Marcelo Vaz Netto

BSCL: Algoritmo de Busca Sequencial de Colisões Lineares

Sorocaba, SP

08 de fevereiro de 2018

Marcelo Vaz Netto

BSCL: Algoritmo de Busca Sequencial de Colisões Lineares

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC-So) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Linha de pesquisa: Computação Científica e Inteligência Computacional.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCC-So

Orientador: Profa. Dra. Sahudy Montenegro González

Sorocaba, SP

08 de fevereiro de 2018

Netto, Marcelo Vaz
BSCL: Algoritmo de Busca Sequencial de Colisões Lineares/ Marcelo Vaz Netto.
– 2018.
64 f. : 30 cm.

Dissertação (Mestrado) – Universidade Federal de São Carlos – UFSCar
Centro de Ciências em Gestão e Tecnologia – CCGT
Programa de Pós-Graduação em Ciência da Computação – PPGCC-So.
Orientador: Profa. Dra. Sahudy Montenegro González
Banca examinadora: Profa. Dra. Annabell Del Real Tamariz, Profa. Dra. Tiemi Christine Sakata
Bibliografia

1. Busca binária. 2. Busca sequencial. 3. Tabelas *hash*. 4. Algoritmo de busca.
I. Orientador. II. Universidade Federal de São Carlos. III. Título

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Marcelo Vaz Netto, realizada em 08/02/2018:

Profa. Dra. Sahudy Montenegro Gonzalez
UFSCar

Profa. Dra. Annabell Del Real Tamariz
UENF

Profa. Dra. Tiemi Christine Sakata
UFSCar

Ao meu Pai, o pai que - se preciso fosse - teria pedido a Deus... (in memorian).

Agradecimentos

Agradeço,

a Deus, sem o qual nada existiria.

Aos meus pais, Nilton e Ruth, por serem a base e exemplo contínuo de perseverança em passar valores éticos e morais aos filhos.

À minha mulher, Glicéria, pelos momentos de compreensão e incentivo ao longo do tempo.

Aos meus filhos, Gabriela e Diogo, fontes de inspiração e razão da minha vida.

Aos meus irmãos, Bruno e Júnior, pelo fato de tê-los como irmãos de ética, moral e união.

À minha orientadora, Profa. Dra. Sahudy, pelos ensinamentos e dedicação constantes, sem os quais não seria possível esta dissertação.

“Make everything as simple as possible,
but not simpler”
(Einstein, Albert)

Resumo

Os acessos aos índices dos bancos de dados, às tabelas de símbolos dos compiladores, aos comandos dos sistemas operacionais, às portas dos dispositivos de roteamento, aos registros de telecomunicações, aos *caches* dos processadores, entre outros, estão inseridos em áreas de pesquisas da Ciência da Computação com um ponto em comum - algoritmo de busca. Estas aplicações podem manipular grandes conjuntos de elementos onde o custo de procurar uma chave pode ser elevado. Assim, há a necessidade de desenvolvimento de algoritmos que além da eficácia foquem - também - em eficiência, tanto no tempo de processamento como no uso dos recursos do sistema. Esta dissertação tem como objetivo propor um algoritmo de busca sequencial intitulado Busca Sequencial de Colisões Lineares (BSCL), baseado na distribuição probabilística de *Poisson*, para grandes matrizes de dados estáticos onde as colunas de chaves são números ordenados e uniformemente distribuídos. Como resultado, é apresentada uma comparação da BSCL com um algoritmo já consagrado e em uso em várias rotinas computacionais - Tabela *Hash* Perfeito. A validação experimental foca em três métricas - tempo de processamento, número de iterações e recursos de memória. A dissertação conclui que a BSCL é superior ao *Hash* Perfeito em tempo de computação e recursos de memória. Sua principal contribuição é demonstrar que rotinas mais simples podem comportar resultados computacionais mais expressivos.

Palavras-chaves: Colisões lineares. Distribuição probabilística de *Poisson*. Busca binária. Busca sequencial. Tabelas hash. Algoritmo de busca.

Abstract

Access to database indexes, compiler symbol tables, operating system commands, routing device ports, telecommunication registers, processor caches, among others, are inserted into areas of Computer Science research with a common point - search algorithm. These applications can handle large sets of elements where the cost of searching for a key can be high. Thus, there is a need to develop algorithms that, in addition to efficiency, focus on efficiency, both in processing time and in the use of system resources. This dissertation aims to propose a sequential search algorithm called Linear Collision Sequential Search (BSCL), based on the probabilistic distribution of Poisson, for large static data arrays where key columns are ordered and evenly distributed numbers. As a result, we present a comparison of the BSCL with an algorithm already consecrated and in use in several computational routines - Perfect Hash Table. The Experimental validation focuses on three metrics - processing time, number of iterations, and memory resources. The dissertation concludes that BSCL is superior to the Perfect Hash in computation time and memory resources. Its main contribution is to demonstrate that simpler routines can have more expressive computational results.

Keywords: Linear collisions. Poisson distribution. Binary search. Sequential search. Hash table. Search algorithm.

Lista de ilustrações

Figura 1 – *Hash* Perfeito. 24
Figura 2 – Princípio de Funcionamento da BSCL. 27
Figura 3 – BSCL - Representação do Pré-processamento. 29
Figura 4 – BSCL - Representação do Processamento. 32
Figura 5 – Gráfico de *Poisson*. 35
Figura 6 – Eficiência no Uso de Memória. 45
Figura 7 – Tempo de Processamento. 49

Lista de tabelas

Tabela 1 – Características mais Vantajosas - Baseado na Literatura 25
Tabela 2 – Probab. *Poisson*. 35
Tabela 3 – Distribuição de Colisões, $x \in \{0, 1\}$ 39
Tabela 4 – Distribuição de Colisões, $x \in \{2, 3\}$ 40
Tabela 5 – Recursos de Memória - em kB. 44
Tabela 6 – Eficiência de Memória 45
Tabela 7 – Número de Iterações - em Acessos à Base de Dados. 47
Tabela 8 – Tempos de Processamento - em μs. 48
Tabela 9 – Características dos algoritmos 51
Tabela 10 – Contagem BSCL por faixa (SMP7) - *Disconnecting Party* 56

Lista de abreviaturas e siglas

Anatel	Agência Nacional de Telecomunicações
ASN1	Abstract Syntax Notation One
BSCL	Busca Sequencial de Colisões Lineares
CPU	*Central Processing Unit*
CCC	Central de Comutação e Controle
CDR	*Call Detail Record*
DPI	Distância da Posição Inicial
EAQ	Entidade Aferidora da Qualidade
Fortran	*Formula Translation*
GCC	*GNU Compiler Collection*
Me	Mediana
NULL	Elemento Nulo
$\overline{M}$	Média aritmética simples
$\mathcal{O}()$	*Big O Function*
PMM	Período de Maior Movimento
QoS	Qualidade de Serviço
RAM	*Random Access Memory*
RGQ-SCM	Regulamento da Gestão de Qualidade do Serviço de Comunicação Multimídia
RGQ-SMP	Regulamento da Gestão de Qualidade do Serviço Móvel Pessoal
SGBD	Sistema de Gerenciamento de Banco de Dados

Lista de símbolos

a	Probabilidade de um evento ocorrer uma vez, *Poisson*
$A(x)$	Número de iterações
β_j	Faixa de índices
cB	Contagem de elementos de faixas com $p(x)$, referente ao BSCL
cH	Contagem de elementos de faixas com $p(x)$, referente ao *hash* perfeito
$\delta_{k'}$	- deslocamento do índice $k^{'}$ em relação a k, referente ao BSC
Δ	Matriz controle de colisão, referente ao BSC
e	Constante de *Euler*
$f^{'}_{min}(i)$	Menor valor de $f^{'}(i)$
$f^{'}_{max}(i)$	Maior valor de $f^{'}(i)$
Φ	Conjunto de vetores secundários, referente ao *hash* perfeito
$f(v_i)$	Equação linear de predição, referente ao BSC
$f^{'}(i)$	Função de deslocamento do índice, referente ao BSC
$f^{(n)}$	Derivada de grau n
$Hp(v_i)$	Função *Hash* Primária, referente ao *hash* perfeito
$Hs(v_i)$	Função *Hash* Secundária, referente ao *hash* perfeito
iB	Número de iterações, referente ao BSC
iH	Número de iterações, referente ao *hash* perfeito
I	Abscissa do vetor V, referente ao BSC
i, j	Índices
k'	índice de v_k predito pela $f(v_i)$, referente ao BSC
k	Índice em $V(i)$ que contém a chave v_k, referente ao BSC
Kg	Vetor de chaves

λ	Taxa de Ocorrência por unidade medida, *Poisson*
$left$	Início da faixa de colisão
lim	limite
MOD	Função módulo da divisão
n	Número de elementos em um conjunto de dados
$\mathbb{N}$	Conjunto dos números naturais
Ω	Conjunto de vetores de dados
$p(x)$	Função de probabilidade de colisão - distribuição de *Poisson*
P	Matriz primária, referente ao *hash* perfeito
q	Número de elementos em Kg
$right$	Fim da faixa de colisão
r	Número de tentativas, *Poisson*
$\sum p(x)\%$	Somatória de probabilidades, *Poisson*
St	Vetor secundário - algoritmo *hash* perfeito
tB	Tempo de processamento, referente ao BSC
tH	Tempo de processamento, referente ao *hash* perfeito
v_k	Chave de busca
V	Vetor de números inteiros, aleatórios e ordenados
Vd	Vetor de dados
x	Número de ocorrências, *Poisson*
x_{max}	Número máximo de ocorrências, *Poisson*
$\mathbb{Z}$	Conjunto dos Números Inteiros

Sumário

Introdução . . . 15

1 REVISÃO DA LITERATURA . . . 18
1.1 Busca Binária . . . 18
1.2 Tabelas *Hash* . . . 21
1.3 Resumo . . . 25

2 ALGORITMO BSCL . . . 26
2.1 Escopo do Algoritmo . . . 26
2.2 Princípio de Funcionamento . . . 27
2.2.1 Pré-processamento . . . 28
2.2.2 Processamento . . . 32
2.3 Fundamentação Matemática . . . 34

3 VALIDAÇÃO EXPERIMENTAL . . . 37
3.1 Protocolo dos Experimentos . . . 37
3.1.1 Recursos de Sistema . . . 37
3.1.2 Bases de Dados . . . 38
3.1.3 Métricas . . . 38
3.2 Validação dos Dados do Experimento com *Poisson* . . . 39
3.3 Resultados e Discussões . . . 41
3.3.1 Recursos de Memória . . . 41
3.3.2 Números de Iterações . . . 46
3.3.3 Tempo de Processamento . . . 48
3.4 Considerações Complementares . . . 49

4 APLICAÇÃO DA BSCL . . . 53

Conclusão . . . 58

Referências . . . 60

APÊNDICE A – FÓRMULAS . . . 62
A.1 Fórmula F.1 . . . 62
A.2 Fórmula F.2 . . . 62
A.3 Fórmula F.3 . . . 63

Introdução

Os acessos aos índices dos bancos de dados, às tabelas de símbolos dos compiladores, aos comandos dos sistemas operacionais, às portas dos dispositivos de roteamento, aos registros de telecomunicações, aos *caches* dos processadores, entre outros, estão inseridos em áreas de pesquisas da Ciência da Computação com um ponto em comum - algoritmo de busca.

As aplicações, anteriormente mencionadas, podem manipular grandes conjuntos de elementos. Nisto, o custo de procurar uma chave pode ser elevado. Assim, há a necessidade de algoritmos que além da eficácia foquem - também - em eficiência, tanto no tempo de processamento como no uso dos recursos do sistema (HOROWITZ; SAHNI, 1978).

Na literatura, há vários trabalhos voltados à eficiência em algoritmos de busca desenvolvidos ao longo dos anos. Como referência inicial, existe o trabalho de Fredman, Komlós e Szemerédi (1984) que apresentou uma estrutura - segundo os autores, fácil de ser implementada - em que uma tabela esparsa de *hash* para o acesso aos índices dos bancos de dados, de ordem $\mathcal{O}(1)$, pode ser construída usando uma técnica chamada de arquivamento linear. Conforme os autores, trata-se de uma indexação ao acesso a tabelas secundárias - com menor consumo de memória - usando funções de *hash*. Na conclusão, argumentam que os resultados de busca são superiores às técnicas tradicionais.

Em prosseguimento, existem trabalhos que focam na otimização do tempo de acesso aos dados - pelos algoritmos de busca - através de mudanças nas arquiteturas de *hardware*. Neste contexto, destaca-se o trabalho de Baer e Chen (1991) que propuseram um processador com uma rotina para prever o fluxo de instruções através de contadores. Quando os contadores encontram instruções de carga/armazenamento, preveem que os endereços dos blocos sejam pré-carregados de imediato em *cache* - no tempo ocioso do processador - ganhando tempo de processamento quando dos carregamentos e buscas.

Envolvendo, também, *cache*, Wilton e Jouppi (1996) propuseram abordagens diferentes nas rotinas de acesso aos dados. Os autores acrescentaram novos parâmetros nas equações já tratadas por Wada, Rajan e Przybylski (1992). Concluem que este novo algoritmo de busca é mais eficiente, uma vez que são introduzidas variáveis importantes como o *cache tag* e o comparador *set-associative memories* nas rotinas - as quais não foram levadas em conta no modelo apresentado inicialmente.

Geem, Kim e Loganathan (2001) estudaram novas possibilidades para os algoritmos de busca. Os autores classificaram-nas como algoritmos heurísticos, baseadas em analogias com fenômenos naturais ou artificiais. Nas suas propostas, as novas rotinas imitariam - como exemplo - a improvisação de instrumentistas na harmonia, quando completando

uma melodia. Os autores chamaram estes novos algoritmos de *Harmony Search*.

No tocante às telecomunicações, em estudos mais recentes, Wang (2017) direcionou o seu trabalho a uma arquitetura de roteamento inteligente. O modelo apresentado inclui algoritmos de busca voltados a uma autopercepção dos usuários - Qualidade de Serviço (QoS). Os algoritmos incluem rotinas que analisam variáveis como: energia, balanceamento de carga, otimização, decisão, aprendizado de máquina e principalmente o QoS. O autor validou a sua abordagem teórica verificando, experimentalmente, que em simulações computacionais os resultados vêm ao encontro da conclusão no seu trabalho.

Dentre os trabalhos estudados ao longo do desenvolvimento da dissertação, observou-se que nos processos computacionais existe uma curva de custo *versus* benefício quando se desenvolve uma nova rotina, ou mesmo uma mudança de *hardware* visando um determinado fim de otimização. Muitas vezes, ganha-se em eficiência no tempo ao custo de muito recurso. Neste contexto, deve-se analisar se o resultado obtido - pelo custo - é vantajoso. Sobre este prisma, a rotina que possui maior chance de estar em um ponto ótimo desta curva é a mais simples, pois sua execução é - geralmente - a que menos consome recursos do sistema (KNUTH, 1998).

Existe um algoritmo de busca que se destaca pelo uso em muitas áreas da Ciência da Computação, o algoritmo baseado em tabelas *hash*. E neste algoritmo, existe uma particularidade que é a tabela de *hash* perfeito. O algoritmo *hash* perfeito é muito eficiente nas buscas em número de iterações[1] - $\mathcal{O}(1)$ em qualquer caso (CORMEN et al., 2009); superior à busca binária que e de ordem $\mathcal{O}(\log n)$ no pior caso. Mas, existe um custo para esta eficiência - os recursos de memória. Em se tratando do uso de memórias, o *hash* perfeito está na ordem $\mathcal{O}(n)$ (CORMEN et al., 2009); inferior à busca binária que é de ordem $\mathcal{O}(1)$.

Baseados em características antagônicas, como as elencadas no parágrafo anterior, os pesquisadores almejam o desenvolvimentos de rotinas que primam pelas melhores características dos algoritmos; uma otimização baseada no custo *versus* benefício visando a eficiência. Esta é a linha de pesquisa neste trabalho.

