e modelos orientados a objeto possibilitaram uma diminuição significativa dos custos de armazenamento e recuperação dos dados tratados.

2 Objetivos

O principal objetivo deste projeto é a utilização de técnicas de inteligência artificial e de indexação por conteúdo para agilizar o processo de armazenamento e recuperação das informações tridimensionais armazenadas pelo projeto SORFACE.

Mais especificamente, os objetivos são:

- Utilização da técnica de Raciocínio Baseado em Casos juntamente com técnicas de banco de dados;
- Diminuir o tempo de recuperação das informações tridimensionais;
- Desenvolvimento de um pequeno protótipo de uma base de dados que utilize as técnicas já citadas para o armazenamento e recuperação dos dados;
- Publicação de artigos científicos;

3 Metodologia e Estrutura do Relatório

Através dos estudos das técnicas de IA juntamente com as técnicas de banco de dados, foi possível o desenvolvimento de um protótipo de armazenamento e recuperação de dados para as imagens das faces tridimensionais geradas pelo projeto SORFACE.

O relatório foi estruturado de modo que as duas primeiras seções tratarão de caracterizar aspectos gerais de sistemas de banco de dados – SGBD – com especial atenção aos métodos de ordenação e recuperação de dados. Em seguida, será realizada uma descrição do método de Raciocínio Baseado em Casos – RBC – com ênfase na técnica de Memory Based Reasoning – Raciocínio Baseado em Memória.

Por fim, a última seção terá por objetivo tratar da descrição do sistema construído, suas características e resultados obtidos.

4 Sistemas de Gerenciamento de Banco de Dados não-convencionais

Os avanços tecnológicos dos últimos anos geraram uma explosão de informações que se apresentam de forma distribuída. Entretanto faz-se necessário que tais informações sejam armazenadas e que sua manipulação seja eficiente e confiável. Neste contexto os SGBDs tomam papel singular.

Em um primeiro momento os dados eram praticamente em sua totalidade numéricos ou textuais simples, porém, com o aumento da capacidade de processamento dos computadores, formatos de dados até então restritos a aplicações em campos específicos tornaram-se populares e acessíveis, aumentando assim a complexidade do seu gerenciamento. Como exemplo deste tipo de dados, podem ser citadas, imagens, arquivos de áudio e vídeo. Tal tipo de dado é designado por muitos autores de não convencionais.

Um SGBD cuja base de dados contenha informações não convencionais deve levar em consideração as características próprias desta classe de dados, as quais muitas vezes não se verificam em dados simples, como por exemplo, (KALIPSIZ, 2000, p.2):

- ***Falta de estrutura:*** dados não convencionais são carentes de uma estrutura bem definida;
- ***Temporiedade:*** alguns tipos de dados são tempo-dependentes (vídeo, áudio).

Tais dados podem ser divididos em duas classes, (i) contínuos e (ii) discretos. Mídias contínuas são ditas tempo-dependentes, podendo-se citar como exemplo vídeo e áudio. Imagens e gráficos pertencem à segunda classe, pois são tempo-independentes, ou seja, não se alteram no decorrer do tempo (KALIPSIZ, 2000, p.2). As informações armazenadas no SGBD desenvolvido pertencem a segunda classe acima descrita.

Como citado anteriormente, SGBD comerciais possuem um apelo genérico no tratamento do tipo de dado aqui tratado, porém vários tipos de aplicações necessitam um tratamento específico dos dados, como bancos de dados geográficos ou temporais. Um novo campo de aplicação de SGBD é aberto e técnicas anteriormente utilizadas são redefinidas.

Um SGBD não-convencional deve não apenas assegurar os requisitos de um SGBD convencional, como integridade dos dados, tolerância à falhas e controle de concorrência, mas deve também suportar outras características impostas pelo tipo de informação gerenciada, como, por exemplo, suportar consultas sensíveis ao conteúdo (KALIPSIZ, 2000, p.3).

Em uma consulta típica a chave de procura deve coincidir com os dados a serem recuperados. Em um banco de dados não convencional, muitas vezes a chave de procura não coincide exatamente com os dados armazenados. Para que seja possível recuperar tais dados a consulta deve considerar o conteúdo das informações.

Este tipo de consulta - denominada consulta baseada em conteúdo - é de especial interesse, visto que a recuperação das faces armazenadas terá como entrada chaves que não serão totalmente idênticas às que estarão na base de dados, sendo assim, o que determinará quais dados serão retornados pela consulta

será o conteúdo ou a semântica das faces. É importante notar que o conceito de semântica aqui utilizado se refere a forma da face, mais ligado ao significado de como percebemos uma face e não a estrutura interna dos dados.

Para que seja possível armazenar e recuperar informações de uma base de dados de maneira rápida sem comprometer a confiabilidade do processo, são utilizadas técnicas de estruturação e organização dos dados. Tais estruturações podem diminuir significativamente o tempo de acesso aos mesmos. Dentre as técnicas comumente utilizadas estão indexação e tabelas Hash[1].

4.1 Banco de dados Orientado a Objetos

A evolução do hardware permitiu o desenvolvimento de sistemas de grande porte e não-convencionais, especialmente utilizados em áreas científicas e de apoio a decisão. Os tipos de dados não-convencionais, além de terem as características mencionadas anteriormente são caracterizados por terem uma estrutura complexa e de exigirem operações complexas sobre eles para a aquisição de informações.

Em comparação com os sistemas relacionais, dados complexos não possuem domínio atômico, e muitas vezes possuem estrutura não fixa.

Sistemas de Banco de Dados Orientados a Objeto - SGBD OO – utilizam o modelo orientado a objetos na definição do seu modelo de dados. Neste caso,

[1] O nome da técnica não possui tradução coerente para a língua portuguesa, por este fato será utilizado o nome em sua língua original no texto.

dados não são representados por tabelas que representam relações, mas sim por objetos, os quais possuem relações entre si, como no modelo OO.

Assim como no modelo OO, objetos possuem atributos e comportamentos. Em um SGBD OO, há a definição de classes de dados e os dados inseridos serão instâncias destas classes.

No modelo relacional de banco de dados, o que identifica um registro é a chave primária, em um SGBD OO todo objeto necessita de uma identificação única, este requisito é cumprido pelo Identificador de Objeto[2] – IDO. Tal identificador consiste de um código provido pelo sistema e único. Com isto fica garantido que apenas com o IDO do objeto será possível recupera-lo da base de dados.

Os relacionamentos entre objetos dentro do banco são realizados com referências a IDOs. Por exemplo, se um objeto A necessita referenciar um objeto B, este terá em sua estrutura um atributo de referência que conterá o IDO de B.

Um IDO pode ser de dois tipos principais: (i) IDO físico e (ii) IDO lógico. Em (i) é armazenado informações do armazenamento físico do objeto, como volume, lado, setor, como conseqüência há uma melhora na performance de recuperação dos objetos mas por outro lado, a dependência do sistema de armazenamento faz com que o IDO tenha que ser alterado caso sua localização física seja modificada. O segundo tipo tem como principal desvantagem a necessidade de um dicionário de dados e a indexação de IDOs para obter a localização física dos objetos, como vantagem, pode-se armazenar no próprio IDO informações sobre qual classe o objeto pertence ou algum outro que seja relevante.

[2] Em documentos que utilizam a definição em língua inglesa normalmente tal abreviatura aparece como OID

4.2 Indexação

Imaginemos uma situação onde os dicionários, quando desenvolvidos, tivessem as palavras adicionadas de maneira aleatória, sem ordem definida. Caso fosse desejado encontrar o significado de uma determinada palavra, no pior caso seria necessário percorrer todo o dicionário para encontrar o objetivo da consulta. Para tornar a busca mais eficiente as palavras são ordenadas alfabeticamente.

Da mesma maneira que a ordenação das palavras em um dicionário torna uma busca mais fácil e rápida, um índice pretende diminuir os custos de uma consulta em uma base de dados mantendo suas informações ordenadas, excluindo a necessidade de se fazer uma busca seqüencial.

Cada estrutura de índice está associada a uma chave de pesquisa particular, tal elemento é utilizado como base da ordenação, no exemplo anterior as palavras catalogadas são o conjunto de valores chave de pesquisa, visto que a ordenação e feita sobre as mesmas. É válido enfatizar que este conceito de chave não é o mesmo de chaves primárias ou IDOs, sendo que estas identificam de maneira única um registro na base de dados.

Dois tipos de índices são identificados por (KORTH, 1994, p.268):

- ***Índice denso****:* há uma entrada no índice para todo valor de chave diferente. A entrada possui o valor da chave e um ponteiro para o registro;
- ***Índice esparso****:* são criadas entradas no índice apenas para alguns valores de chave.

A utilização de índices densa torna a busca mais rápida, porém há um desperdício de espaço maior do que na utilização de índices esparsos, visto que é necessária uma entrada para todo valor de chave.

Uma das principais desvantagens da utilização de índices seqüenciais é que a performance diminui à medida que o numero de entradas no índice aumenta. Tal deficiência pode se tornar um grande problema, especialmente em sistemas onde inclusões e exclusões são realizadas com muita freqüência.

Para lidar com esta deficiência é possível utilizar a indexação por árvore, que pode ser visto como um índice multinível. A seguir será abordado um tipo de árvore nomeada árvore B^+. Pelo fato de sua eficiência manter-se inalterada mesmo com inclusões e exclusões, este método é largamente utilizado. (KORTH, 1994, p.275).

A figura 1 mostra um exemplo de arvore B^+. Este tipo de arvore é dita balanceada, visto que todas as suas folhas possuem o mesmo nível, ou seja, a quantidade de nós necessários para alcançar qualquer folha a partir da raiz é o mesmo.

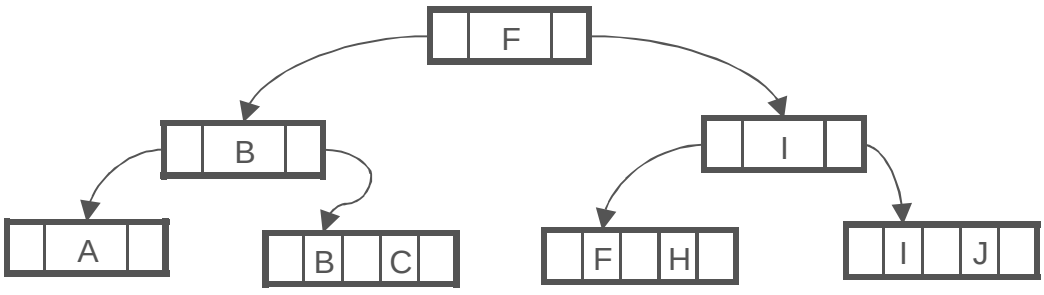

Figura 1 - Árvore B+

Em uma árvore B^+ cada nó deve conter entre *n/2* e *n* filhos, sendo *n* um numero fixo definido para cada árvore em particular. O arquivo de dados e acessado através dos nós folha.

Uma das principais desvantagens da utilização deste tipo de árvore é seu desperdício de armazenamento, visto que os valores de chave dos nos internos se repetem nas folhas da árvore. Entretanto, sua simplicidade de inserção e remoção são grandes atrativos.

Para lidar com a deficiência de armazenamento redundante das árvores B^+, podem ser utilizadas arvores B. Tal categoria de árvores é muito similar ao discutido anteriormente, sua principal diferença esta no fato dos nós internos não se repetirem nas folhas, assim os ponteiros para o arquivo de dados ficam localizados no mesmo nó onde se encontra o valor chave de pesquisa.

Apesar de evitarem a redundância de informação, a estrutura de uma árvore B torna-se mais complexa, assim como seus algoritmos de inserção e remoção. Em árvores B^+, o valor a ser removido sempre estará em um nó folha, porém, o mesmo não acontece em árvores B, com isto, toda vez que um valor chave de pesquisa estiver em um nó interno, faz-se necessário reorganizar a estrutura interna da árvore. No caso anterior, nós internos não precisam ser necessariamente removidos, pois não contém o valor real da chave de pesquisa, são apenas guias que indicam o caminho a ser seguido até as folhas.

Por suas vantagens serem muitas vezes compensadas por suas desvantagens, arvores B são menos utilizadas do que árvores B^+.

Técnicas de indexação tornam o acesso aos dados mais eficiente, porém dependem de uma estrutura de índice organizada para realizar uma busca. O tempo de acesso a tal estrutura, somado ao tempo de manutenção da mesma pode se tornar proibitivo, principalmente em sistemas críticos.

4.3 Hashing

Estruturas de índice podem se tornar um problema, pois aumentam a demanda de acesso a disco, aumentando assim o tempo de resposta de uma busca particular. As técnicas apresentadas nesta seção visam minimizar a utilização destas estruturas, mapeando os dados à sua localização em uma entrada[3] de maneira direta.

De modo semelhante a um índice, um hash é organizado sobre uma chave de pesquisa, entretanto, o que difere uma técnica da outra é a maneira como utiliza tal chave.

Segundo (BRAUER, 1984, p. 118) um hash é composto por um array T[0..m-1] e uma função hash h : U $\rightarrow$ [0..m-1], onde U é o universo. Seja um conjunto S de valores chave de pesquisa, um elemento x ? S é armazenado em T[h(x)] (figura 2).

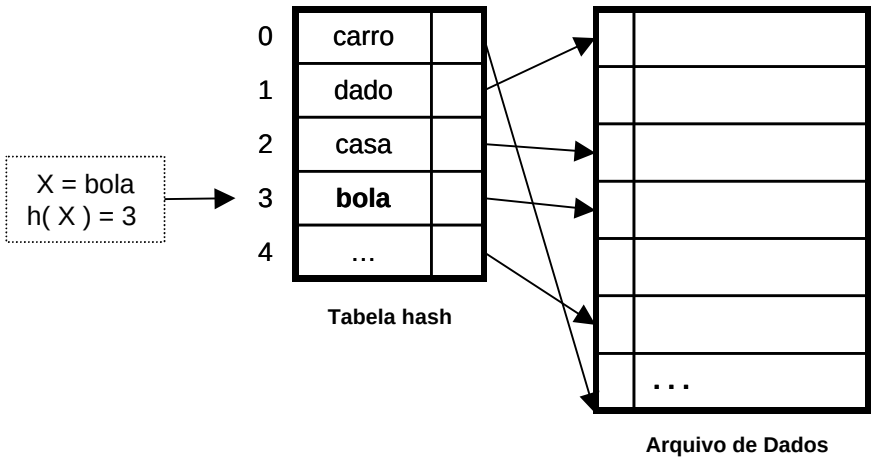

Figura 2 - Função e tabela hash

[3] Em várias referências, entrada de uma tabela hash é citada como *bucket*

Cada endereço do array citado pode ser implementado como sendo um entrada, semelhante aos utilizados nas seções anteriores.

Para que o desempenho de um hash não seja degradado no decorrer de inserções e remoções é desejável que se verifique as seguintes propriedades no mesmo (KORTH, 1994, p.285):

- *Distribuição uniforme:* a cada entrada será designado o mesmo número de valores chave de pesquisa;
- *Distribuição aleatória:* no caso médio, cada entrada terá o mesmo número de valores designados.

Há duas maneiras de um hashing ser implementado, de maneira estática e dinâmica. A principal diferença entre as duas é que na primeira o tamanho da tabela hash é definido quando o hash é implementado, enquanto no segundo a mesma tabela cresce proporcional à demanda de espaço. Estas duas implementações serão brevemente discutidas a seguir.

4.3.1 Hashing Estático

Como descrito anteriormente, a principal característica de um hash estático é o fato de sua tabela de hash ser projetada com um tamanho fixo pré-definido. A função hash, neste caso, será definida também de maneira estática, ou seja, será fixa para o número de entradas utilizadas.

A principal vantagem desta implementação é a simplicidade dos seus algoritmos de inserção e remoção de elementos.

Como desvantagem podemos citar a utilização ineficiente de espaço, pois muitas entradas são alocadas e muitas vezes não são utilizadas.

Para que o sistema mantenha seu desempenho, é importante que o hash seja reorganizado periodicamente, realizando ajustes no tamanho da sua tabela, bem como na função utilizada. Tal manutenção necessita que todo o valor chave de pesquisa armazenado na antiga tabela passe por um processo de re-hashing, o que muitas vezes toma um tempo significativo e não disponível.

A utilização de hashing estático deverá ser feita com cautela, apesar de ter várias desvantagens, é simples e rápido, sendo uma boa opção em casos onde o conjunto de valores chave de pesquisa é bastante conhecido e de tamanho reduzido.

4.3.2 Hashing Dinâmico

O fato de se ter uma tabela de hash de tamanho fixo e o alto custo de redimensionamento da mesma torna, muitas vezes, a utilização de hashing estático inconveniente. Para lidar com estes problemas existe a alternativa do método de hashing dinâmico.

O diferencial desta técnica é que a tabela de hash aumenta e diminui à medida que forem inseridos e removidos valores do hash. Para isto, a função hash também é adaptada para mapear mais ou menos valores.

Hashing dinâmico possui uma característica que a cada nova entrada adicionada, apenas uma reorganização local é realizada, jamais uma reorganização global da tabela de hash.

Há diversas implementações desta classe de hashing. Em (LARSON, 1998) e (HEDRICK, 1990) são apresentadas e avaliadas algumas implementações, como hashing linear[4].

4.3.2.1 Hashing Linear

A técnica de hash linear foi desenvolvida em 1980 por W. Litwin.

A figura 3 mostra um exemplo de hash linear. Neste caso $h(x)_0 = x \bmod 3$. A medida que novas entradas são necessárias, as entradas apontadas por *p* são divididas e uma nova entrada é acrescentada ao final da tabela. Quando o número total de entradas dobra (6 neste caso) a função é alterada, neste caso $h(x)_0 = x \bmod 6$.

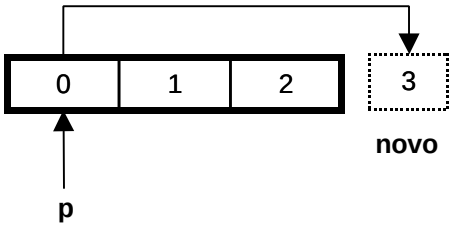

Figura 3 - Hashing Linear

[4] Linear Hashing

Na figura 3 quando se faz necessário a divisão, os registros da entrada partida são realocadas à nova entrada utilizando a próxima função hash a ser utilizada ($h(x)_1$= x mod 6).