Esta dissertação tem como objetivo propor um algoritmo de busca, baseado na distribuição probabilística de *Poisson*, para grandes matrizes de dados estáticos onde as colunas de chaves, que referenciam os dados, são números ordenados e uniformemente distribuídos. O algoritmo proposto, doravante intitulado Busca Sequencial de Colisões Lineares (BSCL), prima pela simplicidade, visando a alta eficiência de processamento com baixos recursos do sistema.

A premissa que ensejou as hipóteses para este trabalho de dissertação baseou-se numa análise da equação probabilística de *Poisson*. De (ROSS, 2009) obtém-se a equação

[1] Doravante, será adotado o termo iteração como sendo o número de vezes que os algoritmos terão que acessar à base de dados para ratificar a existência, ou não, da chave que procura (repetição).

de *Poisson*, com as devidas explicações.

$$p(x;\lambda) = \frac{e^{-\lambda}\ \lambda^{x}}{x!} \tag{1}$$

O parâmetro λ, $\lambda\ > 0$, é a taxa de ocorrência por unidade medida, como definição ($\lambda = ra$). Onde: $\underline{a}$ é a probabilidade do evento ocorrer uma vez, $\underline{p(x)}$ a probabilidade do evento ocorrer $\underline{x}$ vezes em $\underline{r}$ tentativas e $\underline{e}$ é a constante de *Euler*. Se ($\lambda = 1$) na equação, nota-se que a probabilidade de um evento ocorrer $\underline{x}$ vezes é inversamente proporcional a $\underline{x!}$, um amortecimento elevado para valores altos de $\underline{x}$.

Baseada nesta premissa, foi criada a hipótese de pesquisa: se o número de colisões geradas por uma equação de predição de 1º grau - regida pela distribuição probabilística de *Poisson* - for muito baixo para altos valores de x, pode uma busca sequencial - baseada na faixa onde houve a colisão - ser mais eficiente no processamento e nos recursos de memória do que uma tabela *hash* perfeito?

Como resultado de pesquisa, será demonstrado que um algoritmo usando uma busca sequencial, nas condições acima elencadas, é mais eficiente do que o *hash* perfeito em recursos de memória e tempo de processamento. Como contribuição, ratifica-se que rotinas mais simples - porém não simplórias - podem comportar resultados computacionais mais expressivos do que as mais complexas.

A dissertação está estruturada da seguinte forma:

- **Cap. 1 - Revisão da Literatura** - aborda-se o funcionamento de algumas rotinas de busca que são variantes das tabelas *hash* e busca binária. Esta revisão possibilitou delimitar métricas para a Validação Experimental.
- **Cap. 2 - Algoritmo BSCL** - onde são apresentados os fundamentos operacionais do algoritmo, exemplos e seus pseudocódigos. Bem como, a fundamentação matemática que levou à escolha da busca sequencial como motor de busca para o algoritmo.
- **Cap. 3 - Validação Experimental** - abordam-se o ambiente de desenvolvimento das rotinas computacionais, as bases de dados que foram usadas, a metodologia, resultados, as provas matemáticas obtidas e validadas pelos experimentos.
- **Cap. 4 - Aplicação da BSCL** - trata de uma aplicação prática para a BSCL no contexto da análise de qualidade dos serviços de telecomunicações.
- **Conclusão** - conclui-se que os objetivos propostos foram atingidos. Também, é feita uma abordagem que pode ensejar trabalhos futuros relacionados ao tema.
- **Apêndice** - onde são apresentadas deduções de fórmulas, específicas, que subsidiaram as comprovações matemáticas quanto ao uso dos recursos de memória e o número de iterações pelo algoritmo de referência (*hash* perfeito) e o proposto (BSCL).

1 Revisão da Literatura

Neste capítulo, na seção 1.1 e seção 1.2, respectivamente, é feita uma revisão da literatura quanto as rotinas de busca binária e tabelas *hash*, bem como suas nuances. Ressalta-se que a revisão da literatura não embarca todas as propostas existentes nas pesquisas sobre o tema.

Dado um conjunto de n elementos, sendo que cada um é identificado por uma chave, o objetivo de uma rotina de busca é localizar, no conjunto, o elemento que corresponde a uma determinada chave.

Rotinas de busca são tarefas comuns e bastante estudadas pela Ciência da Computação. Vários métodos e estruturas podem ser empregados para a otimização da busca, sendo que cada um tem suas vantagens e desvantagens, como se verá.

1.1 Busca Binária

Sob uma retrospectiva histórica, uma das primeiras publicações sobre rotinas de busca binária - com o objetivo de abordar a possibilidade de utilização prática em computadores da época - foi feita por Derrick Henry Lehmer no *"Teaching combinatorial tricks to a computer"* (LEHMER, 1960).

Em vetores de dados ordenados, a busca binária é um meio eficaz de se chegar à informação pretendida. Esta rotina computacional, na sua complexidade, é da ordem $\mathcal{O}(\log n)$, no pior caso (KNUTH, 1998). Consiste em dividir a base ordenada ao meio, procurar o elemento no ponto de divisão e caso não tenha conseguido encontrá-lo é descartada a metade dos dados que se tem a certeza de não conter o elemento que se procura, e assim sucessivamente no restante da base até encontrar - ou não - o elemento chave. O fim da rotina é o término dos elementos da base, ou o encontro da chave.

A busca binária, pura, não leva em consideração a distribuição dos elementos dentro da lista. Isto implica dizer que a rotina não aproveita das características das chaves existentes nos vetores. Consequentemente, abre mão advinda desta característica para efeito de eficiência. As rotinas de busca por funções de regressão levam em consideração a característica dos dados de forma a usar a predição como meio de obter melhor aproximação inicial de posicionamento das chaves (TORGO, 1997).

Na busca binária - por lógica - é fácil encontrar chaves pertencentes a uma determinada faixa. Achar os elementos de uma faixa corresponde a achar o primeiro e o último elementos da faixa. Os que estão entre estes valores - na base - pertencem à faixa procurada.

Existe uma característica positiva na rotina quanto ao uso de máquina: recurso de memória na ordem $\mathcal{O}(1)$. Isto significa que o espaço tomado pelo algoritmo é o mesmo para qualquer número de elementos no vetor ordenado (FLORES; MADPIS, 1971).

Existem variações no algoritmo de busca binária. Entre elas, destacam-se:

- ***Binary search tree:*** é uma estrutura de dados, em formato de árvore, em que cada nó tem até 2 ramificações. O objetivo deste algoritmo, após a estruturação dos dados, é permitir uma busca binária ao longo dos nós. O algoritmo inicia pelo nó raiz, sendo que: se a árvore está vazia, a chave procurada não existe na base. Por outro lado, se o valor é igual ao que está na raiz, a busca foi bem sucedida. Se o valor é menor do que o da raiz, a busca prossegue pelo nó à esquerda; e no inverso, pelo o da direita. Esta rotina prossegue até o valor ser encontrado num nó ou ser encontrado um nó vazio ($NULL$) - a chave não está na árvore. Este algoritmo, quando a árvore estiver balanceada, possui uma complexidade $\mathcal{O}(\log n)$ no pior caso (SEDGEWICK; WAYNE, 2011).

- ***Interpolation search*** [1]***:*** é um algoritmo que ao invés de usar o ponto médio para procurar a chave, num vetor numérico ordenado, calcula a posição provável da chave levando em consideração uma equação linear - referenciada ao índice - que passa pelo primeiro e último elemento do vetor. A rotina consulta a equação e obtêm a predição do índice da chave. Ele só funciona com alto desempenho se os elementos do vetor tiverem uma distribuição uniforme, caso contrário tem-se um desempenho inferior à busca binária pura. Este algoritmo pode chegar - no pior caso - a $\mathcal{O}(\log \log n)$ se os dados crescem linearmente, ou a $\mathcal{O}(n)$ se exponencialmente. Na prática, a busca binária por interpolação é mais lenta do que a busca binária pura para pequenos vetores, uma vez que a interpolação requer computação extra (PERL; ITAI; AVNI, 1978).

- ***Exponential search:*** é um algoritmo constituído de 2 estágios. O primeiro tem a finalidade de encontrar uma faixa de índices $\beta_j,\ j \geq 1$ $--$ no vetor ordenado $--$ em que a chave possa estar. O segundo consiste em fazer uma busca binária pura, dentro desta faixa, à procura da chave. O termo exponencial se deve ao fato de, no primeiro estágio, o crescimento da faixa β_j ser exponencial de base 2. A rotina inicia o processo analisando se o valor da chave está contida na faixa $\beta_1 = (2^{j-1}, 2^j)$, $j = 1$. Caso negativo, averígua se a chave está contida na faixa $\beta_2 = (2^{j-1} + 1, 2^j)$, $j = 2$. Caso ainda não seja encontrada incrementa-se na próxima faixa $\beta_3 = (2^{j-1} + 1, 2^j)$, $j = 3$, e assim sucessivamente. Não encontrada a faixa, o algoritmo retorna a negativa de procura. Caso encontrada a faixa β_j em que pode ser encontrada a chave, passa-se

[1] Como se verá ao longo da dissertação, este algoritmo foi usado como referência na construção das rotinas - da BSCL - que calculam as colisões lineares na procura das chaves.

para o segundo estágio - a busca binária pura nesta faixa. Este algoritmo trabalha na ordem $\mathcal{O}(\log i)$, onde i é o índice onde a chave deve ser encontrada no vetor ordenado. Se $i = n$, o pior caso, o algoritmo é de ordem $\mathcal{O}(\log n)$. Este algoritmo é indicado para vetores com grande número de elementos (BENTLEY; YAO, 1976) e (MOFFAT; TURPIN, 2002).

- ***Fibonacci search:*** o procedimento deste algoritmo, em relação a busca binária pura, consiste em dividir a faixa baseando em intervalos proporcionais à sequencia dos números de *Fibonacci*. O grande diferencial está na otimização do tempo de processamento, uma vez que a rotina do algoritmo trabalha com somas e subtrações, ao invés de divisões - como na busca binária pura (HASSIN, 1981). Segundo o Autor, a *Fibonacci search* tem a característica de examinar os elemento em menor dispersão na base ordenada. Consequentemente, melhor aproveitamento quanto ao *cache* - nos processadores - na leitura dos dados em memória. Segundo (HASSIN, 1981), o algoritmo tem um grande aproveitamento na alocação dos dados em *cache*, mas continua com uma complexidade $\mathcal{O}(\log n)$ no pior caso.

- ***Fractional cascading:*** o algoritmo tem a função de acelerar uma busca binária, para a mesma chave, em vetores distintos - porém inter-relacionados. Seja a condição de q vetores de números inteiros, distintos e ordenados, de n elementos cada um. Se é necessário achar uma chave v_k em todos os vetores - numa busca binária tradicional - são necessárias $q \cdot log_2(n)$ iterações no pior caso. Existindo uma matriz de índices que relaciona cada elemento de um vetor com os mesmos elementos nos outros vetores, para a procura da chave k basta ser feita num dos vetores - ao custo de $\mathcal{O}(\log n)$ - e após, numa consulta à matriz de índices, achar a mesma chave nos outros vetores ao custo de $\mathcal{O}(1)$. Segundo (CHAZELLE; GUIBAS, 1986) e (CHAZELLE; LIU, 2001) este procedimento propicia diminuição no tempo de processamento. Porém, existe um custo de memória na ordem de $\mathcal{O}(n)$ para a sua operação. Mas, no pior caso, o algoritmo ainda continua sendo de ordem $\mathcal{O}(\log n)$.

Diante o abordado nesta seção, nota-se que a busca binária - incluindo algumas das suas variantes - tende a ser de ordem $\mathcal{O}(\log n)$ no pior caso. Disto, desenvolveram-se algoritmos de forma a diminuir esta ordem no intuito de obter melhor eficiência computacional, são as rotinas apresentadas na seção 1.2.

1.2 Tabelas *Hash*

Uma tabela *hash* é uma estrutura de dados eficiente para implementar dicionários. Em muitas aplicações, como exemplo um compilador para uma linguagem de computador, procura-se somente uma determinada chave para uma determinada ação como *insert* ou *delete*. Nestes casos, as tabelas *hash* funcionam muito bem (CORMEN et al., 2009).

Knuth (1998) explica sucintamente o que é *hashing* quando o descreve como sendo um cálculo aritmético sobre um valor K, tendo como base uma função $f(K)$ que é a localização de K, e os seus dados associados, em uma tabela.

Tanto Cormen et al. (2009) quanto Knuth (1998) descrevem as tabelas *hash* como muito eficientes em se tratando de algoritmo de busca, estando a busca na ordem de $\mathcal{O}(1)$ - no caso médio - quando a função $f(K)$ é bem escolhida. Os autores colocam que um dos problemas das tabelas *hash*, quando não em endereçamento direto, é a colisão - quando valores diferentes de K geram o mesmo $f(K)$.

Segundo Cormen et al. (2009) colisões são indesejáveis, mas existem técnicas para resolver o conflito criado por colisões. Os mecanismos mais comuns para tratamento de colisões são descritos, pelo autores, como se segue:

- **Endereçamento aberto:** neste caso, a informação é armazenada na própria tabela. Devem ser feitas sondagens, através de funções *hash*, para que novas informações sejam alocadas em espaços vazios da tabela. Um pseudo-algoritmo que explica a alocação em endereçamento aberto, para uma determinada chave k, é o seguinte:

```
1- Calcular a posição p onde k deve ser armazenada: p = h(k)
2- Se a posição p não estiver ocupada -> armazene k nesta posição e Fim
3- Se a posição p estiver ocupada -> recalcule p e vá para o passo 2.
```

  A questão é como deve ser recalculada outra posição de p - no último passo. Basicamente, são 3 as técnicas para solucionar este novo valor:

  - **Sondagem linear:** neste caso, a função *hash* que calculará este novo posicionamento é dado por $p(k) = (i + h_1(k))\ mod\ m$, em que m é o tamanho da tabela, $i = 0, 1, ..., m - 1$ incrementado em cada nova sondagem, e h_1 uma função *hash*. Segundo o Autor, é fácil de ser implementada, porém existe um problema gerado conhecido como "agrupamento primário" - longas sequências ocupadas gerando maior tempo de pesquisa.
  - **Sondagem quadrática:** basicamente, usa-se nesta técnica uma equação de segundo grau conjuntamente com uma função *hash* auxiliar. O posicionamento

é calculado pela fórmula $p(k,i) = (h_1(k) + c_1 * i + c_2 * i^2)\ MOD\ m$, onde c_1 e c_2 são constantes auxiliares diferentes de zero. Nota-se que, através da sondagem i existe uma janela de deslocamento na função de procura. Aqui, existe um problema - semelhante ao anterior - chamado "agrupamento secundário", na qual duas chaves tendo a mesma posição de sondagem inicial geram sequências iguais, consequentemente, maior tempo de pesquisa.

  – ***Hash* duplo:** uma maneira de reduzir o agrupamento primário e secundário é incrementar p não por uma constante - como na sondagem linear -, mas por uma quantidade que depende da chave, uma nova função *hash*. Tem-se, assim, duas funções *hash*: $h_1(k)$ e $h_2(k)$ de forma que o novo posicionamento é dado por $p(k,i) = (h_1(k) + i * h_2(k))\ MOD\ m$. É uma das melhores técnicas, porque as permutações geradas são muito aleatórias.

- **Encadeamento:** neste mecanismo, a informação está armazenada em estruturas encadeadas numa tabela externa. Uma função *hash* direciona para o início do encadeamento na tabela. O algoritmo encontra um elemento armazenado seguindo o encadeamento até acha-lo ou encontrar um espaço vago (NIL), retornando a negativa da busca. Inserções na tabela requerem uma busca dentro da lista encadeada até achar um elemento vago, onde será inserida a nova informação. Uma remoção requer a atualização do encadeamento na tabela.

Como na busca binária, apresentada na seção anterior, no algoritmo *hash* existem variações. Estas variações têm suas utilizações em campos específicos da Ciência da Computação. Entre elas, destacam-se as seguintes:

- ***Robin Hood hashing:*** o algoritmo é uma variante do endereçamento aberto. Baseia-se em achar a posição inicial dada pela função *hash*, e a partir desta - havendo colisão - uma sondagem inicia uma rotina: para cada nova posição encontrada é feita uma comparação da "Distância da Posição Inicial (DPI)"[2] do elemento que já está alocado na tabela com a DPI do que se quer fazer a inserção. Se a DPI do elemento que se quer fazer a inserção é maior que a DPI do elemento que está inserido, é feita uma troca - insere-se o elemento a ser inserido nesta posição e o elemento que estava inserido passa agora a ser o que continua na rotina, a ser inserido. Continuando deste ponto, o algoritmo tenta achar um novo elemento alocado - de DPI menor - para substituí-lo. Este procedimento de procura por DPI menor - e fazendo as substituições - continua até ser encontrada uma posição vazia para se alocar o elemento a ser inserido. Na prática, trata-se de um ordenamento em função

[2] A partir que os elementos vão sendo inseridos na tabela, tendem a ficar mais longe da posição calculada inicialmente pela função *hash*. A DPI é obtida pela distância da posição atual em relação à inicial, baseada no número de iterações na sondagem.

da DPI, propiciando menos iterações para achar a chave de busca - uma vez que ela terá a menor DPI possível[3] (CELIS; LARSON; MUNRO, 1985).

- ***Cuckoo hashing:*** é outro algoritmo variante do endereçamento aberto. A ideia básica é usar duas funções *hash* para endereçar as chaves na tabela. Com isto, é possível ter duas posições de endereços válidos durante a busca e na inserção de cada chave. Durante a busca, o algoritmo pesquisa as duas posições na tabela para decidir da existência - ou não - da chave procurada. No caso de uma inserção, a alocação pode ser feita em qualquer um dos endereços das funções *hash*, basta estar vazia. Se as duas posições estiverem ocupadas, o algoritmo faz uma varredura realocando uma delas em outra posição válida, e alocando na que ficou vaga o elemento a ser inserido. O processo de inserção continua até o algoritmo achar uma posição, vaga e válida, em que possa ser realocada a chave. Neste processo, pode haver falha quando o algoritmo tende a entrar em ciclo infinito tentando achar uma posição na tabela. Se isto ocorrer, é reconstruída toda a tabela usando duas novas funções *hash* (PAGH; RODLER, 2001).

- ***2-choice hashing:*** este algoritmo usa, também, duas funções *hash* para endereçar chaves na tabela. O diferencial é que não insere uma chave analisando somente a condição vaga na tabela, e sim - adicionalmente - uma posição em que o comprimento do encadeamento - proveniente de uma função - é menor do que o da outra. Este procedimento contribui com o balanceamento da tabela, proporcionando melhor eficiência na busca - tende a buscar chaves em sequencias mais curtas (RICHA; MITZENMACHER; SITARAMAN, 2001).

- ***Cryptographic hash functions:*** trata-se de uma classe especial de funções *hash* com características de uso específico em criptografia. Neste algoritmo, a função hash tem a característica de mapear uma entrada de tamanho arbitrário para uma saída de tamanho fixo. Segundo (PRENEEL, 1994) e (WANG et al., 2004), uma função *hash* criptográfica - ideal - possui algumas propriedades especiais, destacando-se:

  - **irreversível:** inviável, através da saída, retornar ao valor de origem sem a chave criptográfica;
  - **determinística:** uma mesma entrada resulta sempre na mesma saída;
  - **única:** impossível encontrar duas entradas diferentes com a mesma saída;
  - **veloz:** grande velocidade de cálculo da saída para qualquer entrada;
  - **sensível:** pequenas mudanças na entrada geram grandes mudanças na saída.

[3] No caso da rotina de endereçamento aberto - tradicional -, um elemento a ser inserido pode ficar muito distante da sua posição inicial dada pela função *hash* - em número de iterações. Isto tende a diminuir a eficiência do algoritmo durante a busca.

- ***Perfect hash:*** o algoritmo, conforme representado na Figura 1, usa nas suas rotinas uma matriz primária de controle $P = (p_{ij})_{n\times 2}$ e um conjunto de vetores secundários - $\Phi = \{S0, S1, ..., S(n-1)\}$ - onde um determinado vetor $St = (s_{ij})_{1\times p_{t0}}$, $t \leq n-1$, é associado a um registro i de P. Cada célula do vetor St é acessada através dos elementos do vetor $V = (v_{ij})_{n\times 1}$ por duas funções *hash* em cascata: $Hp(v_i)$ (em primeiro nível) e $Hs(v_i)$ (em segundo nível). O conjunto de vetores Φ é descontínuo: para os elementos de v_i em que $Hp(v_i)$ não inseriu o número de colisões ao quadrado em p_{i0}, não são criados os vetores correspondentes St associados ao registro de P, representados por valores vagos - cinza - em p_{i0} e p_{i1} de P; e a inexistência do vetor St no conjunto Φ.

Figura 1 – *Hash* Perfeito.

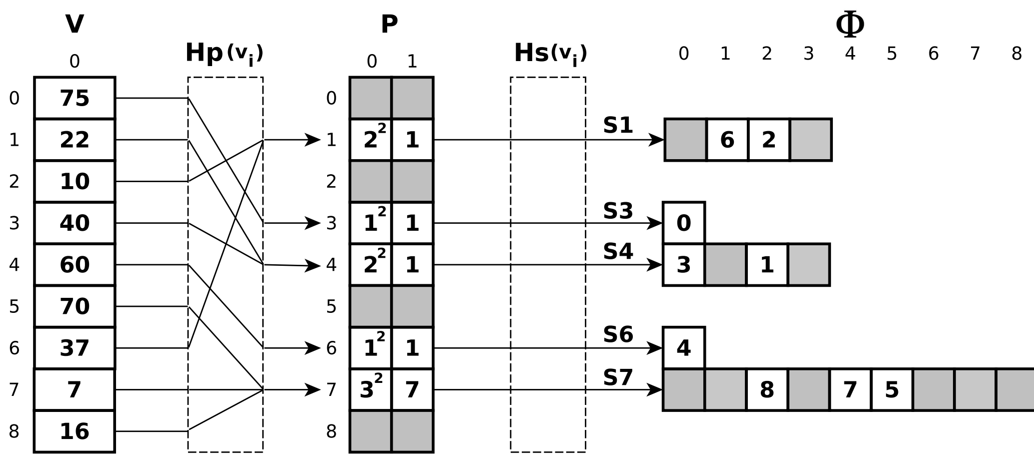


Figura adaptada de (CORMEN et al., 2009)

Baseado em Cormen et al. (2009) - matematicamente provado pelos autores - se um vetor St possuir p_{t0}^2 células, sendo p_{t0} a quantidade de elementos a serem armazenados no vetor - provenientes da colisão gerada pela função $Hp(v_i)$ -, e escolhermos $Hs(v_i)$, ao acaso, de uma classe universal de funções *hash*, a probabilidade de colisões no vetor St é menor do que 50%. Daí o nome de *hash* perfeito: distribuição dos valores de V em St sem colisões. Como consequência, o algoritmo é de ordem $\mathcal{O}(1)$ - na busca - para qualquer caso. Em contrapartida, há desperdício de memória em St - em cinza na Figura 1 - para $p_{t0} > 1$.

Seguindo os modelos de funções apresentadas por Cormen et al. (2009), as funções *hash* - $Hp(v_i)$ e $Hs(v_i)$ - têm como variável o valor de v_i. Onde, a função *hash* primária $Hp(v_i) = MOD(MOD(A_p \cdot v_i + B_p; Pr); n)$ e a secundária $Hs(v_i) = MOD(MOD(p_{i1} \cdot v_i + B_s; Pr); p_{i0})$ são caracterizadas por funções módulos da divisão, sendo: A_p, B_p, p_{i1} e B_s constantes e Pr um número primo maior que a quantidade de registros de P.

Como exemplo, na fase de pré-processamento, sejam: $A_p = 1$, $B_p = 0$, $Pr = 101$, $n = 9$. Então, a função $Hp(v_i)$ será $Hp(v_i) = MOD(MOD(v_i; 101); 9)$. Se aplicarmos os valores de $v_5 = 70$, $v_7 = 7$ e $v_8 = 16$ na função $Hp(v_i)$ terão como resultado o índice 7 da patriz P. Como houve 3 colisões, em $P(7, 0)$ é armazenado o número de colisões ao quadrado $P(7, 0) = 3^2$.

Em seguida, é feita a escolha de $P(7, 1)$ e B_s, na função Hs, de forma que não haja colisão no vetor $S7$. Sejam os valores $P(7, 1) = 7$, $B_s = 0$, $Pr = 101$ e $P(7, 0) = 3^2$. Então, a função $Hs(v_i)$ será $Hs(v_i) = MOD(MOD(7v_i; 101); 3^2)$. Aplicando os valores de $v_5 = 70$, $v_7 = 7$ e $v_8 = 16$, em $Hs(v_i)$, serão direcionados para os índices 5, 4 e 2 de $S7$, respectivamente. Nestes índices, são referenciados os índices do vetor V em que se encontram. Portanto, em $S7(5) = 5$, $S7(4) = 7$ e $S7(2) = 8$.

Agora, na fase de processamento, seja a procura da chave $v_k = 16$. Aplicando este valor de v_k na função $Hp(v_i)$ ela direcionará a busca para o índice 7 de P. Aplicando o valor de v_k na função $Hs(v_i)$ a busca será direcionada para o índice 2 de $S7$. Em $S7(2)$ está armazenado o valor 8 que corresponde ao índice de V onde está armazenado o valor 16. Portanto, $V(8) = 16$, encontrada a chave.

1.3 Resumo

Nesta seção, apresenta-se na Tabela 1 um resumo - baseado na literatura - das principais características dos dois grupos de algoritmos - Busca Binária e Tabelas *Hash* -, comparativamente, que subsidiou tomadas de decisões no projeto do algoritmo BSCL. A característica é marcada com ✓ no algoritmo em que se destaca, mais favoravelmente.

Tabela 1 – Características mais Vantajosas - Baseado na Literatura

Característica	Busca Binária	Tabelas *Hash*
Rotinas mais simples:	✓	
Menor Tempo de Busca:		✓
Manutenção mais simples:	✓	
Busca de Chaves por Faixas:	✓	
Menos Recursos de Memória:	✓	
Não Necessita de Ordenamento:		✓
Busca de Dados Não Numéricos:	✓	
Menor Número de Iterações (Média):		✓

Legenda: ✓- característica mais favorável.

Como se verá ao longo da dissertação, as três características principais no projeto da BSCL - de forma decrescente de relevância - foram: tempo de processamento, recursos de memória e número de iterações.

2 Algoritmo BSCL

Este capítulo aborda a BSCL, sendo estruturado da seguinte forma: na seção 2.1 é mostrado o escopo dos dados em que se enquadra o funcionamento do algoritmo. Na seção 2.2 é descrito o princípio de funcionamento do algoritmo, sendo que nas subseção 2.2.1 e subseção 2.2.2 é mostrado - em maiores detalhes - o tratamento das colisões com exemplos e os pseudocódigos das rotinas. Na seção 2.3 é abordada a fundamentação matemática que serviu de base para a escolha da busca sequencial como motor de busca para a BSCL.

2.1 Escopo do Algoritmo

A BSCL é um algoritmo de busca sequencial, baseada na distribuição probabilística de *Poisson*, para grandes matrizes de dados estáticos onde as colunas de chaves são números ordenados e uniformemente distribuídos. O núcleo do seu funcionamento baseia-se na busca sequencial de uma chave, numa faixa de índices, cujas chaves estão em colisão. Sendo a colisão regida por uma equação linear de predição e por uma matriz controle de colisão.

O Algoritmo tem como objetivo: tratar a busca de chaves visando a eficiência no uso de memória, atrelado ao menor tempo para concluir a busca. Estes resultados são obtidos usando funções lineares de baixa complexidade, e tempo de processamento, atrelados a uma matriz controle de colisão com colunas com poucos *bytes* alocados.

Segundo Hiller e Lieberman (1995), na natureza há fenômenos observáveis que se manifestam-se de forma temporal e/ou espacial, podendo ter características aleatórias. Para os autores, um conjunto destas variáveis aleatórias é descrito como pertencente aos processos estocásticos[1]. Os autores estabelecem estes fenômenos como sendo o oposto dos determinísticos, sendo estes caracterizados por funções que resultam em valores bem definidos ao longo do tempo e/ou espaço.

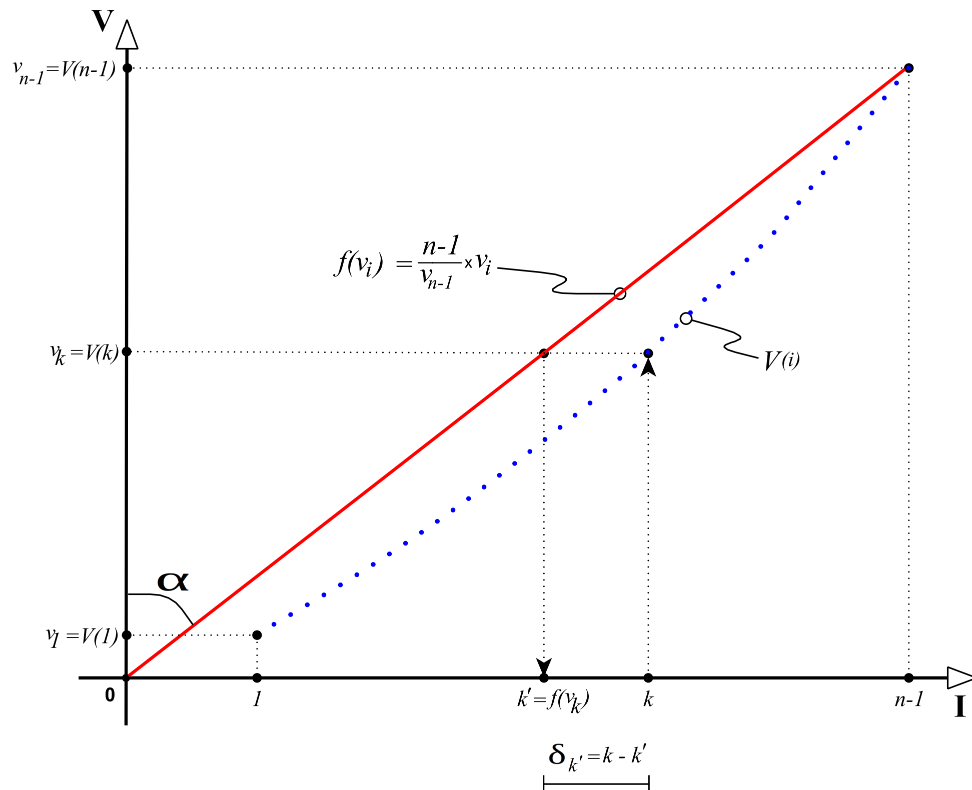
A BSCL foi projetada para o processamento, com maior desempenho, nos processos estocásticos. Sendo que, sob estes tipos de processos, a distribuição probabilística de *Poisson* - de forma a facilitar os cálculos de uma distribuição binomial de probabilidade - pode ser usada para tratar as colisões de chaves. Se a coluna de chaves é proveniente de processos estocásticos, a submissão de colisões - através de uma equação linear - gerará faixas de colisões que podem ser regidas pela distribuição probabilística de *Poisson*[2].

[1] Em inglês: *Stochastic Process* ou *Random Process* (HILLER; LIEBERMAN, 1995).
[2] Esta afirmação será comprovada na seção 3.2 através das tabelas de distribuição de colisões.