A cálculo do endereço de um determinado registro pode ser realizado da seguinte maneira: primeiro calcula-se o endereço com a função atual (x mod 3), caso a entrada alvo seja anterior ao ponteiro *p*, então o mesmo já foi dividido, então será necessário recalcular o endereço utilizando a próxima função hash (x mod 6).

No momento em que o número de entradas dobra, a variável *n* da função hash atual e próxima também dobra, neste caso $h(x)_0$= x mod 6 , $h(x)_1$= x mod 12.

O momento de se dividir uma entrada é determinado por um limite *k* calculado pela razão entre o número total de registros pelo número de entradas.

A implementação deste tipo de hash pode necessita de um array que deverá ser capaz de crescer e diminuir a medida que for necessário. Para isto (LARSON, 1998, p.3) sugere a utilização de uma estrutura de dois níveis, pois são poucas as linguagens de programação que possuem suporte a arrays dinâmicos.

4.4 Hashing Baseado em Conteúdo

Em muitos casos, dados multimídia são indexados utilizando como chave de pesquisa alguma descrição textual, porém, em sistemas que tratam dados não-convencionais não é raro que os mesmos sejam carentes de descrição. Quando isto acontece faz-se necessário métodos que utilizem como chave de pesquisa o próprio conteúdo do dado.

Técnicas de indexação baseadas em conteúdo normalmente são dependentes da extração de características, pré-determinadas ou não, do objeto a ser indexado. Tal processo de extração de características normalmente consome muito tempo de processamento. Caso a técnica exija um número demasiado de acessos a disco, poderá ocorrer que o tempo total de processo se torne proibitivo.

Quando o objetivo principal da utilização de uma técnica de indexação por conteúdo for de otimização de consultas puramente baseadas em conteúdo, poderá ocorrer que métodos de indexação seqüenciais e árvores não sejam uma boa escolha, visto que estes exigem a utilização de estruturas adicionais em sua implementação bem como um número maior de acessos a memória secundária.

A solução mais otimizada seria a utilização de hashing baseados em conteúdo. Três principais abordagens foram estudadas, são elas:

- ***Contorno:*** Algumas técnicas como apresentado em (MORIS, 2001) baseiam-se no contorno do objeto indexado, para tal, primeiramente é realizada a extração do contorno do mesmo, este contorno é utilizado na comparação com os demais objetos já armazenados;
- ***Recipientes:*** Métodos recipientes tem como característica alvo as extremidades do objeto, com isto, é calculado o menor cubo ou esfera que o contenha, este cubo ou esfera é então utilizado para indexar o objeto.
- ***Geométricas:*** Tais técnicas capturam características variadas de um objeto e as representa geometricamente, relações são extraídas desta representação e aproveitadas para efetivamente indexar o objeto. A técnica apresentada a seguir se insere nesta classificação.

4.4.1 Hashing Geométrico

O método de Hashing Geométrico apresentado em (WOLFSON, 1997), originalmente foi desenvolvido para a área de visão computacional e utilizado para armazenar e recuperar objetos segundo características geométricas, porém, é possível que seja utilizada em outras áreas.

As características capturadas são processadas para formar um modelo do objeto a ser indexado, tal modelo serve de base para a definição das entradas relativas ao objeto na tabela hash.

A técnica permite que dados mal-comportados sejam indexados e objetos com partes obscuras sejam reconhecidos, características que a tornaram a mais adequada no tratamento do problema proposto, como será explicitado mais adiante.

O método é constituído de duas fases distintas denominadas *pré-processamento* e *reconhecimento*. No pré-processamento, o objeto é analisado e o seu modelo é criado, permitindo que o dado seja armazenado na base de dados, na fase de reconhecimento, verifica-se a ocorrência de um objeto de entrada na base de dados.

4.4.1.1 Pré-processamento

O objetivo principal desta fase é criar um modelo de um objeto de entrada para armazena-lo na base de dados. Todo o processo é composto de várias ações, são elas:

i. Características são extraídas do objeto de entrada, tais características são definidas de acordo com cada aplicação da técnica, podendo ser arestas, curvas ou mesmo superfícies.

ii. Os pontos característica capturados são então mapeados para um sistema de coordenadas O_{xy} (figura 4a), formando assim o conjunto P de pontos características.

iii. A seguir, um par ordenado de pontos $(p_1,p_2) \in P$ é escolhido, constituindo o vetor $\overrightarrow{p_1p_2}$, denominado base característica.

iv. O conjunto P é então escalonado de modo que $|\overrightarrow{p_1p_2}|$ seja 1 em O_{xy} (figura 4b).

v. P é rotacionado e transladado para que o ponto médio de $\overrightarrow{p_1p_2}$ coincida com a origem de O_{xy} e a direção do vetor seja paralela ao sentido positivo do eixo x (figura 4c).

vi. Assim, teremos um modelo do objeto com base (p_1,p_2), representado por (M,(p_1,p_2)).

vii. O modelo é então armazenado na base de dados de forma que para todo ponto *p* de *P* não pertencente à base *(p_1,p_2)*, *h(p)* recebe *(M,(p_1,p_2))*, sendo *h(p)* a função hash adotada.

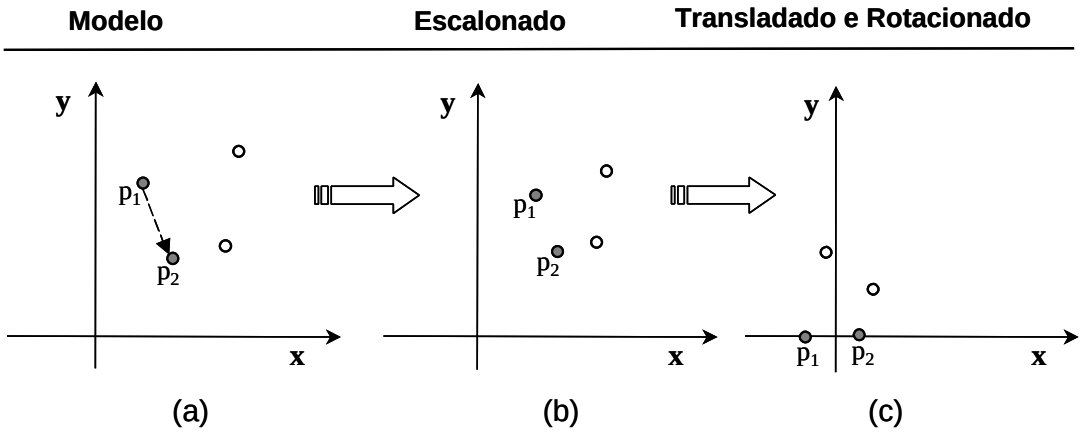

Figura 4 - Pré processamento

A função hash é definida da seguinte forma: a partir da base $\overrightarrow{p_1p_2}$ é possível definir dois vetores $p_x^s = (p_2 - p_1)$ e $p_y^s = Rot_{90}(p_x^s)$, os quais formam uma base ortonormal do sistema de coordenadas O_{xy}. previamente definido. Desta forma, podemos escrever cada ponto *p* do modelo como combinação linear de p_x^s e p_y^s segundo a equação $p - p_0^s = mp_x^s + up_y^s$ (WOLFSON, 1997, p.4).

As constantes m e u são invariáveis, mesmo após as transformações executadas no modelo e sua determinação indicará a entrada do ponto *p.* Assim, cada par (m, u) definirá uma entrada na tabela de hash, nesta entrada será inserida a denominação do modelo (M,(a,b)).

O método foi desenvolvido com o intuito de reconhecer objetos que tenham alguma parte oclusa, para tal todos os passos a partir de (iii) deverão ser repetidos para todos os pares ordenados de *P*.

Sem isto, quando alguma característica do objeto que fizesse parte de uma base não fosse visível, não seria possível encontrar a base característica do objeto, e por conseqüência o mesmo não seria reconhecido, ou seria reconhecido de maneira errada.

4.4.1.2 Reconhecimento

Podemos definir a fase de pré-processamento como o processo de análise e inserção de objetos na base de dados enquanto a fase de reconhecimento recupera objetos da mesma. Os passos a serem executados são os seguintes:

i. Primeiramente o sistema recebe um objeto não processado e então aplica os passos (i) a (vi), descritos na seção anterior, sobre o mesmo apenas uma vez.
ii. Para cada ponto característica do novo modelo o par (m,u) é determinado e sua respectiva entrada é acessado.
iii. Cada registro *(M,(a,b))* na entrada acessada recebe um voto.
iv. Ao final os modelos que mais receberam votos são então retornados.
v. Os vários modelos recuperados são interpretados como possíveis candidatos ao objeto de entrada, por este fato, uma fase adicional de

reconhecimento mais refinada é necessária para, enfim, ser determinado qual candidato corresponde à chave de pesquisa.

O método de hashing geométrico pode ser estendido para um sistema de coordenadas 3D, no qual ao invés de determinarmos (m,u) devemos determinar (m,u,j). A implementação da técnica em um sistema paralelo é relativamente simples e aumenta de modo significativo sua performance (WOLFSON, 1997, p.2).

5 Raciocínio Baseado em Casos

Para explicar o que é Raciocínio Baseado em Casos (ABEL, 1998, p.1) toma como exemplo a formação de um profissional de medicina. Quando estudante, este aprende como o corpo humano é composto, como funciona e como acontecem as doenças. Tal conhecimento é dito genérico da área, ou seja, um conjunto de regras gerais de funcionamento do corpo humano.

Com o tempo, o aluno irá se deparar com vários casos diferentes e guarda os tratamentos aplicados. Este será o conhecimento específico da área, exemplificado através de exemplos de tratamentos realizados com sucesso. É esperado que, quanto mais pacientes o aluno tratar com sucesso, maior será o seu desempenho na solução de novos problemas.

Um sistema de Raciocínio Baseado em Casos (RBC) funciona de maneira bastante similar à descrita anteriormente, a solução de um novo problema é realizada baseando-se na experiência anteriormente adquirida.

A cada novo problema resolvido um novo *caso* é formado, sendo este uma descrição do problema, a solução adotada e o sucesso ou fracasso na solução do mesmo.

Os diversos casos já adquiridos são guardados em uma base de dados e indexados segundo seus atributos (descrição, solução, sucesso/fracasso) para agilizar a sua recuperação. O sistema recupera o casos que mais se assemelha ao problema apresentado, modifica-o para ajustar-se ao problema e apresentar a

solução mais adequada. O novo caso modificado é apreendido pelo sistema e passa a fazer parte do banco de casos disponível (ABEL,1998, p.3).

Um questionamento bastante comum é o fato de sistemas de bancos de dados também serem capazes de armazenar um grande volume de dados e recuperá-los segundo uma entrada de maneira rápida. O grande diferencial é que problemas do mundo real são mais complexos que uma simples chave de entrada, desta forma, um problema dificilmente irá se apresentar exatamente da mesma maneira que outro problema já armazenado no bando de dados. Bancos de dados são muito eficientes em achar problemas iguais, mas são pouco eficazes em realizar buscas por aproximação. Assim sistemas RBC possuem uma vantagem em domínios onde a solução de problemas demanda conhecimento.

De maneira geral, um sistema RBC possui o seguinte ciclo (BARTSCH, 1996, p.2):

i. *Recuperação* dos casos mais similares;
ii. Reuso das informações e conhecimento dos casos para a solução do problema;
iii. Revisão da solução proposta;
iv. Retenção do novo caso solucionado, com informações de sucesso fracasso e até mesmo possíveis modificações para uma melhor solução.

Cada um dos ciclos acima citados é melhor detalhado em (AAMODT, 1999).

Em (AAMODT, 1999, p.5) é realizada uma classificação dos diversos tipos de sistemas RBC identificados, são eles:

- *Raciocínio Baseado em exemplos:* um conceito é definido de maneira extensiva pelos seus diversos casos;

- *Raciocínio baseado em instância:* pode ser visto como uma especialização do caso anterior para a utilização em sistemas onde o conhecimento do domínio é pobre;
- *Raciocínio baseado em memória:* a base de dados dos casos é vista como uma grande memória e o processo de raciocínio é a procura dos casos nesta memória;
- *Raciocínio baseado em casos:* um sistema capaz de modificar um caso já adquirido e resolver novos problemas utilizando um conhecimento global do domínio. Tais problemas devem pertencer ao mesmo domínio;
- *Raciocínio baseado em analogia:* resolve problemas de maneira similar aos sistemas descritos anteriormente, porém pertencentes à domínios diferentes.

Um dos principais problemas em sistemas RBC é tornar a recuperação dos casos eficiente, visto que a mesma exige muitas vezes a busca exaustiva na base de casos, para tal, várias abordagens podem ser utilizadas, como método de recuperação, podem ser utilizados os descritos nas seções anteriores sobre sistemas de bancos de dados.

Os métodos estudados com maior atenção foram *raciocínio baseado em instância* e *raciocínio baseado em memória*. No primeiro as diversas instâncias de um conceito, o definem, porém, muitas referências citavam o método como sendo ruim e pouco exato, levando muitas vezes a solução de um problema de maneira errada.

No método de raciocínio baseado em memória utiliza-se principalmente da técnica de memória dinâmica proposta por Roger C. Schank, neste modelo de memória, a cada nova experiência a memória é alterada, ou seja, a nova experiência é aprendida. Uma memória deverá ser capaz de lembrar uma experiência com poucos dados sobre a mesma (SCHANK, 1988, p.2).

Uma das principais características desta técnica é a utilização maciça da memória, sendo a descrição do domínio muito pobre, desta forma não se faz necessário um conjunto muito grande de regras que definam o domínio.

Alguns sistemas construídos utilizando raciocínio baseado em memória demonstraram grande desempenho, como mostrado em (STANFILL, 1986, p.4) e (STANFILL, 1988, p. 2). Um dos mais populares é o sistema *MBRTalk*, que tinha como problema a pronuncia correta de palavras em inglês. Em um teste, o sistema conseguiu acertar a pronuncia de 86 palavras em 100. O que demonstra o bom potencial da técnica.

6 O Sistema

O sistema tem como principal objetivo gerenciar uma base de dados onde as faces geradas pelo sistema SORFACE deverão ser armazenadas. O fator crítico do sistema é o modo como normalmente as consultas serão realizadas, apenas com faces, sem descrições textuais.

O que diferencia este SGBD dos demais é o fato de levar em consideração a semântica de seus dados não textuais (faces), tornando possível que as buscas sejam realizadas tendo como chave de entrada somente uma face.

Como requisito principal do sistema é possível citar a necessidade de desenvolver processos simples permitindo uma maior velocidade na execução das operações de armazenamento e recuperação. Com isto fica excluída a possibilidade de utilizar técnicas de comparação exaustivas para definir o dado armazenado que tenha maior grau de similaridade com o dado de entrada.

As próximas seções tratarão da descrição do projeto SORFACE e do sistema desenvolvido durante esta pesquisa.

6.1 Sistema Óptico de Reconhecimento de Faces

O projeto SORFACE tem por objetivo gerar modelos tridimensionais de faces humanas e tornar possível o seu reconhecimento. A captura se inicia iluminando a face com um sistema de luz estruturada (figura 5) que projeta franjas

claras e escuras sobre a mesma. A imagem do rosto iluminado é então capturada por uma câmera. Após o processamento da imagem um modelo tridimensional da face é gerado. Tal modelo constitui-se de uma matriz onde cada célula corresponde à altura de um ponto na superfície da face. Tal técnica é apresentada em (ZIMMERMANN , 2003).

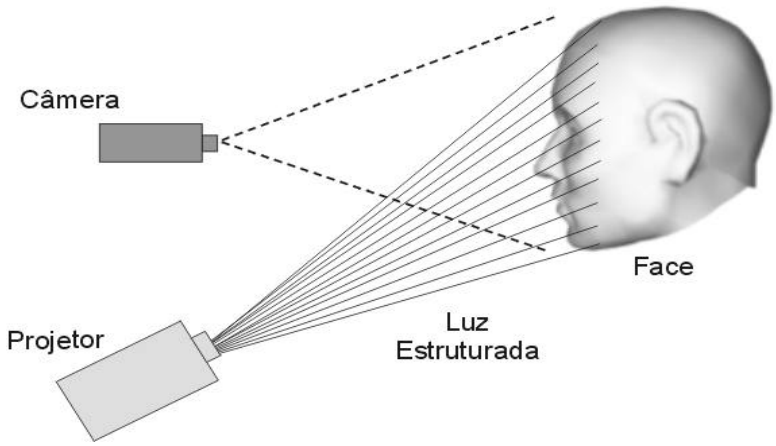

Figura 5-Processo de captura

Dois cenários são propostos pelo projeto, o primeiro é o de verificação, onde o indivíduo declara sua possível identidade e o sistema tem como função confirmar ou refutar tal declaração, este caso foi objeto de estudo de (MARIN, 2003) e foi tratado com técnicas de IA conexionista e evolucionária.

O segundo cenário consiste em identificar um indivíduo capturado pela câmera sem nenhuma declaração realizada. Para tal, o sistema objeto desta pesquisa fica responsável por buscar dentre os indivíduos armazenados na base de dados, aqueles que possuem maior similaridade com a face de entrada. Em uma segunda etapa, o sistema de verificação já desenvolvido é acionado e uma comparação mais refinada é realizada para finalmente a real identidade ser apontada.

6.2 Concepção do Sistema

O modelo do sistema foi concebido tendo como inspiração o processo de reconhecimento humano. Podemos descrever tal processo pela seguinte sucessão de ações:

i. Ao encontrarmos alguém, buscamos ocorrências da pessoa dentre os diversos outros conhecidos;
ii. Possíveis candidatos são escolhidos e um pré-reconhecimento é realizado (sistema proposto);
iii. Os candidatos passam por uma avaliação mais refinada e a decisão da identidade correta é tomada (SORFACE).

Após isto, conseguimos lembrar o nome da pessoa bem como dados mais específicos como idade, índole. Quanto maior a interação ou mesmo a convivência com tal indivíduo, maior será a quantidade de modelos mentais da pessoa apreendidos, e mais rápido será o reconhecimento, pois mais refinada será a descrição de tal pessoa, resultando em um conjunto de possíveis indivíduos com um número menor de identidades incorretas. Por conseqüência, menor será o número de verificações inúteis, o que diminui o tempo de avaliação.

Uma justificativa para tal modelo é o fato de quando não conhecemos muito bem o indivíduo - poucas ocorrências de modelos mentais apreendidos - caso não façamos uma avaliação mais detalhada acabamos por confundi-lo com outra pessoa

- escolha do candidato errado, visto que o conjunto de candidatos é muito heterogêneo.