2.2 Princípio de Funcionamento

De forma a explicar o princípio geral de funcionamento da BSCL é apresentada na Figura 2 alguns elementos abordados na seção 2.1.

Figura 2 – Princípio de Funcionamento da BSCL.



Existem dez elementos do algoritmo BSCL representados no gráfico da Figura 2:

- $V(i)$ - vetor de chaves, representado discretamente pela linha pontilhada.
- Abcissa I - índices do vetor $V(i)$.
- Ordenada V - dados do vetor $V(i)$.
- n - número de elementos no vetor $V(i)$.
- $f(v_i)$ - equação linear de predição da BSCL.
- v_k - valor da chave a ser procurada em $V(i)$.
- $k^{'}$ - índice de v_k predito pela $f(v_i)$.
- k - índice em $V(i)$ que contem a chave v_k.

- $\delta_{k'}$ - deslocamento do índice k' em relação a k.
- Δ - matriz onde são armazenados os valores de $\delta_{k'}$.

Partindo de um índice k da abcissa I e consultando o vetor $V(i)$, onde $i = k$, é fácil achar o valor armazenado $v_k = V(k)$ representado na ordenada **V**. Agora, se o fluxo for o inverso - partindo da ordenada **V** tentar achar o índice da chave v_k na abcissa **I**, só é possível através de um algoritmo de busca em $V(i)$. Para resolver esta condição na BSCL, foi criada a função $f(v_i)$ que dá uma predição do índice de uma chave procurada em $V(i)$. Neste caso, o índice predito para a chave v_k, representado na Figura 2, é dado por:

$$k' = f(v_k) \tag{2.1}$$

Notar que o valor predito k' está deslocado do valor real k por $\delta_{k'}$. Para solucionar este deslocamento da predição, existe na fase de pré-processamento a criação da matriz Δ na qual é armazenada, na posição predita pela função $f(v_i)$, onde $v_i = v_k$, o valor do deslocamento do índice predito k' em relação a k:

$$\delta_{f(v_k)} = k - k' \tag{2.2}$$

Com estes parâmetros, é possível achar o índice de v_k no vetor $V(i)$ através da seguinte fórmula:

$$f(v_k) + \delta_{f(v_k)} \tag{2.3}$$

Substituindo a Equação 2.1 e Equação 2.2 na equação Equação 2.3 valida-se:

$$f(v_k) + \delta_{f(v_k)} = k' + \delta_{k'} = k' + (k - k') = k \tag{2.4}$$

Entretanto, surge um adicional de complexidade não apresentado na seção 2.2, as colisões geradas pela função $f(v_i)$ para chaves de valores muito próximos - mesmos índices preditos sendo gerados. Esta condição é analisada nas subseção 2.2.1 e subseção 2.2.2.

2.2.1 Pré-processamento

Conforme descrito na seção 2.2, a função do pré-processamento era criar a matriz Δ onde seriam armazenadas os valores de $\delta_{k'}$. Mas como surgiu o problema das colisões geradas pela função $f(v_i)$, a matriz Δ passa a controlar colisões e não deslocamentos. Agora, os elementos previamente apresentados na Figura 2 são representados - matematicamente - como se seguem:

- **Vetor de dados:** $V = (V_{ij})_{n\times 1}$ com elementos $v_i \in \mathbb{N}\ \forall i \in \mathbb{N},\ 0 \le i \le (n-1)$. Onde, $v_{i-1} < v_i\ \forall i \in \mathbb{N},\ 1 \le i \le (n-1)$.

- **Equação linear de predição:** representada pela equação de 1º grau.

$$f(v_i) = \frac{n-1}{v_{n-1}} \times v_i \ , \ f(v_i) \in \mathbb{N} \tag{2.5}$$

- **Matriz controle de colisão:** $\Delta = (\Delta_{ij})_{n\times 2}$ com elementos $\delta_{ij} \in \mathbb{Z}\ \forall ij \in \mathbb{N}$ onde $0 \leq i \leq (n-1)$ e $1 \leq j \leq 2$.

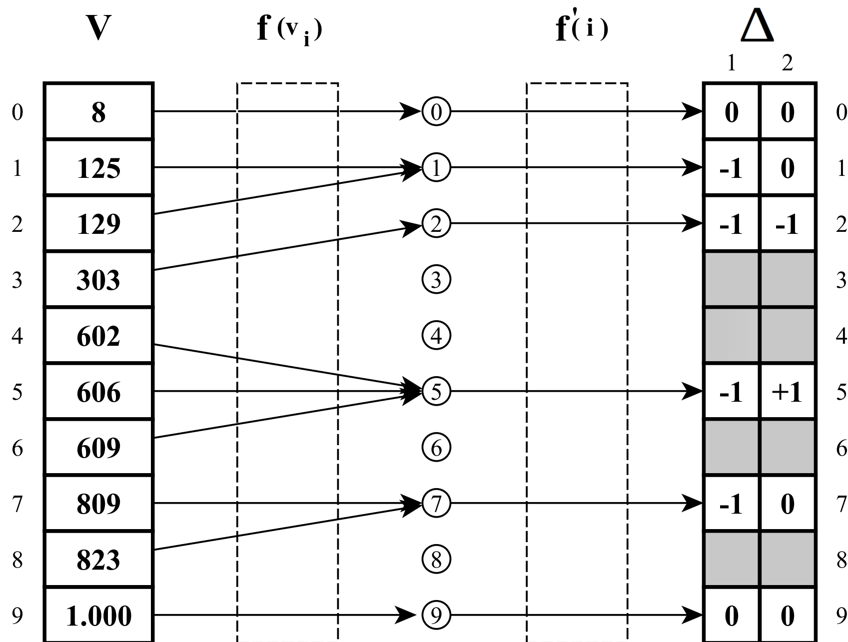
De forma a simplificar a leitura, nas explicações que se seguem, adota-se a representação dos elementos da matriz controle de colisão como:

$$\Delta(f(v_i), j) \tag{2.6}$$

Esta nomenclatura se deve ao fato do índice i da matriz Δ ser referenciado, sempre, pelo índice predito na equação linear de predição $f(v_i)$ (Equação 2.5).

O pré-processamento da BSCL pode ser representado pelo exemplo da Figura 3.

Figura 3 – BSCL - Representação do Pré-processamento.



A primeira rotina do algoritmo consiste em fazer uma varredura em todos os n registros de V e calcular os valores da equação linear[3] de predição $f(v_i)$ (Equação 2.5). Como resultado, os índices gerados pela equação podem estar em colisão - mesmos índices preditos sendo gerados por v_i diferentes.

[3] De onde vem o termo "Colisão Linear" da BSCL.

Havendo colisões, existe uma segunda etapa - representada por $f^{'}(i)$ na Figura 3 - que tem a função de calcular o deslocamento de $f(v_i)$ - índice de predição - em relação ao índice i do elemento v_i na faixa de colisão. Esta equação de deslocamento é dada por:

$$f^{'}(i) = f(v_i) - i \ , \ f^{'}(i) \in \mathbb{Z} \tag{2.7}$$

Em prosseguimento, uma rotina acha o menor e maior deslocamento - da faixa que houve colisões - e arquiva-os nas posições:

$$\Delta(f(v_i), 1) \Leftarrow f^{'}_{min}(i) \qquad \Delta(f(v_i), 2) \Leftarrow f^{'}_{max}(i) \tag{2.8}$$

Usando a Figura 3 como exemplo, e mostrado os passos das rotinas da BSCL nesta fase do pré-processamento, torna mais simples o entendimento:

1º Passo - cálculo da equação linear de predição.

A equação linear de predição é obtida substituindo os elementos de V, representados na Figura 3, diretamente na Equação 2.5. Disto, tem-se:

$$f(v_i) = \frac{n-1}{v_{n-1}} \times v_i = \frac{10-1}{1.000} \times v_i = 0{,}009 v_i \ , \ f(v_i) \in \mathbb{N} \tag{2.9}$$

2º Passo - varredura dos registros de V.

Durante esta fase, para os registros em que $v_4 = 602$, $v_5 = 606$ e $v_6 = 609$, sendo aplicada a Equação 2.9 e considerando só a parte inteira do resultado, encontra-se os índices de predição:

$$v_4 = 602 \longrightarrow f(602) = 0{,}009 \times 602 = 5{,}418 \longrightarrow f(602) = 5$$

$$v_5 = 606 \longrightarrow f(606) = 0{,}009 \times 606 = 5{,}454 \longrightarrow f(606) = 5$$

$$v_6 = 609 \longrightarrow f(609) = 0{,}009 \times 609 = 5{,}481 \longrightarrow f(609) = 5$$

Houve colisões nesta faixa analisada de V $(f(v_i) = 5 \ \forall i \in \mathbb{N}, \ 4 \leq i \leq 6)$. Portanto, devem ser tratadas e referenciadas na matriz de colisão Δ.

3º Passo - cálculo do deslocamento de colisões.

Substituindo a Equação 2.9, já com os devidos valores calculados no 2º Passo $(f(v_i) = 5 \ \forall i \in \mathbb{N}, \ 4 \leq i \leq 6)$, na Equação 2.7, obtém-se a equação de deslocamento:

$$f^{'}(i) = 5 - i \ \ \forall i \in \mathbb{N}, \ 4 \leq i \leq 6 \tag{2.10}$$

Da Equação 2.10 calculam-se os deslocamentos para todos os valores dos índices que colidiram ($4 \leq i \leq 6$):

$$i = 4 \longrightarrow f^{'}(4) = 5 - 4 = +1$$

$$i = 5 \longrightarrow f^{'}(5) = 5 - 5 = \ \ 0$$

$$i = 6 \longrightarrow f^{'}(6) = 5 - 6 = -1$$

4º Passo - armazenamento dos deslocamentos de índices em Δ.

Com os dados de deslocamentos calculados no 3º Passo, juntamente com o valor de $f(v_i) = 5$ calculado no 2º Passo, é feito o armazenamento do menor ($f^{'}_{min}(i) = f^{'}(6)$) e maior ($f^{'}_{max}(i) = f^{'}(4)$) deslocamentos nas posições de Δ referenciadas pela Equação 2.8:

$$\Delta(f(v_i), 1) = \Delta(5, 1) = f^{'}(6) = -1$$

$$\Delta(f(v_i), 2) = \Delta(5, 2) = f^{'}(4) = +1$$

Com isto, na linha 5 da matriz Δ, representada na Figura 3, constam os valores do menor e maior deslocamento do índice predito 5: -1 e +1. Este procedimento é feito em todos elementos de V. Havendo faixas em colisão, são tratadas e arquivadas em Δ.

A título de maiores detalhes, na Rotina 1 apresenta-se o pseudocódigo de pré-processamento da BSCL com os devidos comentários.

Rotina 1. Pseudocódigo BSCL: Pré-Processamento.

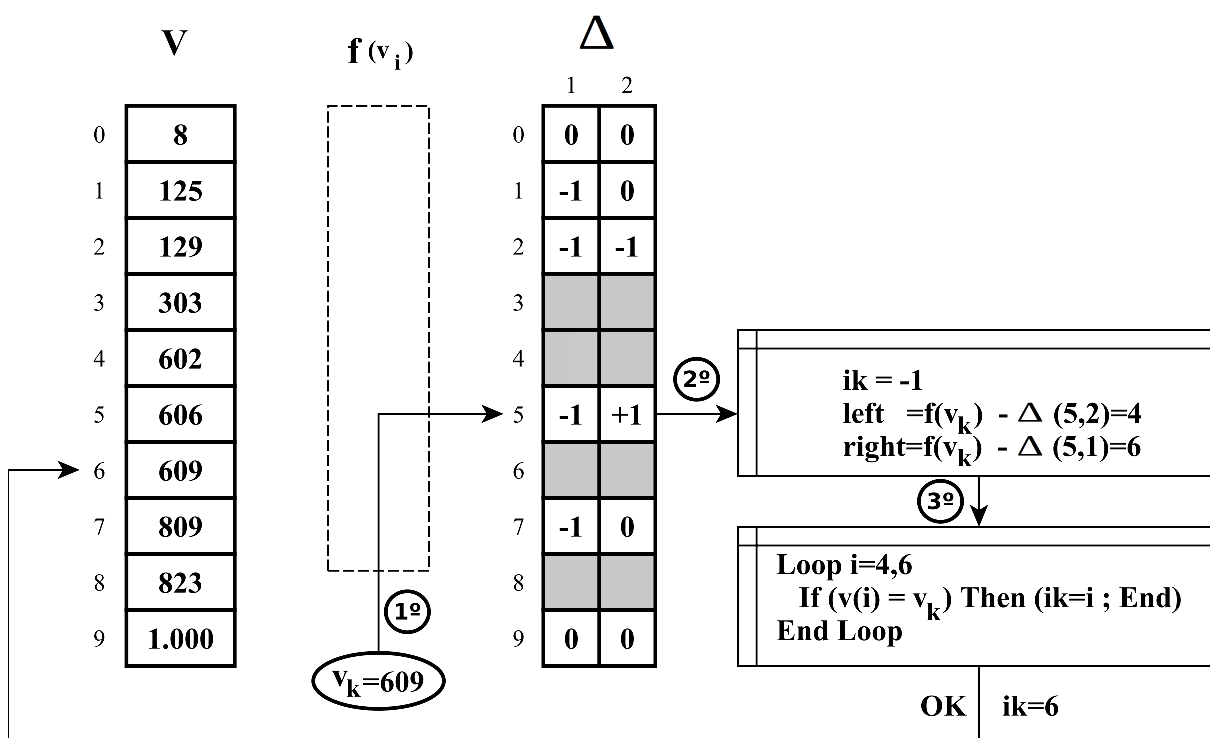
```
Aloc_Dim_Mem                                   #Aloca e Dimensiona memória.
Load_Data                                        #Carrega dados na memória.
lm=STORAGE_SIZE(D)             #Maior valor possível de armazenamento em Delta.
D(:,2)=-lm                      #Alocação inicial do deslocamemto da coluna d2.
D(:,1)=+lm                      #Alocação inicial do deslocamemto da coluna d1.
tg=(n-1)/v(n-1)                          #Inclinação da reta f(vi) <-- tan(alfa).
Loop i=0,n-1                              #Início do Loop para os n índices de V.
  p=tg*v(i)                           #Calcula o índice de v(i) na matriz Delta.
  d=p-i                         #Deslocamento de p em relação ao índice i de V.
  If (d > D(p,2)) Then D(p,2)=d          #Aloca valor a Delta_max caso delta >
  If (d < D(p,1)) Then D(p,1)=d          #Aloca valor a Delta_min caso delta <
End Loop i
```

2.2.2 Processamento

Esta fase do algoritmo consiste em procurar uma chave v_k no vetor V e apresentar - caso a procura tenha êxito - o índice ik em que se localiza no vetor V. Caso contrário, o índice será NIL representado por -1.

O processamento da BSCL pode ser representado pelo exemplo da Figura 4.

Figura 4 – BSCL - Representação do Processamento.



O processamento utiliza a mesma equação do pré-processamento - Equação 2.5 - direcionando a busca para um registo na matriz controle de colisão Δ, o qual possui os parâmetros para a rotina de processamento seguinte calcular a única faixa de índices em V onde é possível encontrar a chave v_k. Posteriormente, dentro da faixa de índices calculados, é feita a procura de v_k em V através de uma busca sequencial. Encontrando a chave v_k, é mostrado o índice em V onde foi encontrada.

Usando a Figura 4 como exemplo, e mostrado os passos nas rotinas da BSCL onde o algoritmo encontra a chave ($v_k = 609$) no vetor V, torna mais simples o entendimento:

1º Passo - cálculo do índice de predição da chave v_k.

A predição da chave ($v_k = 609$), em Δ, é obtida diretamente da Equação 2.5, juntamente com os dados de V disponíveis na Figura 4. É considerada como saída -

procedimento igual ao da subseção 2.2.1 - somente a parte inteira do resultado:

$$f(v_i) = f(v_k) = \frac{n-1}{v_{n-1}} \times v_k = \frac{10-1}{1.000} \times v_k = 0,009v_k \longrightarrow f(609) = 5,481 \longrightarrow f(609) = 5 \quad (2.11)$$

2º Passo - cálculo da faixa de colisão.

Baseado no índice de predição da chave, o 2º Passo tem como objetivo achar a faixa de índice em V onde é possível encontrar a chave v_k.

As equações que calculam o início e fim da faixa de colisão, são:

$$left = f(v_k) - \Delta(f(v_k), 2) \qquad right = f(v_k) - \Delta(f(v_k), 1) \quad (2.12)$$

Usando o valor da predição encontrada na Equação 2.11 e substituindo na Equação 2.12, juntamente com os dados armazenados em Δ, obtêm-se o parâmetro esquerdo da faixa:

$$left = f(v_k) - \Delta(f(v_k), 2) = 5 - \Delta(5, 2) = 5 - (+1) = 4 \quad (2.13)$$

E o direito da faixa:

$$right = f(v_k) - \Delta(f(v_k), 1) = 5 - \Delta(5, 1) = 5 - (-1) = 6 \quad (2.14)$$

3º Passo - busca sequencial.

Baseado nos dados encontrados nas Equação 2.13 e Equação 2.14, esta última etapa do processamento consiste em fazer uma busca sequencial em V, na faixa de índices onde $(4 \leq i \leq 6)$, pela chave $(v_k = 609)$. No exemplo, Figura 4, a chave é encontrada no índice 6 do vetor de dados V.

A título de maiores detalhes, é apresentado na Rotina 2 o pseudocódigo do processamento da BSCL com os devidos comentários.

Rotina 2. Pseudocódigo BSCL: Processamento.

```
Aloc_Dim_Mem                                        #Aloca e Dimensiona memória.
Load_Data                                          #Carrega dados na memória.
tg=(n-1)/v(n-1)                         #Inclinação da reta f(vk) <-- tan(alfa).
p=tg*vk                            #Calcula o índice da chave vk na matriz Delta.
ik=-1                             #Atribui ao indice inicial a condição NIL (-1).
left=p-D(p,2)                  #Faz o cálculo, limite inferior para a busca em V.
right=p-D(p,1)                 #Faz o cálculo, limite superior para a busca em V.
Loop i=left,right                #BUSCA SEQUENCIAL, do índice "left" até "right").
  If (v(i)=vk) Then (ik=i ; Exit)              #Comparação da chave vk com v(i).
End Loop i
```

2.3 Fundamentação Matemática

Na seção 2.3, apresenta-se a fundamentação matemática que serviu de base para a escolha da busca sequencial como motor de busca para a BSCL.

Conforme Ross (2009), a função de probabilidade - advinda de uma distribuição binomial - é dada por:

$$p(x; r, a) = \frac{r!}{x!(r-x)!} a^x (1-a)^{r-x}$$

Onde: a é a probabilidade do evento ocorrer uma vez e $p(x)$ $(x \geq 0)$ a probabilidade do evento ocorrer x vezes em r tentativas. Se é necessário calcular uma probabilidade em que $r \to \infty$ e $a \to 0$ fica difícil resolver a função (ROSS, 2009).

De forma a resolver esta dificuldade, *Siméon-Denis Poisson* fez uma aproximação da função de probabilidade em que usou as condições de $\lim_{r\to\infty}$ e $\lim_{a\to 0}$. Desta aproximação, surgiu a teoria que culminou na equação da distribuição probabilística de *Poisson*. De (ROSS, 2009) resume-se esta equação como sendo: a probabilidade de uma determinada ocorrência, em um conjunto de dados aleatórios, segue a distribuição probabilística de *Poisson* se a sua função de probabilidade puder ser representada por:

$$p(x; \lambda) = \frac{e^{-\lambda}\ \lambda^x}{x!} \tag{2.15}$$

Onde o parâmetro λ, $\lambda\ > 0$, é a taxa de ocorrência por unidade medida, tendo como definição $(\lambda = ra)$.

Em hipótese: se usarmos da distribuição probabilística de *Poisson* $(\lambda = ra)$, sendo $(a = 1/n)$ e n o número de elementos no vetor V, e calcularmos as probabilidades das colisões de x elementos geradas pela Equação 2.5, direcionadas a uma matriz onde $(r = m)$, sendo m o número de registros de $\mathit{\Delta}$ da BSCL, a matriz de colisão pode seguir a distribuição probabilística de *Poisson*. Esta hipótese pode ser comprovada verificando se a distribuição probabilística destas colisões - na prática - segue a equação de *Poisson* [4].

Substituindo o valor de λ na equação de *Poisson*, com os valores acima, tem-se:

$$p(x; m, n) = \frac{e^{-m/n} \left(\dfrac{m}{n}\right)^x}{x!}$$

Uma particularidade na BSCL é o fato de $(m = n)$. Portanto, a função fica simplificada à seguinte fórmula que independe do número de elementos de V e $\mathit{\Delta}$:

$$p(x) = \frac{1}{e \cdot x!} \tag{2.16}$$

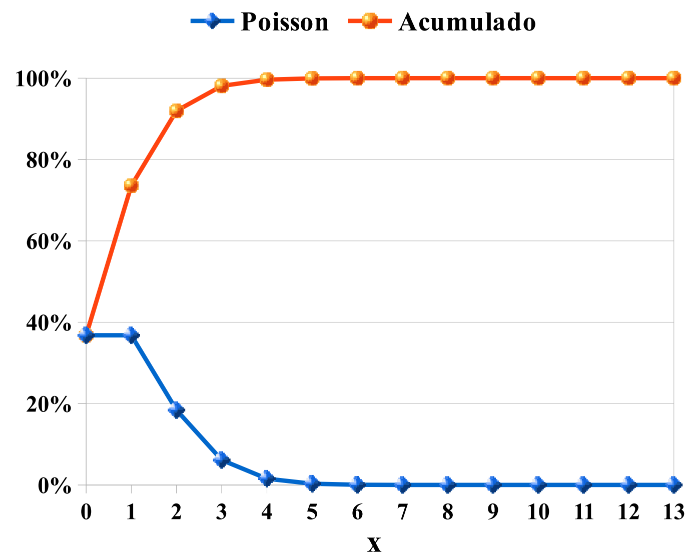
[4] Esta hipótese foi comprovada na seção 3.2, verificando que as distribuições probabilísticas destas colisões - em $\mathit{\Delta}$ e P - seguem a equação de *Poisson*.

Baseado na Equação 2.16 foi desenvolvida a Tabela 2 com as distribuições de probabilidade e seus acumulados - em termos percentuais - até a décima casa após a vírgula para $p(x),\ 0 \leq x \leq 13$. Esta tabela visa subsidiar a análise do comportamento da Equação 2.16 perante os dados de colisões obtidos da Equação 2.5; e como objetivo, fundamentar a escolha do motor de busca para a BSCL.

Tabela 2 – Probab. *Poisson*.

x	$p(x) = \frac{1}{e.x!}\%$	$\sum p(x)\%$
0	36,7879441171	36,7879441171
1	36,7879441171	73,5758882343
2	18,3939720586	91,9698602929
3	6,1313240195	98,1011843124
4	1,5328310049	99,6340153173
5	0,3065662010	99,9405815182
6	0,0510943668	99,9916758851
7	0,0072991953	99,9989750803
8	0,0009123994	99,9998874797
9	0,0001013777	99,9999888575
10	0,0000101378	99,9999989952
11	0,0000009216	99,9999999168
12	0,0000000768	99,9999999936
13	0,0000000059	99,9999999995

Figura 5 – Gráfico de *Poisson*.



Como colocado na seção 2.2, o algoritmo BSCL só procura a chave na faixa de colisão em que a Equação 2.5 o direcionou. Neste contexto, a Tabela 2 pode ser dada, ainda em hipótese[5], como o resultado - para a Equação 2.5 - em termos de probabilidade de colisões, como se segue:

- $p(0) = 36,79\%$ - probabilidade de uma faixa não conter elementos, vazias em Δ.
- $p(1) = 36,79\%$ - probabilidade de uma faixa conter 1 elemento, sem colisão na Δ.
- $p(2) = 18,39\%$ - probabilidade de uma faixa conter 2 elementos em colisão na Δ.
- $p(3) = 6,13\%$ - probabilidade de uma faixa conter 3 elementos em colisão na Δ.
- $\vdots$.
- $p(13) = (6\text{E}{-}9)\%$ - probabilidade de uma faixa conter 13 elementos em colisão na Δ

[5] A validação dos dados usados na experimentação, com a distribuição probabilística de *Poisson*, é apresentada no Capítulo 3.

Na Tabela 2, verifica-se que $\sum_{x=0}^{2} p(x)$ corresponde a aproximadamente 92% de todas as possibilidades de ocorrências em Δ, sendo x o número de elementos em colisão numa faixa - gerado pela Equação 2.5. Pela quantidade de elementos em colisão - máximo dois - já representarem quase a totalidade do universo de colisões, restringiu-se no projeto a um dos dois algoritmos de busca mais simples para motor de busca da BSCL: binária ou sequencial. Do exposto, faz-se as seguintes asserções:

- $p(0) = 36,79\%$ - probabilidade de uma faixa não conter elementos, faixas vazias na Δ. Somente uma iteração é necessária para concluir que o elemento chave não está no vetor V, tanto na busca binária como na sequencial.
- $p(1) = 36,79\%$ - probabilidade de ocorrerem faixas com um elemento, sem colisão na Δ. Também, basta uma única iteração para os dois algoritmos chegarem à conclusão da existência, ou não, da chave em V.
- $p(2) = 18,39\%$ - probabilidade de ocorrerem faixas com dois elementos em colisão na Δ. São necessárias - no máximo - duas interações, independentemente do algoritmo, para chegar a uma decisão quanto à existência, ou não, da chave em V.

A partir destas asserções, conclui-se que: qualquer um dos dois algoritmos escolhidos para motor de busca da BSCL resolveria o problema com o mesmo número de iterações, em 92% dos casos. Entretanto, em termos de processamento, uma busca sequencial executa menos cálculos que uma busca binária para $(0 \leq x \leq 2)$. Consequentemente, menor tempo de processamento é obtido utilizando a busca sequencial. Este foi o motivo da escolha do algoritmo sequencial como motor de busca para a BSCL.

Outro ponto merecedor de fundamentação, na fase de projeto, foi a característica da matriz controle de colisão Δ. Ela, em tese, poderia conter nas suas colunas $(1 \leq j \leq 2)$ as informações $\Delta(f(v_i), 1) = left$ e $\Delta(f(v_i), 2) = right$ (subseção 2.2.1), calculadas durante a fase de pré-processamento. Com isto, não haveria a necessidade de calcular - na fase de processamento - os limites da faixa de colisão através da Equação 2.12 usando os deslocamentos dos índices, o que proporcionaria melhor tempo de processamento no algoritmo BSCL.

Entretanto, existe um detalhe de recurso de memória que pesou na decisão: por usar os deslocamentos dos índices preditos, em relação aos reais da chave no vetor V, e por serem próximos, as colunas $(1 \leq j \leq 2)$ de Δ, calculadas pela Equação 2.8, poderiam ser dimensionadas com a metade de *bytes* que seriam necessários para guardar as informações $left$ e $right$ diretamente. E como os cálculos na Equação 2.8 eram simples, optou-se por esta configuração visando o ganho de memória. Aqui, foi usada a recomendação de (KNUTH, 1998), o uso de rotinas mais simples visando a eficiência na curva custo *versus* benefício.

3 Validação Experimental

Este capítulo tem o objetivo de comprovar, matematicamente e experimentalmente, que os recursos de memória e tempo de processamento são superiores na BSCL se comparado ao algoritmo de *hash* perfeito.

Está estruturado da seguinte forma: na seção 3.1 apresenta-se a metodologia, objetivo, recursos do sistema, bases de dados e as métricas. Na seção 3.2 comprova-se que as matrizes de colisão seguem a distribuição probabilística de *Poisson*. Na seção 3.3 são apresentados os resultados obtidos do protocolo e as discussões. Na seção 3.4 são aprestadas análises complementares ao da seção 3.3.

3.1 Protocolo dos Experimentos

Os experimentos deste trabalho caracterizam-se por análises comparativas entre dois algoritmos (BSCL e *hash* perfeito[1]). A metodologia usada consiste em criar quatorze bases de vetores de dados - de crescimento exponencial - compostos por números inteiros, ordenados e uniformemente distribuídos - criados aleatoriamente - que representam as colunas chaves de matrizes de dados onde os algoritmos têm que buscar chaves específicas de acordo com um vetor de referência (vetor de chaves) - também de números inteiros e aleatórios, porém desordenados[2].

O objetivo dos experimentos consiste em obter dados de três métricas nestas buscas - recursos de memória, número de iterações e tempos de processamento - que serviram de base para validar deduções matemáticas, baseadas na distribuição probabilística de Poisson, sobre a BSCL e o *hash* perfeito.

3.1.1 Recursos de Sistema

As rotinas desenvolvidas neste experimento foram processadas em um computador DELL Inspiron 14R 5437-A40 com processador Intel i7 – 4500U CPU @ 1.8GHz de 4 núcleos e 8GB DDR3 de memória RAM. O Sistema operacional foi o Linux-Ubuntu 16.04.2 LTS - *Xenial Xerus* - com kernel 4.4.0-77-generic.

Como linguagem de programação foram usados *Scripts* do Linux e o *Formula Translation* (Fortran) com compilador Gfortran-6 da GNU *Compiler Collection* (GCC). A compilação foi processada com otimização -O3. Não foram usados recursos de programação

---

[1] O algoritmo de *hash* perfeito, neste experimento, baseia-se no apresentado na seção 1.2.

[2] O vetor chave é desordenado para evitar acesso a memória *cache* no processador.

paralela, de forma a não influenciar os resultados dos motores de busca das rotinas testadas. Todos os dados foram alocados na RAM, evitando tempo de acesso a memória secundária.

3.1.2 Bases de Dados

Desenvolveu-se uma rotina em *Fortran* & *Script* do linux onde se criou, aleatoriamente, um vetor universal $U = (U_{ij})_{2^{28}\times 1}$, $u_{i1} \in \mathbb{N}$ e $u_{i1} \leq (2^{31} - 1)$.[3]

Do vetor universal U extraiu-se - aleatoriamente - os elementos para o vetor de chave $Kg = (Kg_{ij})_{15E6\times 1}$ e para o conjunto de quatorze vetores de dados $\Omega = \{V1, ..., V14\}$, sendo que um vetor $Vd = (Vd_{ij})_{n\times 1},$ tal que $n = 2^{(d+12)}\ \forall d \in \mathbb{N},\ 1 \leq d \leq 14$, onde:

- Vd com elementos $v_i \in U\ \forall i \in \mathbb{N},\ 0 \leq i \leq (n-1)$
- $v_{i-1} < v_i\ \forall i \in \mathbb{N},\ 1 \leq i \leq (n-1)$

3.1.3 Métricas

Usando os vetores do conjunto Ω foi feito o pré-processamento de forma a criar as matrizes Δ da BSCL, conforme seção 2.2, e P do *hash* perfeito, conforme exemplificado na seção 1.2. A partir destas matrizes, foi possível quantificar os recursos de memória em kB - alocada, utilizada e não utilizada - pelos algoritmos de busca da BSCL e *hash* perfeito, conforme fórmulas abaixo:

- **Alocada BSCL**: $4n$, n é o número de registros em Δ. Cada registro de Δ é constituído por 2 colunas, sendo que cada uma requer 2 bytes de alocação.
- **Utilizada BSCL**: $4n'$, n' é o número de registros em Δ os quais foram usados pelo algoritmo para referenciar colisões dos índices i, dos elementos v_i do vetor Vd, na matriz Δ - posições alocadas.
- **Não Utilizada BSCL**: $4(n - n')$, $(n - n')$ é o número de registros em Δ que não foram usados para referenciar deslocamentos dos índices i, dos elementos de Vd.
- **Alocada *hash***: $4n + 4\sum_{i=0}^{n-1} p_{i0}$, n é o número de registros na matriz primária P. Cada registro de P é constituído por 2 colunas, sendo que cada uma requer 2 bytes de alocação. O segundo termo desta equação corresponde à somatória da memória alocada nos vetores secundários St - 4 bytes por célula.
- **Utilizada *hash***: $4n'' + 4\sum_{i=0}^{n-1} \sqrt{p_{i0}}$, n'' é o número de registros em P que foram realmente usados pelo algoritmo para referenciar resultados da função *hash* H_p para os índices i dos elementos v_i do vetor Vd.