É válido esclarecer que os modelos mentais reais são muito mais complexos, abrangendo não somente aspectos físicos - formato da face, altura - como comportamentais - modo de agir, falar. O que se pretende fazer é limitar tais modelos às características faciais de cada indivíduo, e a partir destas buscar outras informações.

6.2.1 Arquitetura do Sistema

A arquitetura básica do sistema é constituída de três elementos principais (figura 7), são eles:

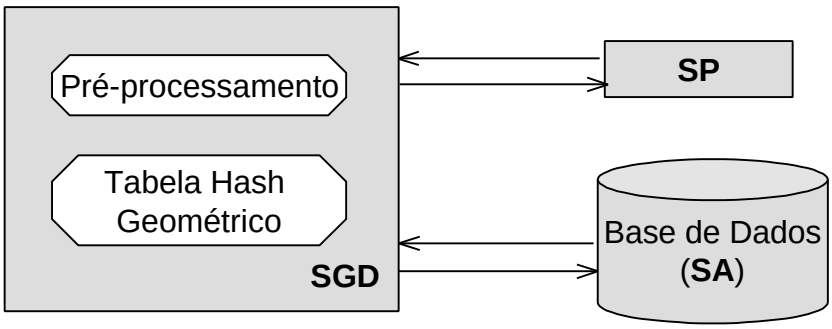

Figura 6 - Arquitetura Básica do Sistema

- ***Sistema Principal (SP)****:* recebe os dados das faces oriundas do sistema SORFACE e as requisições externas. É responsável por decidir se há alguma ocorrência na base de dados que coincida com a face de entrada.
- ***Sistema Gerenciador de Dados (SGD):*** recebe solicitações do SP - retornar possíveis candidatos, incluir face de indivíduo - as processa e então retorna os resultados obtidos. A técnica principal utilizada é a de Hashing Geométrico (seção 4.4.1), devido a sua adequação ao tipo de dado tratado - mal comportado - e ao fato de ter sua expansão para o espaço tridimensional relativamente simples. É responsável pela organização dos dados e provê a visão lógica dos mesmos ao SP. É a interface entre o SP e o sistema de armazenamento.
- ***Sistema de armazenamento (SA):*** tem como função manter a organização física dos dados, ou seja, manter os dados corretamente gravados. Também é de sua competência prover a transição dos dados gravados para a memória principal, e desta para o disco rígido, utilizando para isto, um dicionário de dados.

Assim como em todo SGBD, cada elemento do protótipo possui uma visão dos dados, para o mundo externo a base de dados é um conjunto de informações sobre determinados indivíduos conhecidos.

Para o SP a base de dados é um conjunto de objetos *Indivíduos*, com suas informações organizadas e estruturadas, sendo o SGD o responsável por armazená-los e recuperá-los. Neste nível e no anterior, não há necessidade de se conhecer o

modo como os dados são armazenados ou mesmo como são estruturados, compete apenas ao SP conhecer quais os comportamentos dos indivíduos armazenados.

Ao nível do SGD, objetos *ModeloFace* são armazenados segundo a técnica de Hashing Geométrico e para tal é necessário que o mesmo conheça as características e métodos dos mesmos.

Finalmente, ao SA é apenas necessário ter o conhecimento dos comportamentos relativos a persistência dos dados, as informações sobre os indivíduos são apenas bytes em arquivos determinados, e cabe ao SA prover o acesso eficiente e confiável aos mesmos.

Pode ser observado que o sistema não será do tipo relacional, mas sim orientado a objeto, assim, objetos complexos terão seus identificadores únicos.

6.2.2 Sistema Gerenciador de Dados

Tal subsistema recebe requisições do SP e as processa utilizando os seguintes componentes: pré-processamento e tabela de Hash Geométrico.

Ao receber uma requisição, se necessário um pré-processamento é realizado sobre a face de entrada e um modelo é construído. Somente após o pré-processamento será possível executar alguma requisição - armazenamento, recuperação.

Os modelos construídos irão definir a identidade de um indivíduo, portanto, estarão ligados à um indivíduo, o qual é o objeto principal armazenado. Tais modelos servirão de base para a indexação e aprendizado do sistema.

Fazendo uma comparação com o método de Hashing Geométrico, é possível dizer que o componente pré-processamento executa as tarefas da fase do processo de mesmo nome e a tabela de Hash Geométrico é encarregada de executar as tarefas da fase de reconhecimento, porém não se restringem apenas aos processos do método. A seguir, uma descrição detalhada de cada um dos elementos.

6.2.2.1 Pré-processamento

O principal objetivo desta unidade é construir o modelo da face de entrada. Para o SGD, o pré-processamento se assemelha a um tradutor que recebe faces puras originadas do sistema SORFACE e as traduz em um modelo inteligível.

Como descrito na seção 4.4 a primeira tarefa do pré-processamento é a extração das características.

Primeiramente, foi necessário definir o que seria uma característica de uma face, para tanto várias abordagens foram adotadas. Uma delas seria a busca de elementos comuns a todas as faces como, maçã do rosto, sobrancelhas, etc. Todas as tentativas foram frustradas. O que foi percebido é que as faces capturadas constituíam um tipo de dado mal comportado, principalmente pelos ruídos gerados no processo de aquisição. Caso fossem desenvolvidos algoritmos capazes de capturar tais características muito provavelmente estes seriam caros computacionalmente, descumprindo os requisitos do sistema.

A solução surgiu da observação do modo como percebemos a face de uma pessoa. Cada pessoa possui saliências, curvas em sua face, assemelhando-se a

uma superfície montanhosa. Dependendo da raça ou etnia, a formação óssea delineia uma quantidade maior ou menos de saliências. O conjunto destas deformidades nos dá o formato geral da face de uma pessoa.

Saliências acentuadas significam regiões onde a variância de altura - em termos da matriz gerada pelo SORFACE - de pontos próximos é alta, enquanto saliências tênues, constituem uma variância de altura menor. Como exemplo, a ponta do nariz, nesta região há uma grande variância na altura dos pontos. Com isto, pontos de saliência mais tênue foram definidos como características válidas.

Do ponto de vista da matriz de dados, o que define o processo de captura das saliências é o cálculo da média das variações da altura dos pontos em torno de cada célula (figura 7).

2	2	2	2	2
2	3	3	2	1
2	3	**4**	2	1
2	3	2	2	1
2	2	1	1	1

Variância 2,0833

Figura 7 - Variância do valor central 4

Ao realizar tal processamento sobre as faces foi percebido que pontos característica que possuíam uma variância menor que um certo limiar se repetiam nas diversas amostras faciais de um mesmo indivíduo, ou seja, utilizando a variância dos pontos, cada indivíduo possuía um conjunto de pontos características que de certa forma o determinava. A figura 8 mostra os valores da matriz de variância de quatro diferentes amostras da face de um mesmo indivíduo representadas em um gráfico.

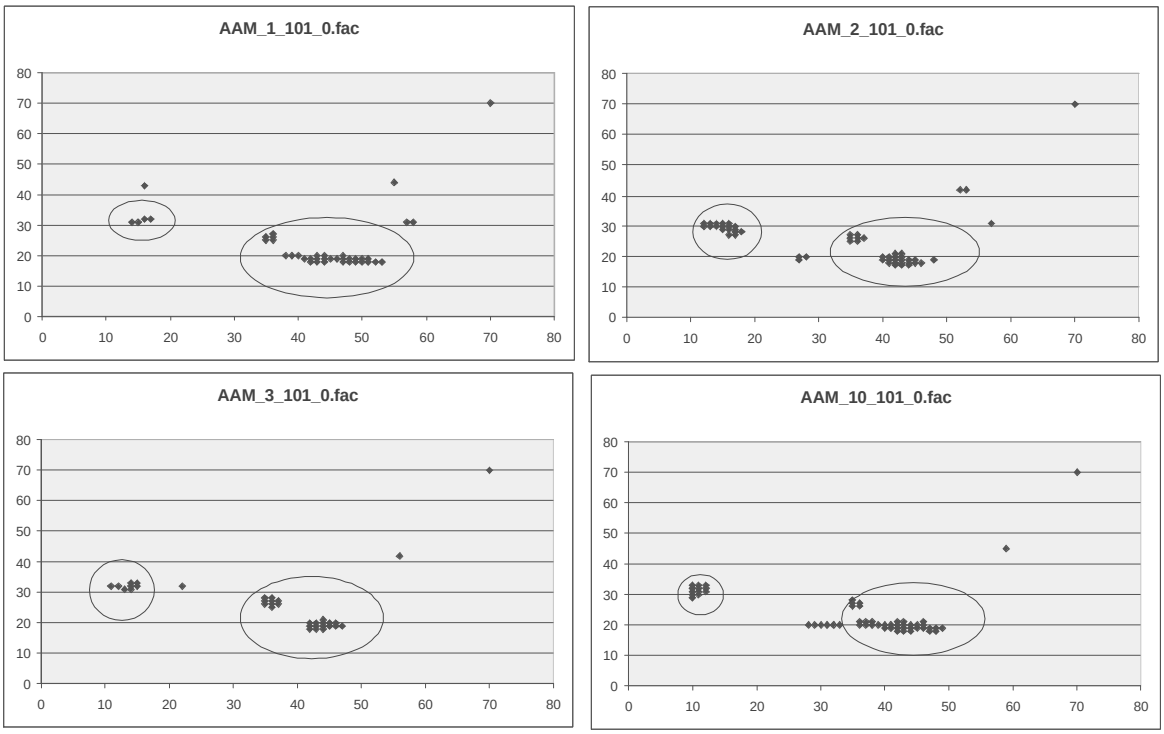

Figura 8 - Matriz de Variância Indivíduo AAM

Com as características relevantes - valor abaixo de um limiar - capturadas, é possível realizar a execução dos demais passos do método de Hashing Geométrico.

Como os valores de cada ponto são tridimensionais, o método é aplicado em um espaço tridimensional.

Neste ponto há uma alteração que contribuiu para a performance do sistema, ao invés do processo ser realizado para cada par ordenado do conjunto de pontos característica, a base característica utilizada em todas as faces é fixa e apenas uma, como mostra a figura 9, desta forma, o processo é realizado uma única vez.

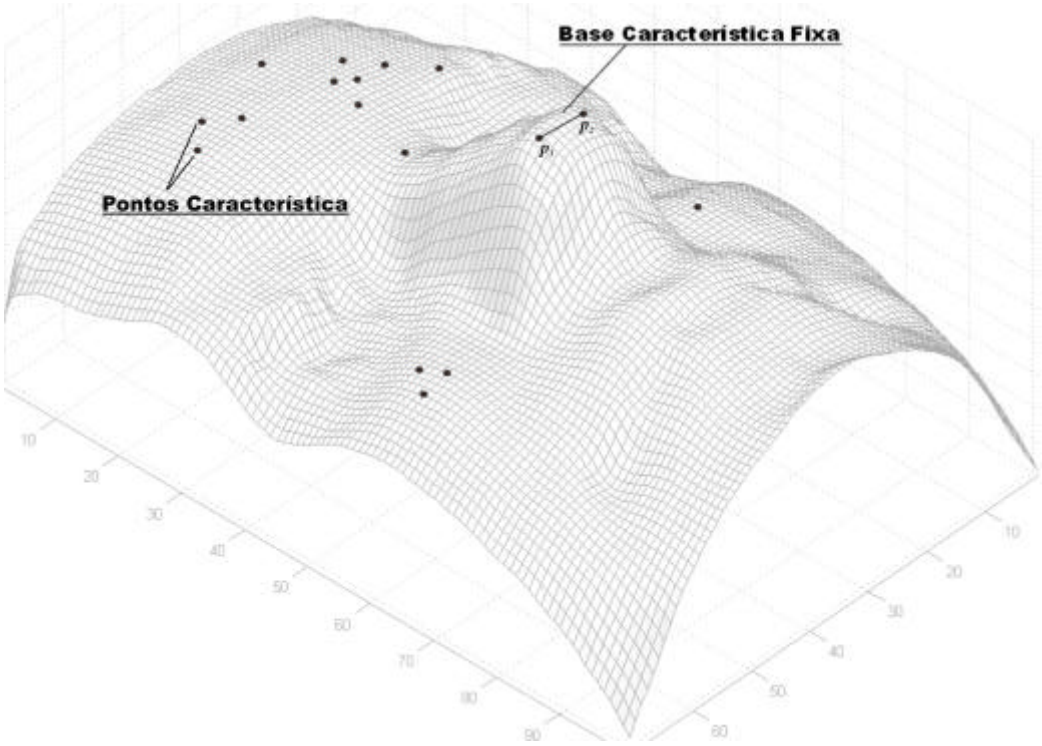

Figura 9 - Face com base e pontos característica

O passo de normalização do conjunto de pontos – rotação, escalonamento e translação - é realizado com a aplicação das matrizes de transformação mostradas na figura 10.

O modelo da face então construído, constituirá um objeto *ModeloFace* o qual conterá a face correspondente e os dados necessários à execução dos demais passos realizados pela tabela hash, nota-se que até o momento a face não está ligada a nenhuma identidade específica.

O fato de se utilizar vários pontos para a indexação da face, permite que a semântica da mesma seja explorada e em caso de alguma parte desta estar obscurecida – por exemplo, cabelo sobre a testa – ainda assim ser possível a sua indexação, e por conseqüência, seu reconhecimento.

Translação

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{bmatrix}$$

Escalonamento

$$S = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotação

$$Rx = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos q & senq & 0 \\ 0 & -senq & \cos q & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Ry = \begin{bmatrix} \cos q & 0 & -senq & 0 \\ 0 & 1 & 0 & 0 \\ senq & 0 & \cos q & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Rz = \begin{bmatrix} \cos q & senq & 0 & 0 \\ -senq & \cos q & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 10 - Matrizes de transformação

6.2.2.2 Tabela de Hash Geométrico

A Tabela de Hash Geométrico trabalha diretamente com os modelos de face construídos pelo subsistema anteriormente descrito. Em um primeiro momento, pode parecer um tanto estranho definir a tabela de Hash Geométrico como um subsistema, porém, tal tabela não se limita apenas em armazenar dados pré-processados, esta realiza processos importantes na indexação dos dados

armazenados, ou seja, é de sua responsabilidade a indexação efetiva das faces, seja para operações de armazenamento ou recuperação.

Para ambos os tipos de processo (armazenamento e recuperação) faz-se necessário a definição dos valores (m, u, j) de cada ponto característica, visto que o espaço é tridimensional. Isto é realizado escrevendo os mesmos como combinação linear da base característica previamente definida (seção 4.4.1.1).

Este processo pode ser realizado através da solução de sistemas, para tal foi utilizada a técnica de pivotação parcial.

Após terem sido computados os valores (m, u, j) automaticamente teremos a localização da entrada na tabela hash para o ponto em questão. No processo de armazenamento, se o sistema fosse implementado segundo o método original, para cada ponto acessaríamos sua entrada na tabela e adicionaríamos um identificador do modelo. Entretanto, algumas alterações foram realizadas.

Primeiramente, junto com a requisição de armazenamento, é informado o identificador de qual indivíduo o novo modelo pertence. Com isto, o armazenamento é direcionado pela identidade do indivíduo, armazenamos um novo modelo de face indicando a qual indivíduo ele pertence.

Outra alteração é o modo como o registro de cada modelo é armazenado nas entradas da tabela, isto será melhor explicitado nas próximas seções.

6.2.3 Sistema de Armazenamento

Este subsistema tem por função principal garantir a organização física dos dados armazenados na base de dados bem como a recuperação de dados do disco rígido para a memória principal e desta para o disco rígido, mantendo a consistência dos dados.

Como mencionado no início da seção, um modelo de uma face está ligado a um determinado indivíduo, sendo este o principal dado armazenado na base e o qual exige o controle executado pelo SA.

Basicamente o subsistema trabalha com duas estruturas principais:

- ***Dicionário de dados:*** armazena a localização física de cada objeto da base de dados. Pode ser visto como a interface que mapeia a visão lógica da base em endereços físicos no disco rígido;
- ***Dados em memória:*** tabela com a relação dos dados que estão no momento na memória principal juntamente com dados relevantes a sua instanciação e persistência, como exemplo, se é um dado novo ou antigo.

Suas requisições são originadas diretamente do SGD, as quais podem ser de dois tipos principais: *retornaInstanciaIndividuo* e *retornaNovaInstanciaIndividuo***.**

Na primeira, o SA recupera uma instância de um indivíduo e a retorna, na segunda uma nova instância de um indivíduo é criada e então retornada.

A definição de quais dados e quando serão armazenados é totalmente controlada pelo SA, já que esta é sua principal função no sistema.

Como o sistema neste momento não gerencia diversas bases de dados, o controle da recuperação e armazenamento da tabela hash correspondente a base não se faz necessário.

O processo de gerência dos dados pode ser descrita pela seqüência abaixo:

i. Uma instância de indivíduo é requisitada;
ii. O SA verifica se a instância requisitada não se encontra na memória no momento, retornando-a em caso positivo;
iii. Em caso negativo, o dicionário de dados é acessado e o endereço físico do dado é consultado;
iv. Caso alguma outra instância esteja ocupando a entrada na tabela de dados em memória necessária, esta é armazenada em disco e eliminada da memória principal;
v. O dado é recuperado do disco rígido e então adicionado na tabela de dados em memória;

Todo o processo descrito acima é guiado pelo IDO de cada objeto indivíduo.

6.2.3.1 Modelo de Dados

O modelo de dados utilizado pelo sistema é simplificado e consiste de um modelo orientado a objeto. Os dados podem ser classificados em duas grandes

classes: (i) dado e (i) dado complexo. A figura 11 mostra resumidamente o modelo de dados do sistema.

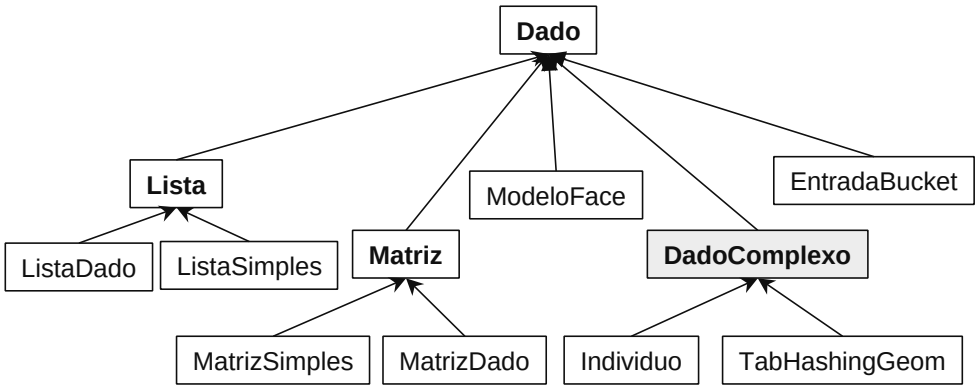

Figura 11 - Modelo de dados do sistema

O que diferencia tais classes das demais criadas no desenvolvimento do sistema é que estas possuem suporte a persistência de dados, ou seja, podem ser armazenadas em disco rígido sem perderem a consistência.

Um dado complexo possui um IDO o que permite a sua recuperação somente através de uma referência a este atributo, pois é garantindo pelo sistema que ele será único.

Abaixo, uma descrição sucinta dos tipos mais especializados:

- ***ListaSimples:*** lista encadeada, armazena valores de tipos simples, como inteiros, booleanos;
- ***ListaDado:*** lista encadeada que armazena elementos subclasse do tipo *Dado*;
- ***MatrizSimples:*** matriz de tamanho fixo que armazena tipos simples;
- ***MatrizDado:*** matriz de tamanho fixo que armazena elementos subclasse do tipo *Dado*;
- ***Individuo:*** representa a identidade de uma pessoa, possui um IDO e armazena consigo os modelos que o definem;

- ***TabHashGeom:*** representa uma tabela de hash geométrico, com todos os seus métodos e atributos;
- ***ModeloFace:*** é o modelo de uma determinada face, armazena a face que lhe deu origem;

Todas as classes subclasses de Dado respeitam as regras de persistência definidas a seguir.

6.2.3.2 Persistência dos dados

Os dados persistentes foram criados respeitando certas regras, as quais definem o modo como devem ser os registros em disco rígido. Apenas dados complexos possuem sua localização definida no dicionário de dados.

Quando o SA deseja que um dado persista, este apenas requisita ao objeto que o faça, enviando juntamente com a mensagem o arquivo onde o dado será armazenado.

Os dados são armazenados segundo uma regra de formação de registro, definida pela gramática apresentada na figura 12.

Apesar do sistema ser orientado a objeto, como demonstrado na anteriormente, objetos podem ser representados por registros em um processo de persistência.

REGISTRO *:= tamanho tipoDADO*

DADO := *DADO_COMPLEXO* | *LISTA* | *MATRIZ* | *MODELO_FACE* | *ENTRADA_BUCKET*

DADO_COMPLEXO := oid INDIVIDUO | oid TAB_HASH

INDIVIDUO := MODELOS

MODELOS := MODELO_FACE MODELOS

TAB_HASH :=MATRIZ_DADO

LISTA := linhas colunas z LISTA_SIMPLES | linhas colunas z LISTA_DADO

LISTA_SIMPLES:= valor LISTA_SIMPLES | e

LISTA_COMPLEXA := DADO LISTA_COMPLEXA | e

MATRIZ := nElem vazio MATRIZ_SIMPLES | nElem vazio MATRIZ_DADO

MATRIZ_SIMPLES := valor MATRIZ_SIMPLES | e

MATRIZ_DADO := DADO MATRIZ_DADO | e

MODELO_FACE:= FACE PONTOS_CARAC MATRIZ_FUNCAO

FACE := MATRIZ_SIMPLES

PONTOS_CARAC := LISTA_DADO

MATRIZ_FUNCAO := MATRIZ_SIMPLES

ENTRADA_BUCKET := peso oid

Figura 12 - Gramática de formação de registro

6.3 Inteligência do Sistema

Cada entrada da tabela hash possui uma lista, a qual contém um elemento *EntradaBucket*, tal elemento armazena o identificador do indivíduo e um valor peso. Este valor peso indica qual a representatividade deste elemento no total de pontos características do indivíduo.

Quando no processo de armazenamento, a representatividade de cada ponto do conjunto característica do modelo a ser armazenado é calculada - 1/(total_pontos_cararacteristica.

Para cada ponto, sua entrada na tabela é calculada e acessada. Dentre os elementos contidos na entrada é procurado um *ElementoBucket* que corresponda ao mesmo indivíduo do modelo da face. Caso nenhum elemento seja encontrado, um novo é criado e então adicionado com a representatividade previamente calculada.

Caso contrário, quando um *ElementoBucket* coincidente é encontrado, o valor de peso armazenado será o maior dentro o atual e o da nova representatividade.

O modelo adotado vem do fato de que uma face é representada pelo conjunto de pontos característica dela gerada. Portanto, cada ponto possui uma representatividade e a soma de todas define uma face.

Algumas faces geram um número pequeno de pontos característica enquanto outras, muitos. Não bastando tal fato, pontos próximos definem entradas iguais na tabela hash. Caso os votos fossem computados somando a cada ocorrência um voto, teríamos *ElementoBucket* com um número enorme de votos, enquanto outros um número muito baixo. Desta forma ficaria pouco evidente qual o indivíduo correto.

Segundo (BARRETO, 2001, p 5) imagina-se que um problema necessita de um determinado grau de inteligência para ser resolvido. Quanto maior a dificuldade em resolve-lo, maior o grau de inteligência necessário. O autor conclui que, se um dispositivo artificial é capaz de resolver um problema, então tal dispositivo possui o grau de inteligência exigido para atingir a solução. Baseado nesta definição pode-se afirma que o sistema construído é inteligente, visto que este é capaz de identificar um indivíduo e refinar seu comportamento, problema este que exige significativo grau de inteligência.

Retomando o modelo proposto na seção 6.2, para o sistema, objetos *ModeloFace* representam os modelos mentais adquiridos a medida que melhor conhecemos um indivíduo, enquanto o processo de designar a representatividade de cada característica permite que o comportamento de identificação se efetive.

O aprendizado do sistema ocorre à medida que modelos das faces são adicionados à tabela hash, pois caso alguma característica relevante não seja evidente em uma face A, e seja em outra B, esta será apreendida quando da adição de B na tabela. Além disto características pouco relevantes podem ter sua real evidência adquirida através dos diversos modelos adicionados.

Como classificação do sistema podemos defini-lo como um sistema de raciocino baseado em caso, mais especificamente de raciocínio baseado em memória. Para tal a tabela de hash é vista como a memória, a qual é utilizada exaustivamente, além disto pela simplicidade das faces tratadas temos uma descrição pobre do domínio. O modelo de banco de dados orientado a objeto permite que ao identificarmos um indivíduo possamos ter acesso – lembrar - a demais características do mesmo. Finalmente a cada experiência adquirida - novo modelo de face adicionado - a memória é alterada.

7 Testes e Resultados

Os testes realizados tiveram como objetivo simular o funcionamento do sistema em uma operação de verificação efetiva. Havia à disposição duas bases de dados, com uma diferença de 13 dias na aquisição das faces.

Os testes foram conduzidos da seguinte forma:

- Dados sobre os indivíduos foram inseridos no sistema utilizando os dados de uma das duas bases disponíveis;
- O procedimento de identificação é realizado para alguns dos indivíduos armazenados utilizando a segunda base de faces disponível.
- No total foram armazenados 5 amostras por indivíduo. Três cenários são propostos.

Cenário 1

O primeiro cenário foi pensando em uma situação onde amostras da face de um indivíduo são capturadas em um dia e após 13 dias, o mesmo indivíduo passa pelo sistema. Este deverá ser capaz de reconhece-lo. É tolerado que o sistema não reconheça com exatidão posições de face que sejam significativamente diferentes das originais capturadas.

Para as posições de face coincidentes com as armazenadas inicialmente, o sistema conseguiu identificar o indivíduo em 100% dos casos. No caso de posições diferentes, em média o sistema reconheceu o indivíduo em 80% dos casos.

Cenário 2

No segundo cenário, pretende-se testar a capacidade de aprendizado do sistema para um mesmo indivíduo. Para o primeiro teste, o sistema com 5 amostras por indivíduo é utilizada, e posteriormente são acrescentadas 3 amostras de face. Os testes de reconhecimento são realizados para 20 posições de face diferentes.

O resultado pode ser vizualizado no Gráfico 1, em média a taxa de reconhecimento teve um aumento de cerca de 30%.

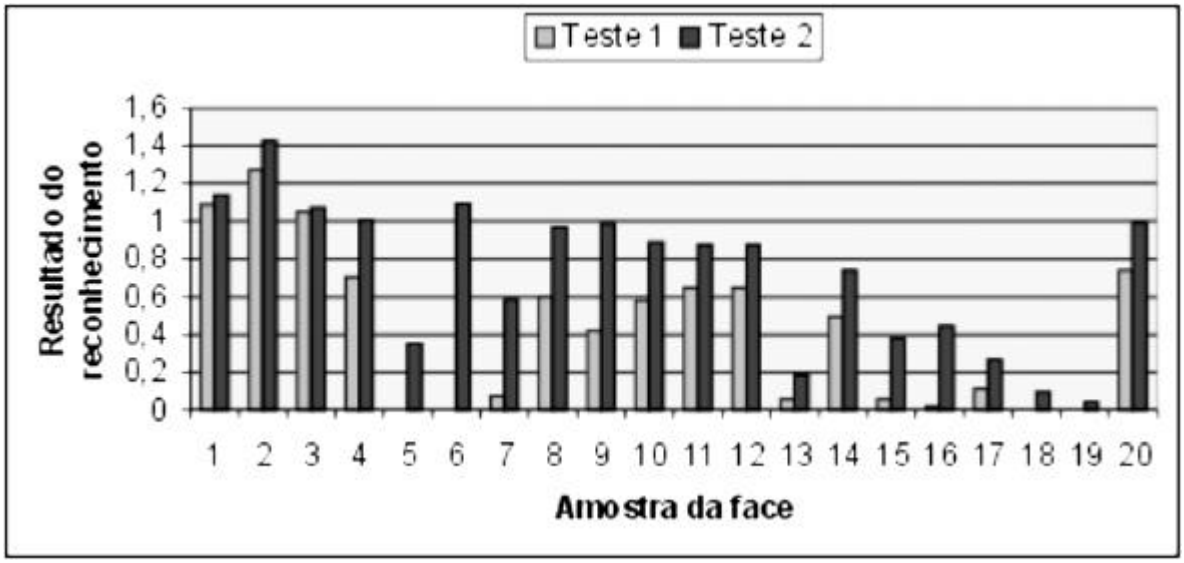

Gráfico 1 - Evolução do aprendizado

Cenário 3

Neste cenário foi testado o impacto no reconhecimento da ocorrência de ruído.

Primeiramente foi utilizado o sistema com 5 amostras por indivíduo, primeiramente foi realizado o reconhecimento com 20 amostras sem ruído, e posteriormente as mesmas 20 posições porém com 1 nível de ruído.

A saída do sistema é da seguinte forma :

Posição 1	
OID	Taxa
1	1,20026
14	0,0734955
17	0,393822
5	0,0533108
3	0,0740741
10	0,0421053

Como pode ser percebido, o sistema retorna as identidades encontradas no reconhecimento. Foi considerado como reconhecimento positivo quando a taxa de reconhecimento do indivíduo estiver dentre as 3 mais altas da lista retornada.

Com o primeiro indivíduo (AAM) houve 80% de reconhecimento satisfatório nas entradas sem ruído. Quando adicionado ruído o reconhecimento saltou para 95%, um aumento considerável.

O segundo indivíduo (aw#) teve 70% de reconhecimento satisfatório sem ruído e 80% sem ruído.

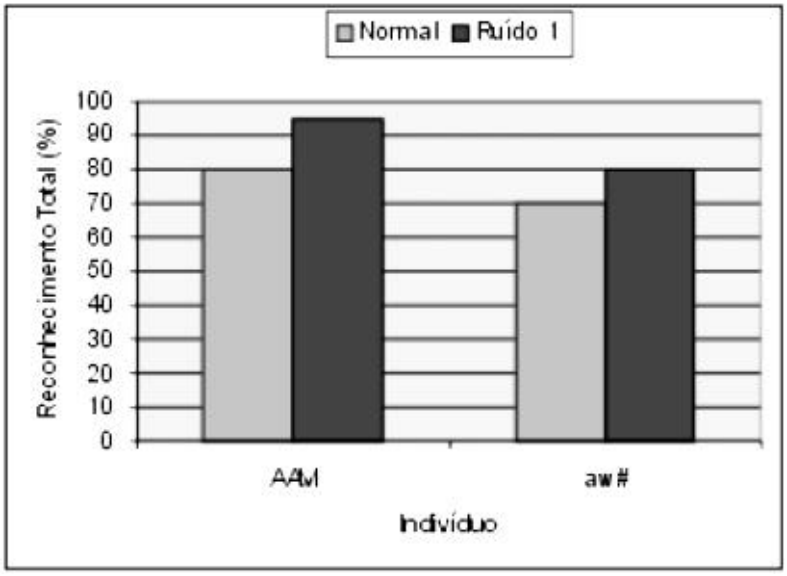

Gráfico 2 - Comparação do reconhecimento com adição de 1 nível de ruído

8 Considerações Finais

O sistema construído atingiu os objetivos propostos e cumpriu os requisitos definidos, isto foi possível principalmente pelo fato da tabela hash permanecer em memória, permitindo que a execução das tarefas de indexação fossem realizadas de modo rápido e eficiente.

Além de ter cumprido seus objetivos, a pesquisa em questão teve uma contribuição importante à área de banco de dados, visto que a técnica de indexação de Hashing Geométrico foi redefinida, tornando possível a sua utilização na construção de um sistema de raciocínio baseado em memória.

Tal método redefinido pode ser utilizado em outros tipos de sistemas, como exemplo, sistemas gerenciadores de dados geográficos. Como principal vantagem do método está a possibilidade de sua utilização em ambientes onde a descrição do domínio é muito pobre e o tipo de dado tratado ser mal comportado.

A abordagem utilizada – raciocínio baseado em memória juntamente com Hashing Geométrico – não foi encontrada na literatura, fato este que se tornou uma das principais dificuldades encontradas no desenvolvimento desta pesquisa. Foi notada uma relativa escassez de fontes que tratassem de aspectos específicos de Raciocínio Baseado em Memória e principalmente Hashing Geométrico, fazendo com que o tempo previsto para a aquisição de embasamento teórico necessário ao desenvolvimento do sistema se prolongasse além do que era esperado.

Serão necessários mais testes, principalmente com bases de faces maiores, para que seja determinada com exatidão a eficiência do método definido. A

exploração de outros tipos de características também é válida, podendo contribuir para a melhora do processo de reconhecimento.

Não foram realizados testes específicos com tipos étnicos ou gêmeos idênticos. Neste último caso, é esperada uma confusão momentânea, no pré-reconhecimento e, posteriormente uma confirmação da identidade do indivíduo, caracterizando mais uma vez um comportamento semelhante ao processo de identificação realizado por um ser humano, inspiração do modelo proposto.

Como proposta de novos trabalhos, o desenvolvimento de um sistema de gerência de memória mais aprimorado, que explore as características dos dados tratados, o desenvolvimento de um compilador de consultas e a definição de uma linguagem de consulta, permitindo a execução de consultas em modo terminal. Por fim o estudo de métodos de armazenamento persistente mais modernos.

REFERÊNCIAS

ABEL, Mara. **Raciocínio Baseado em Casos –** CBR. Universidade Federal do Rio Grande do Sul. 1999.

AGNAR, Aamodt. PLAZA, Enric. Case-Based Reasoning – Foundational Issues, Methodological Variations and System Aproaches. Artificial Intelligence Communications 1994.

BARRETO, Jorge Muniz. Inteligência Artificial: No Limiar do Século XXI, Abordagem Híbrida Simbólica Conexionista e Evolutiva. 3° ed. ??? Edições : Florianópolis. 2001.

DENG, Kan. MOORE, Andrew W. Multiresolution Instance-Based Learning. [s.l]. 1995.

HENRY, F. Korth. SILBERSCHATZ, Abraham. Sistema de Bancos de Dados. 2° ed. Makron Books : São Paulo. 1993.

KALIPSIZ, Oya. **Multimedia Databases**. IEEE. 2000.

LARSON, Per-Ake. **Dynamic Hash Tables**. Communications of the ACM. Abril 1998.

MARIN, Luciene de Oliveira. Investigações sobre Rede Neurais Artificiais para o Reconhecimento de Faces Humanas na Forma 3D. Dissertação (Mestrado em Ciência da Computação) : Universidade Federal de Santa Catarina – Florianópolis. 2003. 132 f.

MEHLHORN, Kurt. **Sorting and Searching**. Springer-Verlag : Berlin. 1977.

MOLINA, Hector Garcia; ULLMAN, Jeffrey D.; WIDOM, Jennifer. Implementação de Sistemas de Bancos de Dados. 2001

MORI, Greg; Belongie, Serge; Malik, Jitendray. Shape Contexts Enable Efficient Retrieval of Similar Shapes. 2001

MORRIS, John. **Data Structures and Algorithms:** Hash Tables. 1998.

RATHI, Ashok. LU, Huizhu. HEDRICK, G.E. Performance Comparison of Extendible Hashing and Linear Hashing Techniques. ACM. 1990.

RICH, Elaine. KNIGHT, Kevin. Inteligência Artificial. 2° ed. Makron Books:São Paulo. 1993.

SHANK, Roger C. Reminding and Memory. Proceedings of the DARPA Case-Based Reasoning Workshop. 1988.

SPÖRL, Brigitte Bartsch. LENZ, Mario. HÜBNER, André. Case-Based Reasoning – Survey and Future Directions. 1999.

STANFILL, Craig. WALTZ, David. The Memory-Based Reasoning Paradigm. Proceedings of the DARPA Case-Based Reasoning Workshop. 1988.

STANFILL, Craig. WALTZ, David. Toward Memory-Based Reasoning. Communications of the ACM. 1986.

WOLFSON, Haim J. Geometric Hashing : An Overview. IEEE Computational Science & Engineering. 1997.

ZIMMERMANN, Antonio Carlos. Reconhecimento de faces humanas através de técnicas de inteligência artificial aplicadas a formas 3D. Tese (Doutorado em Engenharia Elétrica) - Universidade Federal de Santa Catarina. Florianópolis. 2003.