[3] $(2^{31} - 1)$ é o maior número inteiro positivo de 4 bytes - 2.147.483.647.

- **Não Utilizada *hash***: $4(n-n'') + 4\sum_{i=0}^{n-1}(p_{i0} - \sqrt{p_{i0}})$, $(n-n'')$ é o número de registros em P que não foram usados para referenciar resultados da função *hash* H_p para os índices i do vetor Vd.

Dos vetores, de chaves Kg e de Ω, duas outras métricas foram obtidas diretamente no experimento - o número de iterações e o tempo de processamento - sendo esta, através da mediana de 25 medições para cada vetor de Ω. Cabe ressaltar que no experimento não foram levados em consideração: o tempo de pré-processamento para o *hash* perfeito; e na BSCL o de ordenação dos vetores de Ω e o de pré-processamento. Para os algoritmos, as fases de pré-processamento e ordenação têm a função de criar as bases que servirão de repositórios para os futuros processamentos. Sendo estes, o ponto central dos algoritmos.

3.2 Validação dos Dados do Experimento com *Poisson*

Nesta seção é descrita a validação de que as matrizes de colisão do experimento seguem a distribuição probabilística de *Poisson*. Inicia-se, verificando na Tabela 2 que a $\sum_{x=0}^{3} p(x)$ corresponde - probabilisticamente - a mais de 98% das possíveis ocorrências. Contabilizando os números de faixas de colisões nas matrizes Δ e P em que $0 \leq x \leq 3$, para todos os vetores do conjunto Ω, e comparando-os com os resultados teóricos obtidos na Equação 2.16, e sendo o desvio pequeno, pode-se comprovar que: em mais de 98% dos casos de colisões, as matrizes Δ e P seguem a distribuição probabilística de *Poisson*. Desta contagem, foram criadas as Tabela 3 e Tabela 4.

Tabela 3 – Distribuição de Colisões, $x \in \{0, 1\}$

Vd	$x=0$				$x=1$			
	Hash Perfeito		**BSCL**		**Hash Perfeito**		**BSCL**	
n	cH	$\frac{cH.e.x!}{n}$	cB	$\frac{cB.e.x!}{n}$	cH	$\frac{cH.e.x!}{n}$	cB	$\frac{cB.e.x!}{n}$
2^{13}	2.982	-1,05%	3.020	-1,61%	3.059	0,21%	2.965	-1,61%
2^{14}	5.981	-0,77%	6.126	1,64%	6.059	0,53%	5.887	-2,33%
2^{15}	11.963	-0,76%	12.149	0,78%	12.183	1,06%	11.890	-1,37%
2^{16}	24.118	0,04%	24.056	-0,22%	24.086	-0,10%	24.301	0,79%
2^{17}	48.284	0,14%	48.447	0,47%	48.080	-0,29%	47.898	-0,67%
2^{18}	96.440	0,00%	96.487	0,05%	96.528	0,09%	96.340	-0,10%
2^{19}	192.528	-0,18%	192.959	0,04%	193.323	0,23%	192.974	0,05%
2^{20}	385.166	-0,15%	385.207	-0,14%	386.777	0,27%	386.348	0,16%
2^{21}	778.161	0,86%	770.658	-0,11%	764.003	-0,97%	772.676	0,15%
2^{22}	1.543.517	0,03%	1.540.400	-0,17%	1.541.949	-0,07%	1.546.618	0,23%
2^{23}	3.080.032	-0,19%	3.080.832	-0,17%	3.091.957	0,19%	3.090.456	0,14%
2^{24}	6.171.788	0,00%	6.147.753	-0,39%	6.171.667	-0,01%	6.197.088	0,41%
2^{25}	12.247.358	-0,78%	12.246.341	-0,79%	12.441.156	0,79%	12.442.972	0,80%
2^{26}	24.334.851	-1,43%	24.300.083	-1,57%	25.040.604	1,43%	25.075.773	1,57%
Me:		-0,17%		-0,15%		0,20%		0,15%

Tabela 4 – Distribuição de Colisões, $x \in \{2, 3\}$

Vd	$x = 2$				$x = 3$			
	Hash Perfeito		**BSCL**		**Hash Perfeito**		**BSCL**	
n	cH	$\frac{cH.e.x!}{n}$	cB	$\frac{cB.e.x!}{n}$	cH	$\frac{cH.e.x!}{n}$	cB	$\frac{cB.e.x!}{n}$
2^{13}	1.505	-0,12%	1.579	4,79%	495	-1,45%	484	-3,64%
2^{14}	3.074	2,00%	3.004	-0,32%	974	-3,04%	1.048	4,32%
2^{15}	6.034	0,11%	6.086	0,97%	1.973	-1,80%	2.030	1,04%
2^{16}	12.080	0,21%	11.888	-1,38%	3.989	-0,73%	3.985	-0,83%
2^{17}	24.138	0,12%	24.081	-0,12%	8.116	0,99%	8.140	1,29%
2^{18}	48.049	-0,35%	48.190	-0,06%	16.167	0,59%	16.219	0,91%
2^{19}	96.492	0,06%	96.142	-0,31%	32.045	-0,31%	32.115	-0,10%
2^{20}	192.595	-0,15%	193.033	0,08%	64.063	-0,36%	64.235	-0,09%
2^{21}	383.418	-0,60%	385.826	0,02%	129.817	0,96%	128.238	-0,27%
2^{22}	771.828	0,04%	771.326	-0,02%	257.620	0,18%	256.733	-0,17%
2^{23}	1.546.124	0,20%	1.546.286	0,21%	513.121	-0,24%	514.002	-0,06%
2^{24}	3.086.235	0,01%	3.096.940	0,35%	1.029.393	0,07%	1.024.634	-0,39%
2^{25}	6.220.780	0,79%	6.218.487	0,75%	2.039.094	-0,89%	2.041.908	-0,75%
2^{26}	12.523.766	1,46%	12.541.036	1,60%	4.054.182	-1,47%	4.047.954	-1,62%
Me:		0,08%		0,05%		-0,33%		-0,13%

Nas Tabela 3 e Tabela 4 são mostradas as contagens de $p(x)$, $0 \leq x \leq 3$, tanto para P da tabela *hash* perfeito (cH) como para o Δ da BSCL (cB) para cada um dos vetores de Ω, perfazendo um total de mais de 98% do universo de ocorrências. Em seguida, foram calculadas as variações de cH e cB em relação a Equação 2.16 dadas por: $\frac{cH}{n} / \frac{1}{e \cdot x!} = \frac{cH \cdot e \cdot x!}{n}$ e $\frac{cB}{n} / \frac{1}{e \cdot x!} = \frac{cB \cdot e \cdot x!}{n}$ em termos percentuais para uma ocorrência, respectivamente. Posteriormente, foi calculada a mediana para cada algoritmo e em cada valor de x.

Da análise, baseada na mediana das variações, chega-se a conclusão que as colisões em Δ e P divergem dos cálculos teóricos - Equação 2.16 - no máximo em 0,15% e 0,33%, respectivamente. Portanto, pode-se afirmar - pelas pequenas divergências encontradas, perante o grande número de elementos nos vetores de Ω - que a matriz Δ e P seguem a distribuição probabilística de *Poisson*.

Esta condição já era esperada pelo fato dos vetores de Ω serem de tamanhos consideráveis e terem elementos criados de forma aleatória, não influenciados pelos outros elementos já criados, condições primordiais para as análises estarem sob a distribuição probabilística de *Poisson* (ROSS, 2009).

3.3 Resultados e Discussões

Esta seção trata dos resultados e discussões dos experimentos, baseados no protocolo abordado na seção 3.1. Está dividida em três subseções, como se segue: na subseção 3.3.1 analisa-se o uso de memória para o funcionamento dos algoritmos *hash* perfeito e BSCL; na subseção 3.3.2 a análise abrange o número de iterações[4] necessárias para buscar as chaves de Kg em Vd; na subseção 3.3.3 aborda-se o tempo de processamento, para os dois algoritmos.

Cabe ressaltar que as subseção 3.3.1 e subseção 3.3.2 possuem uma estrutura que se fundamenta, primeiramente, numa demonstração matemática baseada na distribuição probabilística de *Poisson*, tendo como base as fórmulas deduzidas no Apêndice A - citadas quando utilizadas. Posteriormente, é feita a validação desta demonstração com os resultados obtidos no experimento. Com esta estrutura, pôde-se comparar as métricas da BSCL em relação ao algoritmo de referência, tabela *hash* perfeito.

3.3.1 Recursos de Memória

Uma vez que as matrizes de colisão da BSCL e do *hash* perfeito seguem a distribuição probabilística de *Poisson*, os recursos de memória podem ser deduzidos da Equação 2.16 - com o auxílio das fórmulas do Apêndice A - e validados pelos dados obtidos nas rotinas.

Os recursos de memória usados, por ambos algoritmos, se enquadram em três categorias a saber: memória que foi alocada, memória realmente utilizada e a porção de memória que é desperdiçada por cada um. Disto, fez-se as deduções que se seguem:

- **Memória alocada pelo *hash* perfeito**.

  Conforme descrito na subseção 3.1.3, o algoritmo *hash* perfeito aloca memória para P e St. A matriz P, constituída de 2 colunas, tem o mesmo número de registros existentes em Vd, sendo que cada célula consome 2 bytes. Diante o exposto, a quantidade de memória alocada para P, onde n é o número de registros em P, é dada por:

  $$n^{\underline{o}}\ colunas \cdot n^{\underline{o}}\ bytes \cdot n = 2 \cdot 2 \cdot n = 4n \tag{3.1}$$

  Já os vetores St só alocam memória quando os respectivos registros de P estiverem em uso[5]. Esta condição é dada pela equação de probabilidade de *Poisson*, Equação 2.16, quando $x \geq 1$.

  Como cada célula de St aloca 4 bytes[6] e a quantidade de células é igual a x^2, sendo x o número de elementos dentro de uma faixa de colisão (CORMEN et al.,

[4] Reiterando, nesta dissertação o termo iteração corresponde ao número de vezes que os algoritmos terão que acessar à base de dados para ratificar a existência, ou não, da chave que se procura.
[5] Valor de $x = 0$ caracteriza elementos vazios em P - não há saída pela função *hash* $Hp(v_i)$ da Figura 1.
[6] O experimento baseou em números inteiros de 4 bytes.

2009), a quantidade de memória alocada para St - para as infinitas colisões de $p(x)$ (Equação 2.16) - é dada pela fórmula[7]:

$$4n \lim_{n\to\infty} \sum_{x=0}^{n} x^2 \cdot p(x) = 4n \lim_{n\to\infty} \sum_{x=0}^{n} \frac{x^2}{e \cdot x!} = \frac{4n}{e} \lim_{n\to\infty} \sum_{x=0}^{n} \frac{x^2}{x!}$$

Substituindo a Equação A.6, na fórmula acima, tem-se:

$$4n \lim_{n\to\infty} \sum_{x=0}^{n} x^2 \cdot p(x) = \frac{4n}{e} \cdot 2e = 8n \tag{3.2}$$

Então, a quantidade de memória alocada para o algoritmo *hash* perfeito, pela distribuição de probabilidade de *Poisson*, é dada pela somatória da Equação 3.1 e da Equação 3.2.

$$MemAlocHash = 4n + 8n = 12n \tag{3.3}$$

Advêm-se que - em termos de memória - o algoritmo de *hash* perfeito é da ordem $\mathcal{O}(n)$, vindo ao encontro das afirmações de (CORMEN et al., 2009).

- **Memória alocada pela BSCL**.

A BSCL - representada pela matriz de colisões Δ - é constituída de n registros com 2 colunas, sendo que cada célula possui 2 bytes. A alocação de memória para esta matriz segue a mesma quantidade da P, dada pela Equação 3.1, e é também da ordem $\mathcal{O}(n)$ em complexidade de memória.

$$MemAlocBSCL = 4n \tag{3.4}$$

Dos dados acima, é possível quantificar a memória alocada pelo *hash* perfeito, em relação à BSCL, fazendo a divisão da Equação 3.3 pela Equação 3.4:

$$\frac{MemAlocHash}{MemAlocBSCL} = \frac{12n}{4n} = 3 \tag{3.5}$$

Pela Equação 3.5, conclui-se que: o *hash* perfeito aloca **200%** a mais de memória, do que a BSCL, para executar suas rotinas.

- **Memória utilizada pelo *hash* perfeito**.

O *hash* perfeito utiliza memória em P e St. A memória utilizada em P, conforme distribuição de Poisson, se dá quando $x \geq 1$ [8] na Equação 2.16.

Diante o colocado, a quantidade de memória utilizada por P, para as infinitas possibilidades de colisões, é dada pela seguinte equação:

$$4n \lim_{n\to\infty} \sum_{x=1}^{n} p(x) = 4n \lim_{n\to\infty} \sum_{x=1}^{n} \frac{1}{e \cdot x!} = \frac{4n}{e} \lim_{n\to\infty} \sum_{x=1}^{n} \frac{1}{x!}$$

[7] Foi mantido o valor inicial de $x = 0$, na somatória, pelo fato de não influenciar o resultado.
[8] Condição de faixas alocadas pelo algoritmo (CORMEN et al., 2009).

A fórmula acima pode ser escrita como:

$$\frac{4n}{e}\left(\lim_{n\to\infty}\sum_{x=0}^{n}\frac{1}{x!}-\sum_{x=0}^{0}\frac{1}{x!}\right) \tag{3.6}$$

Usando a Equação A.1 na 1ª somatória e calculando a 2ª, na Equação 3.6, tem-se:

$$4n\lim_{n\to\infty}\sum_{x=1}^{n}p(x)=\frac{4n}{e}(e-1) \tag{3.7}$$

Já quanto a St, a quantidade de memória usada é dada pela fórmula[9]:

$$4n\lim_{n\to\infty}\sum_{x=1}^{n}x.p(x)=4n\lim_{n\to\infty}\sum_{x=1}^{n}\frac{x}{e\cdot x!}=\frac{4n}{e}\lim_{n\to\infty}\sum_{x=1}^{n}\frac{x}{x!}$$

Usando a Equação A.2 tem-se:

$$4n\lim_{n\to\infty}\sum_{x=1}^{n}x.p(x)=4n \tag{3.8}$$

Então, o total de memória utilizado pelo algoritmo *hash* perfeito é dado pela somatória da Equação 3.7 e da Equação 3.8, e sendo $e \approx 2,718$ tem-se:

$$MemUsadaHash=\frac{4n}{e}(e-1)+4n\approx 6,528n \tag{3.9}$$

- **Memória utilizada pela BSCL**:

  No caso da Δ da BSCL, como na P do *hash* perfeito - já que ambas matrizes estão sujeitas à distribuição de *Poisson* e possuem o mesmo número de registros -, a memória utilizada pela BSCL é dada pela Equação 3.7:

$$MemUsadaBSCL=\frac{4n}{e}(e-1)\approx 2,528n \tag{3.10}$$

  Dos dados acima, é possível quantificar a memória utilizada pelo *hash* perfeito, em relação à BSCL, fazendo a divisão da Equação 3.9 pela Equação 3.10:

$$\frac{MemUsadaHash}{MemUsadaBSCL}=\frac{6,528n}{2,528n}\approx 2,58 \tag{3.11}$$

  Pela Equação 3.11, conclui-se que: o *hash* perfeito usa **158%** a mais de memória, do que a BSCL, para executar suas rotinas.