APÊNDICES

Otimização do Processo de Armazenamento e Recuperação de Informações Tridimensionais de Bancos de Dados

Mauricio Schoenfelder
Departamento de Informática e Estatística
Universidade Federal de Santa Catarina

Resumo

O armazenamento de elementos de dados não convencionais faz com que as técnicas de estruturação e gerenciamento de informações devam ser redefinidas, levando em conta a semântica da informação. Atualmente o projeto SORFACE (Sistema Óptico de Reconhecimento de Faces) trata de identificar se duas faces confrontadas são ou não da mesma pessoa, caracterizando um cenário de verificação, em uma extensão, o sistema deverá ser capaz de realizar a identificação de um indivíduo, ou seja, encontrar a identidade de uma pessoa sem que esta seja declarada. A utilização do método de hashing geométrico e raciocínio baseado em memória permitiu o desenvolvimento de um protótipo de um sistema gerenciador de dados capaz de indexar as faces geradas pelo SORFACE utilizando unicamente seu conteúdo e capaz de aprender com os processos de reconhecimento realizados.

1 Introdução

Sistemas gerenciadores de informações não convencionais devem ser capazes de realizar buscas tendo como chave de procura o próprio dado. Tal fato faz com que as técnicas de estruturação devam ser redefinidas devido às características singulares de cada tipo de informação. Por exemplo, as características que identificam um arquivo de som podem não ser as mesmas que identificam um arquivo de imagem. Normalmente, sistemas de banco de dados comerciais, tratam dados não usuais de modo genérico e efetuam suas consultas segundo dados textuais descritivos.

O desenvolvimento de sistemas de banco de dados que tratem dados não usuais de modo não genérico, normalmente é realizado restringindo o domínio de aplicação do sistema. Por tratarem de dados complexos, muitos sistemas deste tipo utilizam o modelo orientado a objeto para banco de dados, evitando conversões entre modelos – por exemplo, de um modelo orientado a objeto para relacional.

O projeto SORFACE (Sistema Óptico de Reconhecimento de Faces) trata do reconhecimento de faces humanas a partir de sua forma geométrica 3D. Os modelos tridimensionais após terem sido captados serão guardados em uma base de dados para uso posterior. Dois cenários são propostos para o processo de reconhecimento, verificação e identificação.

Os dados referentes às faces podem ser vistos como uma classe de dados complexos, e, portanto possuem características próprias que não se aplicam a dados textuais ou numéricos. Tais informações possuem uma estrutura bastante simples, porém, sua semântica é rica e deve ser a principal característica explorada em uma busca. Caso fossem utilizadas técnicas de comparação exaustiva – tal como distância de Hamming - em uma base de dados com muitas faces o tempo de recuperação de uma imagem próxima à imagem dada pode tornar-se proibitivo, quando somado ao tempo de verificação.

Como solução proposta para tal problema foi a utilização de uma técnica de indexação por conteúdo com o principal requisito de não realizar comparações exaustivas. Somado a técnica de indexação, métodos de inteligência artificial (IA) e modelos orientados a objeto possibilitaram uma diminuição significativa dos custos de armazenamento e recuperação dos dados tratados.

2 Trabalhos Relacionados

O projeto SORFACE tem por objetivo gerar modelos tridimensionais de faces humanas e tornar possível o seu reconhecimento. Tais modelos constituem-se de uma matriz onde cada célula corresponde à altura de um ponto na superfície da face. Tal técnica é apresentada em (ZIMMERMANN , 2003).

Dois cenários são propostos pelo projeto, o primeiro é o de verificação, onde o indivíduo declara sua possível identidade e o sistema tem como função confirmar ou refutar tal declaração, este caso foi objeto de estudo de (MARIN, 2003) e foi tratado com técnicas de IA conexionista e evolucionária.

O segundo cenário consiste em identificar um indivíduo capturado pela câmera sem nenhuma declaração realizada. Para tal, o sistema objeto desta pesquisa fica responsável por buscar dentre os indivíduos armazenados na base de dados, aqueles que possuem maior similaridade com a face de entrada. Em uma segunda etapa, o sistema de verificação já desenvolvido é acionado e uma comparação mais refinada é realizada para finalmente a real identidade ser apontada.

3 Concepção do Sistema

O modelo do sistema foi concebido tendo como inspiração o processo de reconhecimento humano. Podemos descrever tal processo pela seguinte sucessão de ações:

iv. Ao encontrarmos alguém, buscamos ocorrências da pessoa dentre os diversos outros conhecidos;
v. Possíveis candidatos são escolhidos e um pré-reconhecimento é realizado (sistema proposto);
vi. Os candidatos passam por uma avaliação mais refinada e a decisão da identidade correta é tomada (SORFACE).

Após isto, conseguimos lembrar o nome da pessoa bem como dados mais específicos como idade, índole. Quanto maior a interação ou mesmo a convivência com tal indivíduo, maior será a quantidade de modelos mentais da pessoa apreendidos, e mais rápido será o reconhecimento, pois mais refinada será a descrição de tal pessoa, resultando em um conjunto de possíveis indivíduos com um número menor de identidades incorretas. Por conseqüência, menor será o número de verificações inúteis, o que diminui o tempo de avaliação.

3.1 Arquitetura do Sistema

A arquitetura básica do sistema é constituída de três elementos principais (figura 1), são eles:

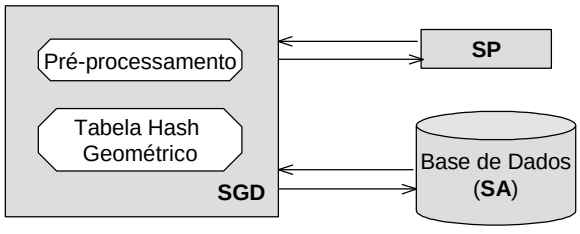

Figura 1 - Arquitetura Básica do Sistema

- ***Sistema Principal (SP)****:* recebe os dados das faces oriundas do sistema SORFACE e as requisições externas. É responsável por decidir se há alguma ocorrência na base de dados que coincida com a face de entrada.
- ***Sistema Gerenciador de Dados (SGD):*** recebe solicitações do SP - retornar possíveis candidatos, incluir face de indivíduo - as processa e então retorna os resultados obtidos. A técnica principal utilizada é a de Hashing Geométrico, devido a sua adequação ao tipo de dado tratado - mal comportado - e ao fato de ter sua expansão para o espaço tridimensional relativamente simples. É responsável pela organização dos dados e provê a visão lógica dos mesmos ao SP. É a interface entre o SP e o sistema de armazenamento.
- ***Sistema de armazenamento (SA):*** tem como função manter a organização física dos dados, ou seja, manter os dados corretamente gravados. Também é de sua competência prover a transição dos dados gravados para a memória principal, e desta para o disco rígido, utilizando para isto, um dicionário de dados.

Pode ser observado que o sistema não será do tipo relacional, mas sim orientado a objeto, assim, objetos complexos terão seus identificadores únicos.

3.2 Modelo de Dados

O modelo de dados utilizado pelo sistema é simplificado e consiste de um modelo orientado a objeto. Os dados podem ser classificados em duas grandes classes: (i) dado e (i) dado complexo. A figura 2 mostra resumidamente o modelo de dados do sistema.

Um dado complexo possui um IDO o que permite a sua recuperação somente através de uma referência a este atributo, pois é garantindo pelo sistema que ele será único.

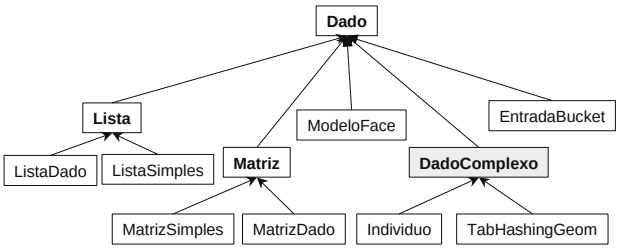

Figura 2 - Modelo de dados do sistema

Todas as classes subclasses de Dado respeitam as regras de persistência definidas a seguir.

3.3 Persistência dos dados

Os dados persistentes foram criados respeitando certas regras, as quais definem o modo como devem ser os registros em disco rígido. Apenas dados complexos possuem sua localização definida no dicionário de dados.

Quando o SA deseja que um dado persista, este apenas requisita ao objeto que o faça, enviando juntamente com a mensagem o arquivo onde o dado será armazenado.

Apesar do sistema ser orientado a objeto, como demonstrado na anteriormente, objetos podem ser representados por registros em um processo de persistência.

4 Testes e Resultados

Os testes realizados tiveram como objetivo simular o funcionamento do sistema em uma operação de verificação efetiva. Havia à disposição duas bases de dados, com uma diferença de 13 dias na aquisição das faces.

Os testes foram conduzidos da seguinte forma:

- Dados sobre os indivíduos foram inseridos no sistema utilizando os dados de uma das duas bases disponíveis;
- O procedimento de identificação é realizado para alguns dos indivíduos armazenados utilizando a segunda base de faces disponível.
- No total foram armazenados 5 amostras por indivíduo.

Três cenários são propostos.

Cenário 1

O primeiro cenário foi pensando em uma situação onde amostras da face de um indivíduo são capturadas em um dia e após 13 dias, o mesmo indivíduo passa pelo sistema. Este deverá ser capaz de reconhece-lo. É tolerado que o sistema não reconheça com exatidão posições de face que sejam significativamente diferentes das originais capturadas.

Para as posições de face coincidentes com as armazenadas inicialmente, o sistema conseguiu identificar o indivíduo em 100% dos casos. No caso de posições diferentes, em média o sistema reconheceu o indivíduo em 80% dos casos.

Cenário 2

No segundo cenário, pretende-se testar a capacidade de aprendizado do sistema para um mesmo indivíduo. Para o primeiro teste, o sistema com 5 amostras por indivíduo é utilizada, e posteriormente são acrescentadas 3 amostras de face. Os testes de reconhecimento são realizados para 20 posições de face diferentes.

O resultado pode ser vizualizado no Gráfico 1, em média a taxa de reconhecimento teve um aumento de cerca de 30%.

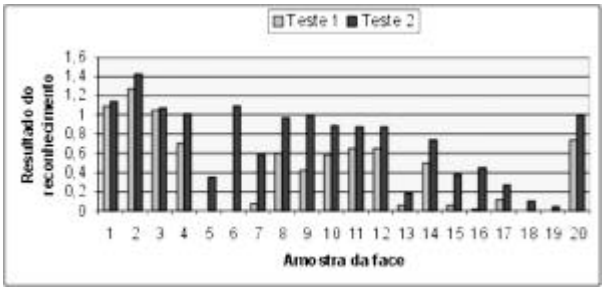

Gráfico 1 - Evolução do aprendizado

Cenário 3

Neste cenário foi testado o impacto no reconhecimento da ocorrência de ruído.

Primeiramente foi utilizado o sistema com 5 amostras por indivíduo, primeiramente foi realizado o reconhecimento com 20 amostras sem ruído, e posteriormente as mesmas 20 posições porém com 1 nível de ruído.

A saída do sistema é da seguinte forma:

Posição 1	
OID	Taxa
1	1,20026
14	0,0734955
17	0,393822
5	0,0533108
3	0,0740741
10	0,0421053

Como pode ser percebido, o sistema retorna as identidades encontradas no reconhecimento. Foi considerado como reconhecimento positivo quando a taxa de reconhecimento do indivíduo estiver dentre as 3 mais altas da lista retornada.

Com o primeiro indivíduo (AAM) houve 80% de reconhecimento satisfatório nas entradas sem ruído. Quando adicionado ruído o reconhecimento saltou para 95%, um aumento considerável.

O segundo indivíduo (aw#) teve 70% de reconhecimento satisfatório sem ruído e 80% sem ruído.

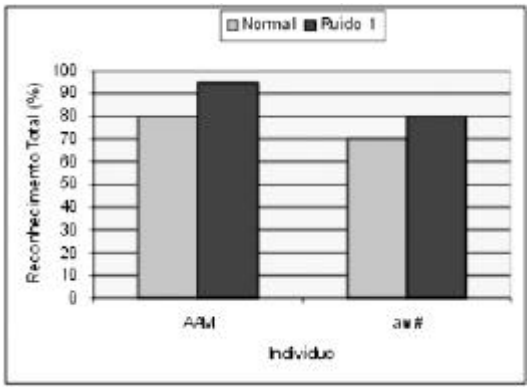

Gráfico 2 - Comparação do reconhecimento com adição de 1 nível de ruído

5 Bibliografia

ABEL, Mara. **Raciocínio Baseado em Casos –** CBR. Universidade Federal do Rio Grande do Sul. 1999.

AGNAR, Aamodt. PLAZA, Enric. **Case-Based Reasoning –** Foundational Issues, Methodological Variations and System Aproaches. Artificial Intelligence Communications 1994.

BARRETO, Jorge Muniz. **Inteligência Artificial:** No Limiar do Século XXI, Abordagem Híbrida Simbólica Conexionista e Evolutiva. 3° ed. ??? Edições : Florianópolis. 2001.

DENG, Kan. MOORE, Andrew W. **Multiresolution Instance-Based Learning.** [s.l]. 1995.

HENRY, F. Korth. SILBERSCHATZ, Abraham. **Sistema de Bancos de Dados.** 2° ed. Makron Books : São Paulo. 1993.

KALIPSIZ, Oya. **Multimedia Databases**. IEEE. 2000.

LARSON, Per-Ake. **Dynamic Hash Tables**. Communications of the ACM. Abril 1998.

MARIN, Luciene de Oliveira. **Investigações sobre Rede Neurais Artificiais para o Reconhecimento de Faces Humanas na Forma 3D**. Dissertação (Mestrado em Ciência da Computação) : Universidade Federal de Santa Catarina – Florianópolis. 2003. 132 f.

MEHLHORN, Kurt. **Sorting and Searching**. Springer-Verlag : Berlin. 1977.

MOLINA, Hector Garcia; ULLMAN, Jeffrey D.; WIDOM, Jennifer. **Implementação de Sistemas de Bancos de Dados**. 2001

RICH, Elaine. KNIGHT, Kevin. **Inteligência Artificial.** 2° ed. Makron Books:São Paulo. 1993.

SHANK, Roger C. **Reminding and Memory**. Proceedings of the DARPA Case-Based Reasoning Workshop. 1988.

SPÖRL, Brigitte Bartsch. LENZ, Mario. HÜBNER, André. **Case-Based Reasoning –** Survey and Future Directions. 1999.

STANFILL, Craig. WALTZ, David. **The Memory-Based Reasoning Paradigm.** Proceedings of the DARPA Case-Based Reasoning Workshop. 1988.

STANFILL, Craig. WALTZ, David. **Toward Memory-Based Reasoning.** Communications of the ACM. 1986.

WOLFSON, Haim J. **Geometric Hashing :** An Overview. IEEE Computational Science & Engineering. 1997.

ZIMMERMANN, Antonio Carlos. **Reconhecimento de faces humanas através de técnicas de inteligência artificial aplicadas a formas 3D.** Tese (Doutorado em Engenharia Elétrica) - Universidade Federal de Santa Catarina. Florianópolis. 2003.

SORFACEBD H

```
#ifndef Sistema_SorfaceBDH
#define Sistema_SorfaceBDH

#include "SGD.h"
#include "Face.h"
#include "ModeloFace.h"
#include "EntradaBucket.h"
#include "Lista.h"
#include "Individuo.h"

class SorfaceBD{
public:
    SorfaceBD();

    void adiciona(MatrizSimples<double>* face, long int oid);
    long int adicionaNovoIndividuo(MatrizSimples<double>* face);
    Individuo* retornaInstancia(long int oid);
    ListaDado<EntradaBucket>* identifica(MatrizSimples<double>* face);

      void alteraLimite(double novoLimite);
      double retornaLimite();

private:
    long int ultimoOID;
    SGD sgd;

};
//---------------------------------------------------------------------
-------
#endif
```

SORFACEBD CPP

```
#include "Sistema_SorfaceBD.h"

SorfaceBD::SorfaceBD()
{
    ultimoOID = 0;

}

void SorfaceBD::adiciona(MatrizSimples<double>* face , long int oid)
{
      sgd.adiciona(face, oid);
}

long int SorfaceBD::adicionaNovoIndividuo(MatrizSimples<double>* face)
{
    ultimoOID++;
    sgd.adicionaNovoIndividuo(face,ultimoOID);
    return ultimoOID;
}

ListaDado<EntradaBucket>* SorfaceBD::identifica(MatrizSimples<double>*
face)
{
    ListaDado<EntradaBucket>* retorno =
sgd.retornaPossiveisIdentidades(face);

    return retorno;
}

void SorfaceBD::alteraLimite(double novoLimite)
{
      sgd.alteraLimite(novoLimite);
}

double SorfaceBD::retornaLimite()
{
      return sgd.retornaLimite();
}

Individuo* SorfaceBD::retornaInstancia(long int oid)
{
    return sgd.retornaIndividuo(oid);
}
```

DADO H

```
#ifndef DadoH
#define DadoH
#include <iostream>
#include <fstream>

#define TAM_INT sizeof(int)
#define TAM_CHAR sizeof(char)
#define TAM_BOOL sizeof(bool)
#define TAM_LONG sizeof(long)
#define TAM_DOUBLE sizeof(double)

/* CONSTANTES DE USO GERAL */

#define TAMANHO 0
#define TIPO TAM_INT
#define OID TIPO + TAM_CHAR

/* CONSTANTES DE USO MATRIZ */

#define LINHAS TIPO + TAM_CHAR
#define COLUNAS LINHAS + TAM_INT
#define NZ COLUNAS + TAM_INT
#define CHEIO NZ+TAM_INT

/* CONSTANTES DE USO LISTA */

#define NELEM TIPO + TAM_CHAR
#define ELEMENTOS NELEM + TAM_INT

/* CONSTANTES DE USO MODELOFACE E TABHASHGEOM */
#define DADOS OID + TAM_LONG

using namespace std;

class Dado
{
    public:
        unsigned tipo(){ return ti; }

    protected:

        Dado(char tipo){
            tam = 0;
            ti = tipo;
        }
        Dado(char* registro){
            tam = *((unsigned*)&registro[TAMANHO]);
            ti = *((char*)&registro[TIPO]);
        }

        void persista(ofstream* stream)
        {
            stream->write((char*)&tam,TAM_INT);
            stream->write((char*)&ti,TAM_CHAR);
        }

        unsigned tamanho()
        {
```

```
            return (TAM_INT + TAM_CHAR);
        }


        unsigned tam;
        char ti;

};


/*************************************************/


class DadoComplexo : public Dado
{
public:
    long retornaOID(){ return oid;}

protected:

    DadoComplexo(long oid, char tipo):Dado(tipo)
    {
        DadoComplexo::oid = oid;
    }

    DadoComplexo(char* registro) : Dado(registro)
    {
        oid = *((long*)&registro[OID]);
    }

    void persista(ofstream* stream)
    {
        Dado::persista(stream);
        stream->write((char*)&oid,TAM_LONG);
    }

    unsigned tamanho()
    {
        unsigned retorno = Dado::tamanho();
        retorno += TAM_LONG;
        return retorno;
    }

    long oid;

};

#endif
```

LISTA H

```
#ifndef ListaH
#define ListaH

#include "Dado.h"
#include <iostream>

using namespace std;

template<class A>

class ElemLista{
public:

    ElemLista(A* dado)
        { ElemLista::dado = dado; }

        A* dado;
        ElemLista<A>* proximo;
};


                /************************/
/* ********************************************************** */
                /************************/



template <class B>
class Lista : public Dado
{
public:
/* Sequencia do registro = tamanho(int)  tipo(char) dados
    dados = nElem( int ) elementos( D )
*/
Lista():Dado((char)1)
{
        nElem = 0;
        nElemInicio = 0;
}


B* retorna(int indice)
{

        if( indice < nElem ){

            ElemLista<B>* atual = primeiro;

            for( int i=0 ; i < indice ; i++)
            {
                atual = atual->proximo;
            }
            return atual->dado;
        }
}

void adiciona(B* dado){
```

```
        if( nElem > 0 ){
            ElemLista<B>* novoElem = new ElemLista<B>(dado);
            ultimo->proximo = novoElem;
            ultimo = novoElem;
        }else{
              primeiro = new ElemLista<B>(dado);
              ultimo = primeiro;
        }

        nElem = nElem+1;
}

void elimina(int indice)
{

        if( indice < nElem ){
            ElemLista<B>* anterior = primeiro;
            ElemLista<B>* atual = primeiro;

            for(int i=0 ; i < indice ; i++)
            {
                anterior = atual;
                atual = atual->proximo;
            }

            anterior->proximo = atual->proximo;

            free(atual);
            nElem = nElem - 1;
        }
}



int retornaNElem(){ return nElem; }

protected:

    unsigned nElemInicio;
    unsigned nElem;
    ElemLista<B>* primeiro;
    ElemLista<B>* ultimo;

    Lista(char* registro):Dado(registro)
    {
        nElem = *((unsigned*)&registro[NELEM]);
        nElemInicio = nElem;

    }

    unsigned tamanho()
    {
        unsigned retorno = Dado::tamanho();
        retorno += TAM_INT;

        return retorno;
    }

};
```

```
                 /************************/
/* ************  LISTA                  SIMPLES  ***************** */
                 /************************/


template <class C>
class ListaSimples : public Lista<C>
{
public:

ListaSimples():Lista<C>(){}

ListaSimples(char* registro):Lista<C>(registro)
{

     if( nElem > 0 ){

         primeiro = new ElemLista<C>((C*)&registro[ELEMENTOS]);
         ElemLista<C>* atual = primeiro;

         C* ponteiro = (C*) &registro[ELEMENTOS];

         for(int i=1 ; i< nElem ; i++)
         {
            ponteiro++;
             atual->proximo = new ElemLista<C>(ponteiro);
             atual = atual->proximo;
        }

        ultimo = atual;
     }
}

unsigned tamanho()
{
    tam = Lista<C>::tamanho();
    tam += nElem*sizeof(C);

    return tam;
}

void persista(ofstream* stream)
{
    Lista<C>::persista(stream);
    stream->write((char*)&nElem,TAM_INT);

    if(nElem>0){
        int tamElem = sizeof(C);
        ElemLista<C>* elemAtual = primeiro;
        stream->write((char*)(elemAtual->dado),tamElem);

        for(int i=1 ; i < nElem ; i++){
            elemAtual = elemAtual->proximo;
            stream->write((char*)(elemAtual->dado),tamElem);
        }
    }

}
```

```
};



                    /************************/
/* ************** LISTA                    DADO ***************** */
                    /************************/


template <class D>

class ListaDado : public Lista<D>
{

public:

ListaDado() : Lista<D>()
{}

ListaDado(char* registro):Lista<D>(registro)
{
      if( nElem > 0 ){
            primeiro = new ElemLista<D>(new
D((char*)&registro[ELEMENTOS]));
            ElemLista<D>* atual = primeiro;
            unsigned* ponteiro = (unsigned*) &registro[ELEMENTOS];

            for(int i=1 ; i<nElem ; i++)
            {
                ponteiro = (unsigned*) (((char*)ponteiro) +
*ponteiro);
                atual->proximo = new ElemLista<D>(new
D((char*)ponteiro));
                atual = atual->proximo;
            }
        }
}

unsigned tamanho()
{


    if(nElem>0){

        unsigned retorno = Lista<D>::tamanho();

        ElemLista<D>* elemAtual = primeiro;
        retorno = retorno + (elemAtual->dado)->tamanho();

        for(int i=1 ; i<nElem ; i++)
        {
            elemAtual = elemAtual->proximo;
            retorno += elemAtual->dado->tamanho();
        }
        tam = retorno;
        return retorno;
    }
}
```

```
void persista(ofstream* stream)
{
    Lista<D>::persista(stream);
    stream->write((char*)&nElem,TAM_INT);


    if(nElem>0){
        ElemLista<D>* elemAtual = primeiro;
        (elemAtual->dado)->persista(stream);
        for(int i=1 ; i < nElem ; i++){
            elemAtual = elemAtual->proximo;
            (elemAtual->dado)->persista(stream);
        }
    }

}

};


//-------------------------------------------------------------------------
-------
#endif
```

MATRIZ H

```
#ifndef MatrizH
#define MatrizH

#include "Dado.h"


template<class B>
class Matriz : public Dado{

public:

/* Sequencia do registro = tamanho(int)  tipo(char) dados
dados = linhas(unsigned) colunas(unsigned) nz(unsigned)
celulas(linhas*colunas*nz*T)
*/


Matriz(unsigned numX, unsigned numY) : Dado((char)0)
{
    l = numX;
    c = numY;
    z = 1;
    valores = new B*[l*c*z];

    tamCheio = l*c*z;
    tamCheio = tamCheio%8 ? (tamCheio/8)+1 : tamCheio/8;
    cheio = new char[tamCheio];

    for(int i=0 ; i<tamCheio ; i++){
        cheio[i] = 0;
    }

}

Matriz(unsigned numX,unsigned numY, unsigned numZ) : Dado((char)0)
{
    l = numX;
    c = numY;
    z = numZ;
    valores = new B*[l*c*z];

    tamCheio = l*c*z;
    tamCheio = tamCheio%8 ? (tamCheio/8)+1 : tamCheio/8;

    cheio = new char[tamCheio];

    for(int i=0 ; i<tamCheio ; i++)
        cheio[i] = 0;

}


B* retorna(unsigned linha,unsigned coluna,unsigned Z){

    if( ( linha < l && linha>=0) && (coluna < c && coluna>=0) && (Z <
z && Z>=0)){
        return valores[linha + l*(coluna + c*Z)];
    }
```

```
}

void determina(unsigned linha,unsigned coluna,unsigned Z, B* valor){

    if( ( linha < l && linha>=0) && (coluna < c && coluna>=0) && (Z <
z && Z>=0)){
        int indice = linha+l*(coluna + c*Z);
        valores[indice]  = valor;
        cheio[indice/8] = cheio[(indice)/8] | ( 1 << (indice%8) );

    }
}

bool vazio(unsigned linha, unsigned coluna, unsigned Z)
{
    if( ( linha < l && linha>=0) && (coluna < c && coluna>=0) && (Z <
z && Z>=0)){
        int indice = linha+l*(coluna + c*Z);
        if( cheio[(indice)/8] & ( 1 << indice%8 ) ){
            return false;
        }else{
            return true;
        }
    }
}


unsigned linhas() { return l; }
unsigned colunas() { return c; }
unsigned nZ() { return z; }


protected:

    unsigned l;
    unsigned c;
    unsigned z;

    B** valores;
    char* cheio;
    int tamCheio;

    Matriz(char* registro):Dado(registro)
    {
        l = *((unsigned*)&registro[LINHAS]);
        c = *((unsigned*)&registro[COLUNAS]);
        z = *((unsigned*)&registro[NZ]);

        tamCheio = l*c*z;
        tamCheio = tamCheio%8 ? (tamCheio/8)+1 : tamCheio/8;

        (char(*)[tamCheio])cheio = &registro[CHEIO];

        valores = new B*[l*c*z];


    }

    unsigned tamanho()
    {
```

```
        unsigned retorno = Dado::tamanho();
        retorno += 3*TAM_INT + tamCheio;
    }

};



                /************************/
/* ************   MATRIZ            SIMPLES   ***************** */
                /************************/

template< class D>
class MatrizSimples : public Matriz<D>{

public:

/* Sequencia do registro = tamanho(int)  tipo(char) dados
dados = linhas(unsigned) colunas(unsigned) nz(unsigned)
celulas(linhas*colunas*nz*T)
*/

MatrizSimples(char* registro):Matriz<D>(registro)
{
    D* ponteiro = (D*)&registro[CHEIO + tamCheio];

    for(int i=0 ; i<l*c*z ; i++){
        if( cheio[(i)/8] & ( 1 << i%8 ) ){
            valores[i] = ponteiro;
            ponteiro++;
        }
    }
}


MatrizSimples(unsigned numX, unsigned numY) : Matriz<D>(numX,numY)
{}

MatrizSimples(unsigned numX,unsigned numY, unsigned numZ) :
Matriz<D>(numX,numY,numZ)
{}

unsigned tamanho(){

    int tamD = sizeof(D);
    unsigned retorno = Matriz<D>::tamanho();

    for(int i=0 ; i<l*c*z ; i++){
        if( cheio[(i)/8] & ( 1 << i%8 ) ){
            retorno += tamD;
        }
    }

    tam = retorno;
    return retorno;
}


void persista(ofstream* stream)
{
    Matriz<D>::persista(stream);
```

```
    stream->write((char*)&l,TAM_INT);
    stream->write((char*)&c,TAM_INT);
    stream->write((char*)&z,TAM_INT);
    stream->write(cheio,tamCheio);

    for(int i=0 ; i<l*c*z ; i++){
        if( cheio[(i)/8] & ( 1 << i%8 ) ){
            stream->write((char*)valores[i],sizeof(D));
        }
    }
}


};



                /************************/
/* ************   MATRIZ             DADO    ***************** */
                /************************/

template <class C>
class MatrizDado : public Matriz<C>
{
public:

       MatrizDado(char* registro):Matriz<C>(registro)
       {

            unsigned* ponteiro = (unsigned*) &registro[CHEIO +
tamCheio];

            for(int i=0 ; i<l*c*z ; i++){
                if( cheio[(i)/8] & ( 1 << i%8 ) ){
                    valores[i] = new C((char*)ponteiro);
                    ponteiro = (unsigned*) (((char*)ponteiro) +
*ponteiro);
                }
            }

       }

       MatrizDado(unsigned numX,unsigned numY):Matriz<C>(numX,numY)
       {}

        MatrizDado(unsigned numX,unsigned numY, unsigned
numZ):Matriz<C>(numX,numY,numZ)
        {}


       unsigned tamanho()
       {
            unsigned retorno = Matriz<C>::tamanho();

            for(int i=0 ; i<l*c*z ; i++){
                if( cheio[(i)/8] & ( 1 << i%8 ) ){
                    retorno += valores[i]->tamanho();
                }
            }

            tam = retorno;
```

```
            return retorno;
        }

    void persista(ofstream* stream)
    {
        Matriz<C>::persista(stream);
        stream->write((char*)&l,TAM_INT);
        stream->write((char*)&c,TAM_INT);
        stream->write((char*)&z,TAM_INT);
        stream->write(cheio,tamCheio);

        for(int i=0 ; i<l*c*z ; i++){
            if( cheio[(i)/8] & ( 1 << i%8 ) ){
                valores[i]->persista(stream);
            }
        }
    }


};

//-----------------------------------------------------------------------
-------
#endif
```

TAB HASH GEOM H

```
#ifndef TabHashGeomH
#define TabHashGeomH

#include <fstream>
#include <iostream>

#include "Matriz.h"
#include "ModeloFace.h"
#include "Lista.h"
#include "Gauss.h"
#include "EntradaBucket.h"
#include "Dado.h"

using namespace std;
class TabHashGeom : public DadoComplexo{
public:
      TabHashGeom(long int novoOID);
    TabHashGeom(char* registro);

      int retornaIndice(ListaDado<EntradaBucket>* lista, long int
oidIndividuo);
      void adiciona(ModeloFace* modelo, long int oidAlvo);
      ListaDado<EntradaBucket>* retornaCandidatos(ModeloFace* modelo);

    void persista(ofstream* stream);
    unsigned tamanho();



private:
    unsigned maxCand;

      Gauss gauss;
    MatrizDado<ListaDado<EntradaBucket> >* tabela;

    double* executaFuncaoHash(MatrizSimples<double>* A,double*
pontoCaracteristica);

};



//---------------------------------------------------------------------
-------
#endif
```

TAB HASH GEOM CPP

```
#include "TabHashGeom.h"

TabHashGeom::TabHashGeom(long int novoOID):DadoComplexo(novoOID,30)
{
      tabela = new MatrizDado<ListaDado<EntradaBucket> >(150,150,150);
}

TabHashGeom::TabHashGeom(char* registro):DadoComplexo(registro)
{
    unsigned* ponteiro = (unsigned*) &registro[DADOS];
    maxCand = *ponteiro;
    ponteiro++;

    tabela = new MatrizDado<ListaDado<EntradaBucket>
>((char*)ponteiro);

}

void TabHashGeom::persista(ofstream* stream)
{
    DadoComplexo::persista(stream);

    stream->write((char*)&maxCand,TAM_INT);
    tabela->persista(stream);
}

unsigned TabHashGeom::tamanho()
{
    unsigned retorno = DadoComplexo::tamanho();
    retorno += TAM_INT;
    retorno += tabela->tamanho();

    return retorno;
}


void TabHashGeom::adiciona(ModeloFace* modelo, long int oidAlvo)
{
      ListaDado<EntradaBucket>* lista;
      EntradaBucket* entrada;

    double* ponto;

    int totPontosCarac =modelo->pontosCaracteristicas->colunas();
      double** bucket = new double*[totPontosCarac];
    double peso = (double)1/(totPontosCarac-2);
      int indice;

      for(int i=2;i<totPontosCarac;i++)
      {
        ponto = (double*)modelo->pontosCaracteristicas-
>retorna(0,i,0);


            bucket[i] = executaFuncaoHash(modelo-
>matrizFuncaoHash,ponto);
            ((bucket[i])[0]<0) ? ((bucket[i])[0]= -1*(bucket[i])[0]) :
(bucket[i])[0]= (bucket[i])[0];
```

```
            ((bucket[i])[1]<0) ? ((bucket[i])[1]= -1*(bucket[i])[1]) :
(bucket[i])[1]= (bucket[i])[1];
            ((bucket[i])[2]<0) ? ((bucket[i])[2]= -1*(bucket[i])[2]) :
(bucket[i])[2]= (bucket[i])[2];
          if( ((bucket[i])[0] < 150 ) && ( (bucket[i])[1] < 150 ) && (
(bucket[i])[2] <150 ))
            {
            if( (tabela-
>vazio((int)(bucket[i])[0],(int)(bucket[i])[1],(int)(bucket[i])[2]))
){
                    tabela-
>determina((int)(bucket[i])[0],(int)(bucket[i])[1],(int)(bucket[i])[2]
,new ListaDado<EntradaBucket>());
            }
            lista = tabela-
>retorna((int)(bucket[i])[0],(int)(bucket[i])[1],(int)(bucket[i])[2]);
            indice = retornaIndice(lista,oidAlvo);

            //Caso indice == -1 entao nao ha nenhuma ocorrencia de oid
            if( indice == -1 ){
                entrada = new EntradaBucket(oidAlvo,peso);
                lista->adiciona(entrada);
            }else{
                entrada = lista->retorna(indice);
                entrada->votoSimples(peso);
            }
        }else{
            cout << "MAIOR 150" << endl;
        }
    }

}


ListaDado<EntradaBucket>* TabHashGeom::retornaCandidatos(ModeloFace*
modelo)
{

      ListaDado<EntradaBucket>* lista;
      ListaDado<EntradaBucket>* candidatosGerais = new
ListaDado<EntradaBucket>();

      EntradaBucket* entrada;

    double* bucket;
      int indice;


    for(int i=2;i<modelo->pontosCaracteristicas->colunas();i++)
      {

       bucket = executaFuncaoHash(modelo-
>matrizFuncaoHash,(double*)modelo->pontosCaracteristicas-
>retorna(0,i,0));

         (bucket[0]<0) ? (bucket[0]= -1*bucket[0]) : bucket[0]=
bucket[0];
         (bucket[1]<0) ? (bucket[1]= -1*bucket[1]) : bucket[1]=
bucket[1];
         (bucket[2]<0) ? (bucket[2]= -1*bucket[2]) : bucket[2]=
bucket[2];
```

```
            if( (bucket[0] < 150 ) && ( bucket[1] < 150 ) && (
bucket[2] <150 )
            && !(tabela-
>vazio((int)bucket[0],(int)bucket[1],(int)bucket[2]))){

                  lista = tabela-
>retorna((int)bucket[0],(int)bucket[1],(int)bucket[2]);


                  // lista contem os votos, para cada voto
               for(int j=0 ; j<lista->retornaNElem() ; j++)
                  {
                        // caso o oid ja esteja em candidatos gerais
...
                        indice = retornaIndice(candidatosGerais,(lista-
>retorna(j))->oid);

                        if(indice == -1)
                        {
                              entrada = new EntradaBucket((lista-
>retorna(j))->oid);
                              entrada->peso += ( (lista->retorna(j))-
>peso );
                              candidatosGerais->adiciona(entrada);

                        }else{
                              entrada = candidatosGerais-
>retorna(indice);
                              entrada->peso += ( (lista->retorna(j))-
>peso );
                        }
                  }
            }
    }

      return candidatosGerais;

}

int TabHashGeom::retornaIndice(ListaDado<EntradaBucket>* lista, long
int oidIndividuo)
{

      for(int i=0; i<lista->retornaNElem() ; i++)
      {
            if( (lista->retorna(i))->oid == oidIndividuo )
                  return i;
      }
      return -1;

}

double* TabHashGeom::executaFuncaoHash(MatrizSimples<double>*
A,double* pontoCaracteristica)
{

    // Coloca os dados do ponto caracteristica na quarta coluna de A
    for(int i=0;i<3;i++)
      {
        A->determina(i,3,0,new double(pontoCaracteristica[i]));
```

```
        }
    return gauss.aplicaGaussPivotacaoParcial(A);
}
```