- **Memória não utilizada pelo *hash* perfeito**.

  A memória não utilizada é dada, diretamente, pela subtração das memórias: alocada - Equação 3.3 - e utilizada - Equação 3.9:

$$12n-6,528n=5,472n \tag{3.12}$$

[9] Vide Figura 1 da seção 1.2.

- **Memória não utilizada pela BSCL**.

  Para a BSCL, é dada pela subtração das memórias: alocada - Equação 3.4 - e utilizada - Equação 3.10:

$$4n - 2,528n = 1,472n \tag{3.13}$$

  Novamente, fazendo a divisão do valor da Equação 3.12 pelo da Equação 3.13 tem-se o valor relativo da memória não usada pelo *hash* perfeito em relação ao BSCL:

$$\frac{5,472n}{1,472n} \approx 3,72 \tag{3.14}$$

  Pela Equação 3.14, conclui-se que: o *hash* perfeito desperdiça **272%** a mais de memória, do que a BSCL, para executar suas rotinas.

De forma a validar os cálculos acima, foi feito o levantamento de todas as memórias - alocadas, usadas e não utilizadas - das matrizes de colisão dos algoritmos, conforme metodologia abordada na subseção 3.1.3. Na Tabela 5 são apresentados os resultados obtidos, em kB - $Hash$ (*hash* perfeito) e $BSCL$, referentes às execuções das rotinas do experimento. Os dados apresentados são referenciados à quantidade de memória envolvida na BSCL - $\frac{Hash}{BSCL}$ -, para as bases dos vetores de Ω.

Tabela 5 – Recursos de Memória - em kB.

Vd	**Memória Alocada**			**Memória Usada**			**Memória Não Usada**		
n	$Hash$	$BSCL$	$\frac{Hash}{BSCL}$	$Hash$	$BSCL$	$\frac{Hash}{BSCL}$	$Hash$	$BSCL$	$\frac{Hash}{BSCL}$
2^{13}	95	32	2,97	52	20	2,60	43	12	3,58
2^{14}	191	64	2,98	104	40	2,60	87	24	3,63
2^{15}	382	128	2,98	209	80	2,61	173	48	3,60
2^{16}	767	256	3,00	417	162	2,57	350	94	3,72
2^{17}	1.535	512	3,00	835	322	2,59	700	190	3,68
2^{18}	3.073	1.024	3,00	1.671	647	2,58	1.402	377	3,72
2^{19}	6.137	2.048	3,00	3.343	1.294	2,58	2.794	754	3,71
2^{20}	12.282	4.096	3,00	6.687	2.591	2,58	5.595	1.505	3,72
2^{21}	24.702	8.192	3,02	13.344	5.181	2,58	11.358	3.011	3,77
2^{22}	49.155	16.384	3,00	26.738	10.366	2,58	22.417	6.018	3,72
2^{23}	98.179	32.768	3,00	53.504	20.733	2,58	44.675	12.035	3,71
2^{24}	196.585	65.536	3,00	106.963	41.521	2,58	89.622	24.015	3,73
2^{25}	391.187	131.072	2,98	214.302	83.234	2,57	176.885	47.838	3,70
2^{26}	778.989	262.144	2,97	429.229	167.221	2,57	349.760	94.923	3,68
$Me:$			3,00			2,58			3,71

Pela mediana, conclui-se que a quantidade de memória alocada pelo *hash* perfeito - em relação a BSCL - é 3,00; a usada é 2,58; a não usada é 3,71. Ambos os resultados vêm ao encontro dos cálculos teóricos obtidos nas Equação 3.5, Equação 3.11 e Equação 3.14, respectivamente, validando-os.

- **Eficiência de memória**.

Com base nos dados das memórias - utilizadas e alocadas - é possível achar a eficiência no uso deste recurso e concluir qual algoritmo possui o melhor aproveitamento. As razões das memórias utilizadas em relação às alocadas, para cada um dos algoritmos, são possíveis de serem obtidas diretamente dos resultados das equações acima.

No caso da BSCL, é dada pela razão da Equação 3.10 pela Equação 3.4:

$$EficMemBSCL = \frac{MemUsadaBSCL}{MemAlocadaBSCL} = \frac{2,528n}{4n} = 0,632 = 63,2\% \tag{3.15}$$

Já para o *hash* perfeito, é dado pela razão da Equação 3.9 pela Equação 3.3:

$$EficMemHash = \frac{MemUsadaHash}{MemAlocadaHash} = \frac{6,528n}{12n} = 0,544 = 54,4\% \tag{3.16}$$

Destes dados, é possível deduzir a eficiência de memória da BSCL em relação ao *hash* perfeito:

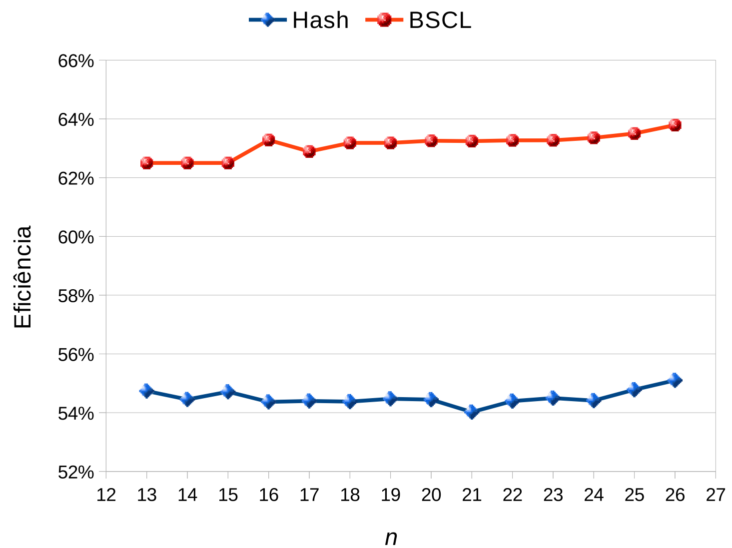
$$EficMem = EficMemBSCL - EficMemHash = 0,632 - 0,544 = 0,088 = 8,8\% \tag{3.17}$$

Na Tabela 6, juntamente com a Figura 6, é feita uma apresentação da eficiência quanto ao uso de memória em função do número de elementos de Vd, obtida da Tabela 5.

Tabela 6 – Eficiência de Memória

n	*Hash*%	BSCL%
2^{13}	54,74	62,50
2^{14}	54,45	62,50
2^{15}	54,71	62,50
2^{16}	54,37	63,28
2^{17}	54,40	62,89
2^{18}	54,38	63,18
2^{19}	54,47	63,18
2^{20}	54,45	63,26
2^{21}	54,02	63,24
2^{22}	54,40	63,27
2^{23}	54,50	63,27
2^{24}	54,41	63,36
2^{25}	54,78	63,50
2^{26}	55,10	63,79
$Me:$	54,45	63,25

Figura 6 – Eficiência no Uso de Memória.



Os valores percentuais apresentados - *Hash*% e BSCL% - correspondem à razão das memórias utilizadas em relação às alocadas, para cada um dos algoritmos. Através das

medianas da Tabela 6 é possível validar os dados obtidos nas Equação 3.15, Equação 3.16 e Equação 3.17. Conclui-se que existe uma eficiência, quanto ao uso de memória, na ordem de 9%, da BSCL em relação ao *hash* perfeito.

Como se denota, através dos cálculos teóricos e da quantificação de memórias, a BSCL tem uma melhor eficiência quanto ao uso destes recursos. Isto se deve ao fato da BSCL não usar vetores secundários de forma a guardar todos os índices dos dados dos vetores de Ω - ela guarda o primeiro e último índice dentro de cada faixa de colisão na matriz de colisão Δ. Soma-se a isto, o fato do *hash* perfeito necessitar de alocação de memória nos vetores St, de forma quadrática, para conseguir a probabilidade maior do que 50% de não haver colisão (CORMEN et al., 2009).

3.3.2 Números de Iterações

Da mesma forma que nos cálculos dos recursos de memória pôde ser usada a distribuição probabilística de Poisson, nesta métrica pode ser usada para calcular o número de iterações da BSCL, tendo como referência a Tabela 2 e a Figura 5.

Os elementos do vetor chave Kg, aleatórios, podem estar presentes, ou não, nos vetores do conjunto Ω. Disto, o algoritmo BSCL pode ter que procurar as chaves em todas as faixas de colisão, referenciadas pela matriz Δ, onde $x \geq 0$.

A Equação 2.5 direciona a rotina de busca sequencial para um índice da faixa de colisão em Δ, em que $x \geq 1$, a rotina pode ter que comparar a chave de Kg com todos[10] os x elementos da faixa de colisão no vetor Vd. Consequentemente, a função matemática que representa o número de iterações ($A(x)$) pela busca sequencial, nestas faixas, para todos os seus elementos, pode ser dada por:

$$A(x) = x \ , x \geq 1$$

Existe uma particularidade quanto à faixa em que $x = 0$ (vazia), o número de iterações não é função de x. Basta uma única iteração para saber que o elemento não está no faixa. Portanto, a função matemática em $x = 0$ é dada por $A(0) = 1$.

Pela distribuição de Poisson, para cada faixa de x onde $0 \leq x \leq \infty$, existe uma probabilidade de colisão dada por $p(x)$ - Equação 2.16. Então, o número total de iterações para achar os elementos de Kg em Vd, probabilisticamente, é dado por:

$$p(0).A(0) + \lim_{n \to \infty} \sum_{x=1}^{n} p(x) \cdot A(x) \tag{3.18}$$

[10] Ressalta-se que pela natureza dos dados de K_g e V_d, extraídos do vetor universal U - com muito mais elementos do que K_g e V_d -, a probabilidade do algoritmo não encontrar a chave é muito maior do que encontra-la.

Substituindo os valores de $A(0)$, $A(x)$, $p(0)$ e $p(x)$ na Equação 3.18, tem-se:

$$\frac{1}{e} + \lim_{n \to \infty} \sum_{x=1}^{n} \frac{x}{e \cdot x!}$$

Como e é a constante de Euler pode-se escrever o limite como:

$$\frac{1}{e} + \frac{1}{e} \lim_{n \to \infty} \sum_{x=1}^{n} \frac{x}{x!} \tag{3.19}$$

Usando a Equação A.2 na Equação 3.19, tem-se:

$$\frac{1}{e} + \frac{1}{e} \cdot e$$

Sendo $e \approx 2,718$ pode-se afirmar que:

$$\frac{1}{e} + \frac{1}{e} \lim_{n \to \infty} \sum_{x=1}^{n} \frac{x}{x!} \approx \mathbf{1{,}37} \tag{3.20}$$

De forma a validar os cálculos teóricos do número de iterações, foi feito um levantamento de todas as iterações necessárias para achar as chaves procuradas do vetor Kg em cada base de vetor de Ω - para os dois algoritmos. Estes dados são os apresentados na Tabela 7 através dos índices: iH e iB; juntamente com o número de elementos n e q dos respectivos vetores Vd e Kg.

Tabela 7 – Número de Iterações - em Acessos à Base de Dados.

Vd ; Kg	**Hash Perfeito**		**BSCL**	
n ; q	iH	$\frac{iH}{q}$	iB	$\frac{iB}{q}$
2^{13} ; 15E6	15.000.000	1,00	20.526.470	1,37
2^{14} ; 15E6	15.000.000	1,00	20.603.474	1,37
2^{15} ; 15E6	15.000.000	1,00	20.559.237	1,37
2^{16} ; 15E6	15.000.000	1,00	20.502.230	1,37
2^{17} ; 15E6	15.000.000	1,00	20.545.822	1,37
2^{18} ; 15E6	15.000.000	1,00	20.524.686	1,37
2^{19} ; 15E6	15.000.000	1,00	20.518.085	1,37
2^{20} ; 15E6	15.000.000	1,00	20.503.779	1,37
2^{21} ; 15E6	15.000.000	1,00	20.504.774	1,37
2^{22} ; 15E6	15.000.000	1,00	20.487.201	1,37
2^{23} ; 15E6	15.000.000	1,00	20.473.684	1,36
2^{24} ; 15E6	15.000.000	1,00	20.435.746	1,36
2^{25} ; 15E6	15.000.000	1,00	20.359.848	1,36
2^{26} ; 15E6	15.000.000	1,00	20.205.532	1,35
Me:		1,00		1,37

Como era de se esperar, o número de iterações para achar uma chave de Kg em Vd não altera no *hash* perfeito, é sempre 1,00 (CORMEN et al., 2009) - conforme representado em $\frac{iH}{q}$ da Tabela 7.

O número de iterações na BSCL se mostrou, dentro da faixa do experimento, não depender do número de elementos das bases Vd. Este número é representado em $\frac{iB}{q}$ na tabela. Pela mediana da Tabela 7 está na ordem de $1,37$ iteração para achar uma chave de Kg em Vd, validando o valor encontrado na Equação 3.20.

Observa-se, através das medianas, que o BSCL gasta - em média - 37% a mais de iteração do que o *hash* perfeito para achar um elemento de Kg em Vd. Nesta métrica, a BSCL demonstrou estar na mesma ordem de grandeza do algoritmo *hash* perfeito, uma vez que o número de iterações - também - não depende do número de elementos de Vd. Ambos demonstram ser $\mathcal{O}(1)$ no caso médio.

3.3.3 Tempo de Processamento

Conforme descrito na seção 3.1, foram obtidos os tempos de processamento baseados nas medianas de 25 medições - tempo necessário para procurar todas as chaves de Kg nos vetores de Ω. Os tempos estão representados nas colunas tH e tB na Tabela 8, em μs.

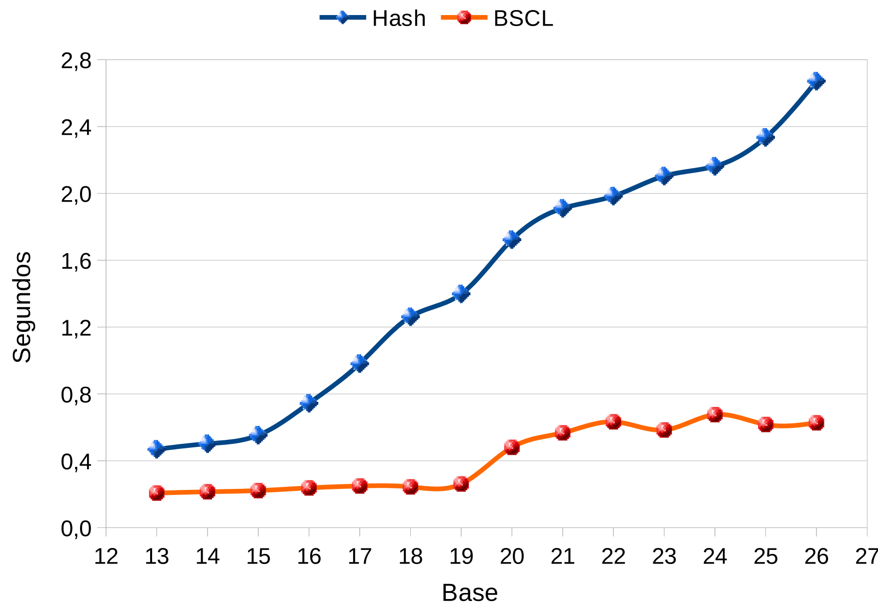
Tabela 8 – Tempos de Processamento - em μs.