MODELO FACE H

```
#ifndef ModeloFaceH
#define ModeloFaceH

#include "Matriz.h"
#include "Dado.h"



class ModeloFace : public Dado
{

public:

    MatrizSimples<double>* face;
    MatrizSimples<double[3]>* pontosCaracteristicas;
    MatrizSimples<double>* matrizFuncaoHash;

    void persista(ofstream* stream);
    unsigned tamanho();

    ModeloFace(char* registro);

private:
    friend class PreProcessamento;



      ModeloFace(MatrizSimples<double>* face);

};
//---------------------------------------------------------------------
-------
#endif
```

MODELO FACE CPP

```
#include "ModeloFace.h"
//-----------------------------------------------------------------------
-------
ModeloFace::ModeloFace(MatrizSimples<double>* face):Dado(20)
{
    ModeloFace::face = face;
}

/* Sequencia do registro = tamanho(int)  tipo(char) oid(long) dados
dados = face pontosCaracteristicas matrizFuncaoHash
*/

ModeloFace::ModeloFace(char* registro):Dado(registro)
{
    unsigned* ponteiro = (unsigned*) &registro[DADOS];

    face = new MatrizSimples<double>((char*)ponteiro);
    ponteiro = (unsigned*) (((char*)ponteiro) + *ponteiro);

    pontosCaracteristicas = new
MatrizSimples<double[3]>((char*)ponteiro);
    ponteiro = (unsigned*) (((char*)ponteiro) + *ponteiro);

    matrizFuncaoHash = new MatrizSimples<double>((char*)ponteiro);

}

void ModeloFace::persista(ofstream* stream)
{
    Dado::persista(stream);

    face->persista(stream);
    pontosCaracteristicas->persista(stream);
    matrizFuncaoHash->persista(stream);

}

unsigned ModeloFace::tamanho()
{
    unsigned retorno = Dado::tamanho();
    retorno += face->tamanho();
    retorno += pontosCaracteristicas->tamanho();
    retorno += matrizFuncaoHash->tamanho();

    return retorno;
}
```

PRE PROCESSAMENTO H

```
#ifndef PreProcessamentoH
#define PreProcessamentoH

#include "ModeloFace.h"
#include "CacadorDeCaracteristicas.h"
#include "Matriz.h"
#include "Operacoes.h"
#include <math.h>
#include <iostream>

using namespace std;

class PreProcessamento{

public:
        PreProcessamento();
        ModeloFace* criaModelo(MatrizSimples<double>* face);
        void alteraLimite(double novoLimite);
            double retornaLimite();

private:

        CacadorDeCaracteristicas cacador;
        Operacoes operacoes;

        void preparaMatrizFuncaoHash(ModeloFace* modelo);
        void executaPreProcessamento(ModeloFace* modelo);
        void reposicionaEescalona(MatrizSimples<double[3]>*
caracteristicas);
        double* determinaPontoCentral(double* A,double* B);

};




//-------------------------------------------------------------------
-------
#endif
```

PRE PROCESSAMENTO CPP

```
#include "PreProcessamento.h"

PreProcessamento::PreProcessamento()
{
    //oidAtual = 0;

}

void PreProcessamento::alteraLimite(double novoLimite)
{
      cacador.limite = novoLimite;

}

double PreProcessamento::retornaLimite()
{
      return cacador.limite;
}


ModeloFace* PreProcessamento::criaModelo(MatrizSimples<double>* face)
{

    //oidAtual++;
    //long int novoOID = oidAtual;
    ModeloFace* modelo = new ModeloFace(face);

    executaPreProcessamento(modelo);

    return modelo;
}

void PreProcessamento::executaPreProcessamento(ModeloFace* modelo)
{


    modelo->pontosCaracteristicas =
cacador.buscaCaracteristicas(modelo->face);

    reposicionaEescalona(modelo->pontosCaracteristicas);
      preparaMatrizFuncaoHash(modelo);



}

void PreProcessamento::preparaMatrizFuncaoHash(ModeloFace* modelo)
{
    MatrizSimples<double>* matriz = new MatrizSimples<double>(3,4);
    double* baseSimples = (double*)modelo->pontosCaracteristicas-
>retorna(0,0,0);
    double* baseRotZ = new double[3];
    double* baseRotY = new double[3];

    // Valor apontado por baseRot = valor apontado por modelo-
>retorna(0,0)
    *baseRotZ = *baseSimples;
```

```
    *baseRotY = *baseSimples;

    operacoes.rotaciona(baseRotZ,0,0,M_PI_2);
    operacoes.rotaciona(baseRotY,0,M_PI_2,0);

    for(int i=0;i<3;i++)
    {
        matriz->determina(i,0,0,new double(baseSimples[i]));
        matriz->determina(i,1,0,new double(baseRotZ[i]));
        matriz->determina(i,2,0,new double(baseRotY[i]));
    }

    modelo->matrizFuncaoHash = matriz;
}

double* PreProcessamento::determinaPontoCentral(double* A,double* B)
{
    double* pontoCentral = new double[3];

    // x
    pontoCentral[0] = A[0] + (B[0]-A[0])/2;

    // y
    pontoCentral[1] = A[1] + (B[1]-A[1])/2;

    // z
    pontoCentral[2] = A[2] + (B[2]-A[2])/2;

      return pontoCentral;


}

void PreProcessamento::reposicionaEescalona(MatrizSimples<double[3]>*
caracteristicas)
{
    double* baseA = (double*)caracteristicas->retorna(0,0,0);
    double* baseB = (double*)caracteristicas->retorna(0,1,0);
    double* pontoCentral;

    // Determinacao do ponto Central da Base
    pontoCentral = determinaPontoCentral(baseA,baseB);

    // TRANSLADA PONTO CENTRAL DA BASE PARA ORIGEM
    operacoes.translada(caracteristicas,-pontoCentral[0],-
pontoCentral[1],-pontoCentral[2]);

    // Determinacao do angulo de rotacao em torno de Y
    double rotacaoY = (baseB[2] - baseA[2])/(baseB[0] - baseA[0]);
      rotacaoY = atan(rotacaoY);

    // ROTACIONA PONTOS CARACTERISTICAS EM TORNO DE Y
    operacoes.rotaciona(caracteristicas,0,rotacaoY,0);

    // Determinacao do fator de escala
    double* vetor = new double[3];
    vetor[0] = baseB[0]-baseA[0];
    vetor[1] = baseB[1]-baseA[1];
    vetor[2] = baseB[2]-baseA[2];

    double norma = sqrt(vetor[0]*vetor[0] + vetor[1]*vetor[1] +
```

```
vetor[2]*vetor[2]);

    double escala =  1 - norma;


    // ESCALONA PONTOS CARACTERISTICAS
    operacoes.escalona(caracteristicas,escala,0,0);

      pontoCentral = determinaPontoCentral(baseA,baseB);

}
```

CACADOR DE CARACTERISTICAS H

```
#ifndef CacadorDeCaracteristicasH
#define CacadorDeCaracteristicasH


#include "Matriz.h"
#include "Operacoes.h"
#include "Lista.h"
#include <iostream>


class CacadorDeCaracteristicas{
public:

    CacadorDeCaracteristicas();


    MatrizSimples<double[3]>*
buscaCaracteristicas(MatrizSimples<double>* face);

private:
      friend class PreProcessamento;
    double montaVarianciaPonto(MatrizSimples<double>* face, int linha,
int coluna);
      double limite;

};
//---------------------------------------------------------------------
-------
#endif
```

CACADOR DE CARACTERISTICAS CPP

```
#include "CacadorDeCaracteristicas.h"

CacadorDeCaracteristicas::CacadorDeCaracteristicas(){}


MatrizSimples<double[3]>*
CacadorDeCaracteristicas::buscaCaracteristicas(MatrizSimples<double>*
face)
{
    ListaSimples<double[3]>* lista = new ListaSimples<double[3]>();

    // Com a variancia montada passamos ao processo de deteccao das
caracteristicas

    // Espacamento das linhas e colunas para tres regioes
    int espacamentoLinhas = (face->linhas() -2)/3;
    int espacamentoColunas =(face->colunas() -2)/3;

    // Limite de duas vezes um espacamento de linha
    int limiteLinhas  = espacamentoLinhas*2;

    // Determinacao da coluna e linha de centro da face
    int colunaCentro = face->colunas()/2;
    int linhaCentro = 0;

    double* coordenada = new double[3];

    // Determinacao da linha de centro
    for( linhaCentro ;linhaCentro < face->linhas();linhaCentro++)
    {
        if( *face->retorna(linhaCentro,colunaCentro,0) == 50 )
                break;
    }

    //************* BASE A  BASE B ***************

    // ATENCAO: metodo "retorna" da matriz retorna ponteiro
    coordenada[0] = colunaCentro-1;
    coordenada[1] = linhaCentro;
    coordenada[2] = * face->retorna(linhaCentro,colunaCentro-1,0);

    // eh necessario cast pois pretende-se inserir um ponteiro para
double[3]
    lista->adiciona((double(*)[3])coordenada);

    coordenada = new double[3];

    coordenada[0] = colunaCentro+1;
    coordenada[1] = linhaCentro;
    coordenada[2] = * face->retorna(linhaCentro,colunaCentro+1,0);

    lista->adiciona((double(*)[3])coordenada);

    //************* FIM BASE A BASE B ***************

    // Determina as regioes e os menores elementos
    // vai ateh o limite de linhas e ateh o total de colunas,
    // isto corresponde ateh a regiao onde comeca a boca
```

```
    // eh descartada a regiao da boca pois contem muito ruido natural
(ex. barba)


    for(int i=2;i<limiteLinhas;i++)
    {
      for(int j=2;j<face->colunas()-2;j++)
      {
          if( montaVarianciaPonto(face,i,j) <= limite )
            {
                        //cout << j << "  " << i << "  " << * face-
>retorna(i,j,0) << endl;
                        coordenada = new double[3];
                        coordenada[0] = j; // x
                        coordenada[1] = i; // y
                        coordenada[2] = * face->retorna(i,j,0); // z
                        lista->adiciona((double(*)[3])coordenada);
            }
        }
     }

    // Com a deteccao concluida montamos uma matriz das
caracteristicas
    // Os dois primeiros pontos formam a BASE
    MatrizSimples<double[3]>* matrizRetorno = new
MatrizSimples<double[3]>(1,lista->retornaNElem());
    for(int i=0; i<lista->retornaNElem() ;i++)
    {
        matrizRetorno->determina(0,i,0,(double(*)[3])lista-
>retorna(i));
      }

    return matrizRetorno;
}


double
CacadorDeCaracteristicas::montaVarianciaPonto(MatrizSimples<double>*
face, int linha, int coluna)
{


    // Variancia = SOMATORIO( (elemContorno - elemAtual)^2 ) / 8
    double elemAtual = 0;
      double elemContornoAtual = 0;
    double somatorio = 0;

    // diferenca = elemContorno - elemAtual
    double diferenca = 0;


    // i e j determinam a posicao do elemAtual na matriz de dados da
face
    // sao necessarios tantos for pois necessitasse capturar os
elementos de contorno

    elemAtual = * face->retorna(linha,coluna,0);

    // PREPARA SOMATORIOS : w e z determinam posicao dos 8 contornos
de elemAtual
    for(int w=linha-2;w<=linha+2;w++)
```

```
    {
    for(int z=coluna-2;z<=coluna+2;z++)
    {
      if( (w!=linha) && (z!=coluna) ){
              elemContornoAtual = (* face->retorna(w,z,0));
          diferenca =  elemContornoAtual - elemAtual;
          somatorio += (diferenca * diferenca);
          }
      }
    }

    // i-1 e j-1 pois estamos nos elementos internos da face
    somatorio = somatorio/24;
    return somatorio;
}
```

OPERACOES H

```
#ifndef OperacoesH
#define OperacoesH
#include "Matriz.h"

#include <iostream>
using namespace std;


class Operacoes{
public:
    Operacoes();

    // OPERACOES GRAFICAS SOBRE OBJETOS INTEIROS : REFERENCIAS
PRESERVADAS
    void translada(MatrizSimples<double[3]>* objeto,double x, double
y,double z);
    void escalona(MatrizSimples<double[3]>* objeto,double x,double y,
double z);
    void rotaciona(MatrizSimples<double[3]>* objeto,double  x,double
y,double z);

    //OPERACOES GRAFICAS SOBRE PONTOS : REFERENCIA PRESERVADA
    void translada(double* ponto,double x, double y,double z);
    void escalona(double* ponto,double x,double y, double z);
    void rotaciona(double* ponto,double  x,double y,double z);



private:

        // MULTIPLICACAO DE PONTO : REFERENCIA AO PONTO NAO E ALTERADA
        double* multiplicacaoGrafica(double*
ponto,MatrizSimples<double>* B);

    // MULTIPLICACAO DE OBJETO : REFERENCIAS A PONTOS DO OBJETO NAO
SAO ALTERADAS
    void multiplicacaoGrafica(MatrizSimples<double[3]>* objeto,
MatrizSimples<double>* B);

    void preparaMatrizTranslacao(double x, double y, double z);
    void preparaMatrizEscalonamento(double x, double y, double z);
    void preparaMatrizRotacao(double x, double y, double z);


    MatrizSimples<double>* translacao;
    MatrizSimples<double>* escalonamento;
    MatrizSimples<double>* rotacionamentoX;
    MatrizSimples<double>* rotacionamentoY;
    MatrizSimples<double>* rotacionamentoZ;

};

//-----------------------------------------------------------------
-------#endif
```

OPERACOES CPP

```
#include "Operacoes.h"

using namespace std;
//-----------------------------------------------------------------------
-------

Operacoes::Operacoes()
{
    translacao = new MatrizSimples<double>(4,4);
    escalonamento = new MatrizSimples<double>(4,4);
    rotacionamentoX = new MatrizSimples<double>(4,4);
    rotacionamentoY = new MatrizSimples<double>(4,4);
    rotacionamentoZ = new MatrizSimples<double>(4,4);

    for(int i=0;i<4;i++){
        for(int j=0;j<4;j++){
            if(i!=j){
                double aux = 0.0;
                translacao->determina(i,j,0,new double(aux));
                escalonamento->determina(i,j,0,new double(aux));
                rotacionamentoX->determina(i,j,0,new double(aux));
                rotacionamentoY->determina(i,j,0,new double(aux));
                rotacionamentoZ->determina(i,j,0,new double(aux));
            }else{
                double aux = 1.0;
                translacao->determina(i,j,0,new double(aux));
                escalonamento->determina(i,j,0,new double(aux));
                rotacionamentoX->determina(i,j,0,new double(aux));
                rotacionamentoY->determina(i,j,0,new double(aux));
                rotacionamentoZ->determina(i,j,0,new double(aux));
            }
        }
    }

}

// MULTIPLICACAO DE OBJETO : REFERENCIAS A PONTOS DO OBJETO NAO SAO
ALTERADAS
void Operacoes::multiplicacaoGrafica(MatrizSimples<double[3]>* objeto,
MatrizSimples<double>* B)
{
    double* novo;
    double* A;

    // Para cada celula da matriz "objeto" multiplicar com matriz "B"

    for(int i=0;i<objeto->linhas();i++)
    {
        for(int j=0;j<2;j++)
        {
            A = (double*)objeto->retorna(i,j,0);

            novo = multiplicacaoGrafica(A,B);

            // NOTE: nao altera endereco de A
            A[0] = novo[0];
```

```
            A[1] = novo[1];
            A[2] = novo[2];


        }
    }
}

// MULTIPLICACAO DE PONTO : REFERENCIA AO PONTO NAO E ALTERADA
double* Operacoes::multiplicacaoGrafica(double*
ponto,MatrizSimples<double>* B)
{
    double soma = 0;
    double* novo = new double[3];

    for(int coluna=0;coluna<4;coluna++)
    {
        for(int colunaLinha = 0; colunaLinha<3;colunaLinha++)
        {
            soma += (ponto[colunaLinha])*(*(B-
>retorna(colunaLinha,coluna,0)));
        }
        soma += *(B->retorna(3,coluna,0));
        novo[coluna] = soma;
            soma = 0;
    }
    return novo;

}

//***** TRANSLACAO *****//

void Operacoes::preparaMatrizTranslacao(double x, double y, double z)
{

    translacao->determina(3,0,0,new double(x));
    translacao->determina(3,1,0,new double(y));
    translacao->determina(3,2,0,new double(z));
}

// OBJETO
void Operacoes::translada(MatrizSimples<double[3]>* objeto,double x,
double y,double z)
{
    preparaMatrizTranslacao(x,y,z);
    multiplicacaoGrafica(objeto,translacao);
}

// PONTO
void Operacoes::translada(double* ponto,double x, double y,double z)
{
    double* novo = new double[3];
    preparaMatrizTranslacao(x,y,z);
    novo = multiplicacaoGrafica(ponto,translacao);
    ponto[0] = novo[0];
    ponto[1] = novo[1];
    ponto[2] = novo[2];
}


//***** ESCALONAMENTO *****//
```

```
void Operacoes::preparaMatrizEscalonamento(double x, double y, double
z)
{
    escalonamento->determina(0,0,0,new double(x));
    escalonamento->determina(1,1,0,new double(y));
    escalonamento->determina(2,2,0,new double(z));
}

// OBJETO
void Operacoes::escalona(MatrizSimples<double[3]>* objeto,double
x,double y, double z)
{
    preparaMatrizEscalonamento(x,y,z);
    multiplicacaoGrafica(objeto,escalonamento);
}

// PONTO
void Operacoes::escalona(double* ponto,double x,double y, double z)
{
    double* novo = new double[3];

    preparaMatrizEscalonamento(x,y,z);
    novo = multiplicacaoGrafica(ponto,escalonamento);
    ponto[0] = novo[0];
    ponto[1] = novo[1];
    ponto[2] = novo[2];
}


//***** ROTACIONAMENTO *****//

void Operacoes::preparaMatrizRotacao(double x, double y, double z)
{
    double coseno;
    double seno;


    if(x!=0)
    {
        coseno = cos(x);
        seno = sin(x);

        rotacionamentoX->determina(1,1,0,new double(coseno));
        rotacionamentoX->determina(2,2,0,new double(coseno));
        rotacionamentoX->determina(2,1,0,new double(-seno));
        rotacionamentoX->determina(1,2,0,new double(seno));
    }
    if(y!=0)
    {
        coseno = cos(y);
        seno = sin(y);

            rotacionamentoY->determina(0,0,0,new double(coseno));
        rotacionamentoY->determina(2,2,0,new double(coseno));
        rotacionamentoY->determina(2,0,0,new double(seno));
        rotacionamentoY->determina(0,2,0,new double(-seno));
    }
    if(z!=0)
    {
        coseno = cos(z);
```

```
        seno = sin(z);

        rotacionamentoZ->determina(0,0,0,new double(coseno));
        rotacionamentoZ->determina(1,1,0,new double(coseno));
        rotacionamentoZ->determina(1,0,0,new double(-seno));
        rotacionamentoZ->determina(0,1,0,new double(seno));
    }



}

// OBJETO
void Operacoes::rotaciona(MatrizSimples<double[3]>* objeto,double
x,double y,double z)
{
    preparaMatrizRotacao(x,y,z);

    if(x!=0)
        multiplicacaoGrafica(objeto,rotacionamentoX);
    if(y!=0)
        multiplicacaoGrafica(objeto,rotacionamentoY);
    if(z!=0)
        multiplicacaoGrafica(objeto,rotacionamentoZ);
}


// PONTO
void Operacoes::rotaciona(double* ponto,double  x,double y,double z)
{
    double* novo = new double[3];

    preparaMatrizRotacao(x,y,z);

    if(x!=0)
    {
        novo = multiplicacaoGrafica(ponto,rotacionamentoX);
        ponto[0] = novo[0];
        ponto[1] = novo[1];
        ponto[2] = novo[2];
    }
    if(y!=0)
    {
        novo = multiplicacaoGrafica(ponto,rotacionamentoY);
        ponto[0] = novo[0];
        ponto[1] = novo[1];
        ponto[2] = novo[2];
    }
    if(z!=0)
    {
        novo = multiplicacaoGrafica(ponto,rotacionamentoZ);
        ponto[0] = novo[0];
        ponto[1] = novo[1];
        ponto[2] = novo[2];
    }
}
```

GAUSS H

```
#include "Matriz.h"

#include <iostream>
using namespace std;

class Gauss
{
public:
    double* aplicaGaussPivotacaoParcial(MatrizSimples<double>* A);

private:
    void pivotacaoParcial(MatrizSimples<double>* A,int k);
    void triangularizacao(MatrizSimples<double>* A,int k);
    double* retrosubstituicao(MatrizSimples<double>* A);

};
```

GAUSS CPP

```
#include "Gauss.h"

double* Gauss::aplicaGaussPivotacaoParcial(MatrizSimples<double>* A)
{


    for(int i=0;i<(A->linhas())-1;i++)
    {
        pivotacaoParcial(A,i);
        triangularizacao(A,i);
    }

    return retrosubstituicao(A);

}

void Gauss::pivotacaoParcial(MatrizSimples<double>* A,int k)
{
    // Sempre lembrar, Matriz->retorna um ponteiro

    double max = *(A->retorna(k,k,0));
    (max<0) ? (max = -1*max) : max = max;
    int lMax = k;
    double aux;

    // Escolha do maior pivo
    for( int i=k+1 ; i < (A->linhas()) ; i++)
    {
        aux = *(A->retorna(i,k,0));
        (aux<0) ? (aux = -1*aux) : aux = aux;

        if( aux > max )
        {
            max = aux;
            lMax = i;
        }
    }

    for( int j=k ; j < (A->linhas())+1 ; j++)
    {
        aux = *(A->retorna(k,j,0));
        A->determina(k,j,0,A->retorna(lMax,j,0));
        A->determina(lMax,j,0,new double(aux));
    }
}



void Gauss::triangularizacao(MatrizSimples<double>* A,int k)
{
    double aux,divisao,elemCanc;
      double pivo = *(A->retorna(k,k,0));

      for(int i=k+1 ; i < (A->linhas()) ; i++)
    {
        // (elemento a ser cancelado) / (pivo)
```

```
            elemCanc = *(A->retorna(i,k,0));
            divisao = (elemCanc) / (pivo);

        for(int j=k+1 ; j < (A->linhas())+1 ; j++)
        {
            aux = (*(A->retorna(i,j,0))) - (divisao*(*(A-
>retorna(k,j,0))));
            A->determina(i,j,0,new double(aux));
        }
        aux = 0;
        A->determina(i,k,0,new double(aux));
    }

}

double* Gauss::retrosubstituicao(MatrizSimples<double>* A)
{
    int linhasA = (A->linhas());
    double* retorno = new double[linhasA];
    double soma = 0;

      retorno[linhasA-1] = *(A->retorna(linhasA-1,linhasA,0))/(*(A-
>retorna(linhasA-1,linhasA-1,0)));



      for(int i=linhasA-2 ; i >= 0 ; i--)
    {
        soma = 0;
        for( int j=i+1 ; j<linhasA ; j++ )
        {
            soma = soma + (*(A->retorna(i,j,0)))*retorno[j];
        }
        retorno[i] = ((*(A->retorna(i,linhasA,0))) - soma)/(*(A-
>retorna(i,i,0)));
    }

    return retorno;

}
```

ENTRADA BUCKET H

```
#ifndef EntradaBucketH
#define EntradaBucketH

#include "Dado.h"
#include <fstream>
#include <iostream>

class EntradaBucket:public Dado
{

      public:
            double peso;
        long int oid;

        EntradaBucket(long int oid);
            EntradaBucket(long int oid, double valor);
        EntradaBucket(char* registro);

        void votoSimples(double valor);

        void persista(ofstream* stream);
        unsigned tamanho();









};



#endif
```

ENTRADA BUCKET CPP

```
#include "EntradaBucket.h"
using namespace std;

EntradaBucket::EntradaBucket(long int oid):Dado((char)10)
{
    peso = 0;
    EntradaBucket::oid = oid;
}

EntradaBucket::EntradaBucket(long int oid, double
valor):Dado((char)10)
{
      peso = valor;
    EntradaBucket::oid = oid;
}

EntradaBucket::EntradaBucket(char* registro):Dado(registro)
{
    peso = *((double*)&registro[OID]);
    oid = *((long*)&registro[OID + TAM_DOUBLE]);
}


void EntradaBucket::persista(ofstream* stream)
{
    Dado::persista(stream);
    stream->write((char*)&peso,TAM_DOUBLE);
    stream->write((char*)&oid,TAM_LONG);

}

unsigned EntradaBucket::tamanho()
{
    unsigned retorno = Dado::tamanho();
    retorno += TAM_DOUBLE + TAM_LONG;

}

void EntradaBucket::votoSimples(double valor)
{
    if(valor > peso ){
        peso = valor;
    }
}
```

SA H

```
#ifndef SAH
#define SAH

#include "Dado.h"
#include "Lista.h"
#include "Matriz.h"
#include "Individuo.h"


#include <iostream>
#include <fstream>

class EntradaDicionario:public Dado
{

public:
    EntradaDicionario(long int oid, long int deslocamento);
    EntradaDicionario(char* registro);

    void persista(ofstream* stream);
    unsigned tamanho();

    long int oid;
    long unsigned deslocamento;

};


class SA
{

public:
    SA();


    Individuo* retornaInstanciaIndividuo(long int oid);
    Individuo* retornaNovaInstanciaIndividuo(long int oid);
    void novaInstanciaIndividuo(long int oid,Individuo* ponteiro,bool
Ehnovo);

    void inicializaSistema();
    void sairSistema();


private:
    MatrizDado<ListaDado<EntradaDicionario> >* dicionario;

    int tamMemoria;

    // Array com os oid's dos objetos em memoria
    long int* memoria;

    // Array com os enderecos dos objetos
    Individuo** ponteiros;
    // Flag indicando se o dado eh novo ou antigo
    bool* novo;
    // Tamanho antigo do objeto
    unsigned* tamanhoAntigo;
```

```
    ofstream* arquivoIndividuos;
    unsigned deslocIndividuos;
    unsigned tamanhoIndividuos;


    ifstream* arquivoIndividuosIn;
    unsigned deslocIndividuosIn;



};


#endif
```

SA CPP

```
#include "SA.h"


EntradaDicionario::EntradaDicionario(long int oid, long int
deslocamento):Dado((char)0)
{
    EntradaDicionario::oid = oid;
    EntradaDicionario::deslocamento = deslocamento;
}


EntradaDicionario::EntradaDicionario(char* registro):Dado(registro)
{
    oid = *((long int*)&registro[OID]);
    deslocamento = *((long int*)&registro[OID+TAM_LONG]);
}

void EntradaDicionario::EntradaDicionario::persista(ofstream* stream)
{
    Dado::persista(stream);
    stream->write((char*)&oid,TAM_LONG);
    stream->write((char*)&deslocamento,TAM_LONG);
}

unsigned EntradaDicionario::EntradaDicionario::tamanho()
{
    unsigned retorno = Dado::tamanho();
    retorno += 2*TAM_LONG;
}



SA::SA()
{
    dicionario = new MatrizDado<ListaDado<EntradaDicionario> >(15,1);

    arquivoIndividuos = new ofstream("individuos.dad",ios::binary);
    arquivoIndividuosIn = new ifstream("individuos.dad",ios::binary);

    deslocIndividuos = 0;
    deslocIndividuosIn = 0;

    tamMemoria = 10;
    memoria = new long int[tamMemoria];
    for(int i=0;i<tamMemoria;i++)
        memoria[i] = 0;

    ponteiros = new Individuo*[tamMemoria];
    novo = new bool[tamMemoria];
    tamanhoAntigo = new unsigned[tamMemoria];


}


void SA::novaInstanciaIndividuo(long int oid,Individuo* ponteiro,bool
Ehnovo)
{
```

```
    unsigned deslocamento = 0;
    unsigned localizacaoMemoria = oid%tamMemoria;
    unsigned localizacaoDicionario = oid%dicionario->linhas();

    // Se a memoria nesta localizacao nao esta vazia, eh preciso
persistir
    if( memoria[localizacaoMemoria] != 0 )
    {
        unsigned tamanho = 0;

        tamanho = (ponteiros[localizacaoMemoria])->tamanho();


        // Se o tamanho atual eh diferente do tamanho antigo,
significa que o dado foi alterado
        // Se o dado eh novo , nos dois casos deve ser armazenado ao
final do arquivo
        if( (tamanho != tamanhoAntigo[localizacaoMemoria]) ||
(novo[localizacaoMemoria]) )
        {

             cout << "MEMORIA OCUPADA " << endl;
            // Grava ao final do arquivo
            arquivoIndividuos->seekp(0,ios::end);
            (ponteiros[localizacaoMemoria])-
>persista(arquivoIndividuos);
             cout << "MEMORIA OCUPADA " << endl;
            // Atualiza no dicionario de dados
            if( dicionario->vazio(localizacaoDicionario,0,0) )
                dicionario->determina(localizacaoDicionario,0,0,new
ListaDado<EntradaDicionario>());

            ListaDado<EntradaDicionario>* listaDicionario =
dicionario->retorna(localizacaoDicionario,0,0);


            // Se eh dado novo, adiciona uma nova entrada no
dicionario de dados para o objeto
            if(novo[localizacaoMemoria])
            {

                listaDicionario->adiciona(new
EntradaDicionario(memoria[localizacaoMemoria],tamanhoIndividuos));
            }else{
                // Caso nao seja novo, entao eh necessario atualizar o
deslocamento do objeto

                for( int i=0;i<listaDicionario->retornaNElem();i++)
                {
                    if(((EntradaDicionario*)listaDicionario-
>retorna(i))->oid == oid)
                    {
                        ((EntradaDicionario*)listaDicionario-
>retorna(i))->deslocamento = tamanhoIndividuos;

                        i = listaDicionario->retornaNElem();
                    }
                }
            }
```

```
            cout << "AAAA" << endl;
            tamanhoIndividuos += tamanho;
            deslocIndividuos = tamanhoIndividuos;

        }else{
            cout << "TAMANHO IGUAL" << endl;
            ListaDado<EntradaDicionario>* listaDicionario =
dicionario->retorna(oid%dicionario->linhas(),0,0);
            // Caso nao haja nenhum objeto em memoria que ocupe este
bucket ...
            for( int i=0;i<listaDicionario->retornaNElem();i++)
            {
                if(((EntradaDicionario*)listaDicionario->retorna(i))-
>oid == oid)
                {
                    deslocamento =
((EntradaDicionario*)listaDicionario->retorna(i))->deslocamento;

                    i = listaDicionario->retornaNElem();
                }
            }

            deslocIndividuos = deslocamento - deslocIndividuos;
            arquivoIndividuos->seekp(deslocIndividuos,ios::beg);
            (ponteiros[localizacaoMemoria])-
>persista(arquivoIndividuos);
            deslocIndividuos += tamanho;
        }
    }
        memoria[localizacaoMemoria] = oid;
        ponteiros[localizacaoMemoria] = ponteiro;
        novo[localizacaoMemoria] = Ehnovo;
        tamanhoAntigo[localizacaoMemoria] = ponteiro->tamanho();
}

Individuo* SA::retornaNovaInstanciaIndividuo(long int oid)
{


    Individuo* individuo = new Individuo(oid);
        cout << "NOVA INSTANCIA" << endl;
    novaInstanciaIndividuo(oid,individuo,true);

    return individuo;
}


Individuo* SA::retornaInstanciaIndividuo(long int oid)
{
    Individuo* retorno;

    unsigned localizacaoMemoria = oid%tamMemoria;

    // Caso o dado esteja em memoria ...
    if( memoria[localizacaoMemoria] == oid )
    {
        return ponteiros[localizacaoMemoria];

    }else{
        // Caso nao esteja em memoria ...
        unsigned localizacaoDicionario = oid%dicionario->linhas();
```

```
        long int deslocamento = -1;

        // caso nao esteja em memoria é necessario primeiramente
recupera-lo do disco

        ListaDado<EntradaDicionario>* listaDicionario = dicionario-
>retorna(localizacaoDicionario,0,0);
        for( int i=0;i<listaDicionario->retornaNElem();i++)
        {
            if(((EntradaDicionario*)listaDicionario->retorna(i))->oid
== oid)
            {
                deslocamento = ((EntradaDicionario*)listaDicionario-
>retorna(i))->deslocamento;
                i = listaDicionario->retornaNElem();
            }
        }
        if(deslocamento > -1){
            char* dadosIndividuo;
            unsigned tamanho = 0;

            deslocIndividuosIn = deslocamento - deslocIndividuosIn;
            arquivoIndividuosIn->seekg(deslocIndividuosIn,ios::cur);
            arquivoIndividuosIn->read((char*)&tamanho,TAM_INT);
            arquivoIndividuosIn->seekg(-TAM_INT,ios::cur);
            deslocIndividuosIn += tamanho;

            dadosIndividuo = new char[tamanho];
            arquivoIndividuosIn->read(dadosIndividuo,tamanho);


            retorno = new Individuo(dadosIndividuo);
            novaInstanciaIndividuo(oid,retorno,false);

        }
    }
}
```

INDIVIDUO H

```
#ifndef IndividuoH
#define IndividuoH

#include "Lista.h"
#include "ModeloFace.h"
#include "Dado.h"
#include <iostream>
#include <fstream>

class Individuo : public DadoComplexo
{
    public:

        Individuo(long int oid);
        Individuo(char* registro);

        unsigned tamanho();
        void persista(ofstream* stream);


        ListaDado<ModeloFace>* modelos;
};







#endif
```

INDIVIDUO CPP

```
#include "Individuo.h"

Individuo::Individuo(long int oid):DadoComplexo(oid,(char)40)
{
    modelos = new ListaDado<ModeloFace>();
}

Individuo::Individuo(char* registro):DadoComplexo(registro)
{
    modelos = new ListaDado<ModeloFace>(&registro[DADOS]);
}

unsigned Individuo::tamanho()
{
    unsigned retorno = DadoComplexo::tamanho();
    retorno += modelos->tamanho();
}


void Individuo::persista(ofstream* stream)
{
    DadoComplexo::persista(stream);
    modelos->persista(stream);
}
```

SGD H

```
#ifndef SGDH
#define SGDH

#include "Face.h"
#include "ModeloFace.h"
#include "PreProcessamento.h"
#include "TabHashGeom.h"
#include "EntradaBucket.h"
#include "Lista.h"
#include "SA.h"

class SGD{
public:

        SGD();

        Individuo* retornaIndividuo(long int oid);
            void adiciona(MatrizSimples<double>* face, long int
oidIndividuo);
        void adicionaNovoIndividuo(MatrizSimples<double>* face , long
int oidIndividuo);
        ListaDado<EntradaBucket>*
retornaPossiveisIdentidades(MatrizSimples<double>* face);
            void alteraLimite(double novoLimite);
            double retornaLimite();

private:
        TabHashGeom* tabHash;
        PreProcessamento preProcessamento;
        SA* sa;

};
//-----------------------------------------------------------------------
-------
#endif
```

SGD CPP

```
#include "SGD.H"

SGD::SGD()
{
    tabHash = new TabHashGeom(100);
    sa = new SA();
}

void SGD::adicionaNovoIndividuo(MatrizSimples<double>* face , long int
oidIndividuo)
{

    ModeloFace* modelo = preProcessamento.criaModelo(face);
    tabHash->adiciona(modelo,oidIndividuo);

    cout << "SGD" << endl;

    Individuo* individuo = sa-
>retornaNovaInstanciaIndividuo(oidIndividuo);
    individuo->modelos->adiciona(modelo);

}

void SGD::adiciona(MatrizSimples<double>* face , long int
oidIndividuo)
{
      ModeloFace* modelo = preProcessamento.criaModelo(face);
    tabHash->adiciona(modelo,oidIndividuo);

    Individuo* individuo = sa-
>retornaInstanciaIndividuo(oidIndividuo);
    individuo->modelos->adiciona(modelo);
}

Individuo* SGD::retornaIndividuo(long int oidIndividuo)
{
    return sa->retornaInstanciaIndividuo(oidIndividuo);
}


ListaDado<EntradaBucket>*
SGD::retornaPossiveisIdentidades(MatrizSimples<double>* face)
{
    ModeloFace* modelo = preProcessamento.criaModelo(face);

    return tabHash->retornaCandidatos(modelo);

}

void SGD::alteraLimite(double novoLimite)
{
      preProcessamento.alteraLimite(novoLimite);
}

double SGD::retornaLimite()
{
       return preProcessamento.retornaLimite();
```

```
}
```