Vd Kg	**Hash Perfeito**		**BSCL**	
n ; q	tH	$\frac{tH}{tB}$	tB	$\frac{tB}{tB}$
2^{13} ; 15E6	468.886	2,27	206.865	1,00
2^{14} ; 15E6	501.917	2,34	214.490	1,00
2^{15} ; 15E6	553.052	2,49	221.804	1,00
2^{16} ; 15E6	744.036	3,13	237.824	1,00
2^{17} ; 15E6	981.434	3,94	249.040	1,00
2^{18} ; 15E6	1.261.772	5,17	243.933	1,00
2^{19} ; 15E6	1.398.610	5,35	261.322	1,00
2^{20} ; 15E6	1.723.507	3,59	480.538	1,00
2^{21} ; 15E6	1.910.730	3,37	567.295	1,00
2^{22} ; 15E6	1.983.913	3,13	633.177	1,00
2^{23} ; 15E6	2.103.879	3,60	584.443	1,00
2^{24} ; 15E6	2.162.517	3,20	676.009	1,00
2^{25} ; 15E6	2.335.290	3,78	616.998	1,00
2^{26} ; 15E6	2.671.494	4,26	627.106	1,00
$\overline{M}$:		3,57		1,00

Na Tabela 8, foram calculados os tempos de tH e tB em relação ao tempo tB do algoritmo BSCL. Este cálculo teve como objetivo encontrar quantas vezes o algoritmo BSCL é mais rápido do que ao *hash* perfeito - $\frac{tH}{tB}$ - no processamento. A coluna $\frac{tB}{tB}$ serviu de referência na tabela. Os tempos apresentados estão em (μs) para a procura das chaves de Kg, constituído de 15E6 elementos, nas bases variáveis dos vetores de Ω. Denota-se que: embora o algoritmo BSCL necessite de 37% a mais de iterações, no caso médio, não impacta no tempo de processamento quando comparado ao *hash* perfeito.

De forma a visualizar graficamente os dados da Tabela 8 foi construída a Figura 7 que ilustra o tempo de processamento, em segundos, dos dois algoritmos:

Figura 7 – Tempo de Processamento.



Na Figura 7 observa-se uma mudança na inclinação do tempo de processamento - para o *hash* perfeito - a partir da base 2^{15}; e que a mesma mudança, em menor grau, só é sentida na BSCL a partir da base 2^{19} - 16 vezes maior. Então, embora o *hash* perfeito necessite de apenas 1 (uma) iteração para achar um elemento da chave Kg em Vd - em qualquer caso - denota-se que o tempo de processamento para achar a chave está mais na dependência no número de elementos n de Vd do que se comparado ao BSCL.

3.4 Considerações Complementares

Nesta seção são apresentadas algumas considerações complementares, baseadas em outras averiguações, às apresentadas nas seções anteriores.

Em se tratando de *cache* no processador, na BSCL a procura é feita somente na faixa de colisão e sendo os elementos do vetor Vd agrupados e ordenados na memória, a primeira leitura à RAM - na procura de um determinado elemento da faixa - tende a colocar os demais em *cache* no processador devido à proximidade na memória durante a leitura do bloco. Daí para frente, o processamento da rotina de busca sequencial tende a ser extremamente rápida, pois não há necessidade de procurar, novamente, os demais elementos da faixa na RAM - existe uma maior probabilidade de estarem em *cache*. No caso do *hash* perfeito será irrelevante, pois o algoritmo fará - em qualquer caso - somente uma única iteração para concluir da existência, ou não, da referida chave no vetor Vd.

Tudo indica que este possa ter sido um dos motivos do desempenho computacional na BSCL - para o quesito tempo de processamento - não ter sido influenciado, de forma significativa, pelas faixas com maior número de colisões. Atrelado a isto, obviamente, a menor probabilidade de ocorrências de faixas com muitas colisões, conforme Equação 2.16.

Reafirma-se o fato da Equação 2.5 - da BSCL - ser de tempo computacional inferior se comparado à necessidade de duas funções *hash* em cascata, como no *hash* perfeito do experimento. A rotina da BSCL faz somente uma multiplicação e duas subtrações para delimitar o posicionamento na faixa de colisão, enquanto o *hash* perfeito tem que passar por duas equações em cascata envolvendo funções MOD - muito mais custosa do que a equação linear da BSCL.

Outro destaque - na memória - é o número de metadados usados pelos vetores St. Para cada um, no caso do *Fortran*, são gastos 64 bytes. Como a quantidade de metadados é diretamente proporcional ao número de registros da matriz P em que $p_{m0} \neq NULL$ - pois a cada registro preenchido de P é associado um vetor St - a quantidade de metadados torna-se excessivamente grande para as bases maiores de Vd. Consequentemente, maior custo para o controle - por parte do sistema operacional - no gerenciamento da memória.

Esta ocorrência foi verificada - de forma expressiva - no Vd de base 2^{26}, o qual consumiu praticamente todos os 8 *Gbytes* do computador no processamento das rotinas *hash* perfeito. Este problema não ocorreu na execução da BSCL, bastou um único metadado de 64 bytes para referenciar a matriz Δ. Este fator pode impactar no tempo de processamento das rotinas do *hash* perfeito. Cabe ressaltar que não foi possível trabalhar no *hash* perfeito com bases superiores a 2^{26} elementos. Em pré-testes, a BSCL teria condições de trabalhar com bases muito superiores a 2^{26}, limite do experimento.

Na função *hash* secundária - $Hs(v_i)$ do *hash* perfeito - para que não haja colisão nos vetores St, conforme Cormen et al. (2009), a escolha da função é feita por "experiências aleatórias". Segundo os autores, se escolhermos $Hs(v_i)$, ao acaso, de uma classe universal de funções *hash*, a probabilidade de colisões no vetor St da Figura 1 é menor do que 50%. Entretanto, pode haver colisão e será necessário escolher uma nova função *hash*. No caso da BSCL não existe "experiências aleatórias", já se sabe de antemão qual é a função de predição pelo fato da Equação 2.5 depender única e exclusivamente do maior valor de Vd, existente em v_{n-1}, e do valor de $n-1$.

O *hash* perfeito - no quesito número de iterações - é da ordem $\mathcal{O}(1)$ no pior caso. A BSCL é da ordem $\mathcal{O}(x_{max})$, sendo x_{max} o número de elementos na faixa de colisão no pior caso. A probabilidade de colisão - para os dados analisados - é dada pela distribuição probabilística de *Poisson* (Equação 2.16). Como a quantidade de elementos dentro de uma faixa de colisão tende a ser pequena e não dependente do número de elementos no vetor ordenado, como se verifica na Tabela 2 e Figura 5, a BSCL tende a ser da mesma ordem de grandeza do *hash* perfeito neste quesito. Tanto que o número de iterações (caso médio)

no experimento foi constante - 1,37 - para o conjunto Ω de crescimento exponencial.

As principais diferenças, do *hash* perfeito e da BSCL, podem ser assim resumidas:

- A BSCL tem como condição para um processamento eficiente a necessidade de que as colunas de chaves sejam uniformemente distribuídas. Já o *hash* perfeito não necessita desta uniformidade;
- a BSCL foi projetado para trabalhar "somente" com colunas de chaves ordenadas, enquanto o *hash* perfeito pode trabalhar com "qualquer" tipo de colunas de chaves;
- a BSCL, pela necessidade de ordenação das colunas de chaves, adquire uma característica intrínseca que é a facilidade de procurar faixas de valores de chaves que satisfaçam determinadas características. No *hash* perfeito, sob condições normais de funcionamento, a procura de faixa tem que ser por busca exata - caracterizando como sendo um algoritmo não projetado para este fim;
- o *hash* perfeito usa o princípio de "dispersão" dos índices ao longo da matriz P; enquanto o BSCL usa o da "concentração" ao longo da matriz Δ;
- o *hash* perfeito baseia-se em 2 funções, usando módulos, para localizar os elementos das chaves Kg no respectivo Vd; enquanto o BSCL usa uma simples equação linear para delimitar a área da memória que alocam os respectivos índices da faixa de colisão, onde é possível encontrar as chaves Kg no respectivo Vd;
- o *hash* perfeito usa somente funções de *hash*; enquanto a BSCL usa um misto de busca por interpolação (quando usa a equação de 1º grau para achar o endereço da faixa de colisão) e uma busca sequencial baseada nos índices alocados nesta faixa.

Na Tabela 9, é feita uma comparação dos dois algoritmos no tocante às características analisadas e mencionadas ao longo da dissertação. A característica é marcada com ✓ no algoritmo em que se destaca, mais favoravelmente:

Tabela 9 – Características dos algoritmos

Característica	*Hash* Perfeito	BSCL	Referência
Memória Alocada:	$12n$	$4n$ ✓	Equação 3.3 e Equação 3.4
Memória Utilizada:	$6,528n$	$2,528n$ ✓	Equação 3.9 e Equação 3.10
Eficiência de Memória:	$54,4\%$	63% ✓	Equação 3.16 e Equação 3.15
Memória Não Utilizada:	$5,472n$	$1,472n$ ✓	Equação 3.12 e Equação 3.13
Tempo de Processamento:	$3,57$	$1,00$ ✓	Tabela 8
Busca de Chaves por Faixas:	Não	Sim ✓	Página 51
Necessidade de Ordenamento:	Não ✓	Sim	Página 51
Número de Iterações (Pior Caso):	1,00 ✓	x_{max}	Página 50
Número de Iterações (Caso Médio):	1,00 ✓	1,37	Tabela 7
Chaves Uniformemente Distribuídas:	Não ✓	Sim	Página 51

Legenda: ✓- característica mais favorável.

Denota-se - baseado na Tabela 9 e dentro das características dos dados analisados - que em duas métricas, das três analisadas, que a BSCL foi superior ao *hash* perfeito: nos recursos de memória e tempo de processamento. No tocante a superioridade do número de iterações médias do *hash* perfeito em relação à BSCL (1,00 *versus* 1,37) não impactou no tempo de processamento da BSCL.

O *hash* perfeito, pelo fato de ser um algoritmo de dispersão, mantém um bom desempenho em processos não estocásticos. Na BSCL, sob esta circunstância, haveria muitas colisões - prejudicando a busca sequencial.

No próximo capítulo, será descrita uma aplicação real onde pode ser usada a BSCL, explorando suas vantagens e pontos positivos quando os dados são estocásticos e com chaves ordenadas.

4 Aplicação da BSCL

Este capítulo trata de uma aplicação prática para a BSCL no contexto da análise de qualidade dos serviços de telecomunicações. Entretanto, o algoritmo pode ser usado para outros fins em qualquer base de dados que tenha as características abaixo.

Segundo Hiller e Lieberman (1995) um conjunto de variáveis aleatórias é descrito como pertencente aos processos estocásticos, podendo se manifestar de forma temporal e/ou espacial. A título de exemplo, os processos que se enquadram nos fenômenos estocásticos são os das telecomunicações em chamadas telefônicas. Pois, as Centrais de Comutação e Controle[1] (CCC) alocam nos *Call Detail Record*[2] (CDR), de forma temporal e espacial, registros advindos de eventos que são independentes. As CCCs estão espalhadas por todas as regiões atendidas pelas ligações telefônicas[3], e cada uma faz o registro da ocorrência de cada evento independente nos CDRs de forma ordenada e crescente no tempo[4].

Um matemático dinamarquês - Erlang, A. K. - fez vários estudos probabilísticos baseados nas condições estocásticas das ligações telefônicas. Entre eles, destacam-se (ERLANG, 1909) e (ERLANG, 1948) onde propôs fórmulas para calcular a probabilidade de chamadas serem efetivadas quando um usuário “A”, locado em uma CCC, tentasse se conectar com um usuário “B”, locado em outra CCC, em um determinado período de análise. Seus trabalhos resultaram nas fórmulas de Erlang, que até hoje são usadas no dimensionamento das linhas troncos entre as CCCs de forma a aumentar a probabilidade de que as ligações sejam completadas.

Nos processos estocásticos, conforme o das telecomunicações, enquadra-se a distribuição probabilística de *Poisson* como meio de facilitar as análises de probabilidade dos eventos no processo.[5] No contexto de um universo de ligações telefônicas, a probabilidade de encontrar - numa única tentativa - uma determinada ligação[6] é baixa; e a quantidade de ligações[7] é alta. Sob estas condições, é utilizável *Poisson* nas análises de probabilidade[8].

Entretanto, não basta somente a análise de probabilidade em si. Muitas vezes, é necessário ter em mãos os dados que ensejaram os resultados para análises complementares.

No Brasil, a lei nº 9.472, de 16 de julho de 1997 dispõe no seu Art. 1º que compete

[1] Equipamentos de redes de telecomunicações que fazem as comutações e registros de ligações telefônicas.
[2] Registros das ligações telefônicas.
[3] Condição espacial.
[4] Condição temporal.
[5] Segundo Ross (2009), para a aplicabilidade de *Poisson*: se a probabilidade de cada ocorrência em um processo é baixa, e a quantidade das ocorrências for alta, a distribuição probabilística de *Poisson* se enquadra para efeito da análise de probabilidade.
[6] Representada pela constante $\underline{a}$ na equação de *Poisson* (Equação 1), que tem de ser baixa.
[7] Representada pela constante $\underline{r}$ na equação de *Poisson* (Equação 1), que tem de ser alta.
[8] A equação de *Poisson* foi usado po Erlang nos seus trabalhos.

à União, por intermédio do órgão regulador[9] e nos termos das políticas estabelecidas pelos Poderes Executivo e Legislativo, organizar a exploração dos serviços de telecomunicações. Dentro desta organização, existem duas resoluções emitidas pela Anatel que regulamentam a Gestão da Qualidade de dois serviços: a 574, de 28 de outubro de 2011, que aprova o Regulamento da Gestão da Qualidade do Serviço de Comunicação Multimídia[10] (RGQ-SCM) e a 575, de 28 de outubro de 2011, que aprova o Regulamento da Gestão da Qualidade do Serviço Móvel Pessoal[11] (RGQ-SMP).

Em ambas as resoluções, foram designadas Entidades Aferidoras da Qualidade (EAQs). EAQs são empresas contratadas pelas prestadoras dos serviços no intuito de processar os dados das redes e gerar os indicadores de qualidade dos serviços, a serem enviados à Anatel para auditoria e fiscalização. Tal designação se deve à necessidade de processamento de um grande volume de dados que chega a ordem de dezenas de TeraBytes diários de CDRs, oriundos das prestadoras dos serviços de telecomunicações.

Nos Art. 8 e 10 das Resoluções 574 e 575, respectivamente, regem que a Anatel poderá utilizar métodos alternativos aos previstos - nestes regulamentos - na coleta de dados dos indicadores para comprovação do cumprimento das metas estabelecidas. E é neste ponto que se fundamenta a proposta deste algoritmo de busca (BSCL), vindo ao encontro no que regem nestes artigos.

A BSCL foi projetada para trabalhar com alto desempenho e utilizando baixos recursos na busca de chaves específicas, ou faixas de chaves, em grandes matrizes de dados estáticos e estocásticos onde as colunas chaves, que referenciam os dados, são números ordenados e uniformemente distribuídos. Neste contexto, os CDRs - que são as fontes de dados mais confiáveis na geração destes indicadores - se enquadram dentro das bases que podem ser processadas pela BSCL. São estocásticos, ordenados no tempo e espaço, estáticos e com as colunas chaves - representadas por números - uniformemente distribuídas.

A título de maiores detalhes para a utilização da BSCL, usando os CDRs, temos o indicador SMP7 no RGQ-SMP [12]. No Art. 19 da Resolução 575 rege que a quantidade de chamadas interrompidas por queda de ligação na rede da prestadora, em cada Período de Maior Movimento (PMM) [13], no mês, deve ser inferior a 2% (dois por cento). A sua fórmula é dada por $SMP7 = \frac{A}{B} \times 100$. A Resolução 575 no Art. 19 especifica como deve ser o método de coleta para cada uma das variáveis da fórmula do SMP7:

[9] Agência Nacional de Telecomunicações (Anatel) – autarquia vinculada ao Ministério da Ciência, Tecnologia, Inovações e Comunicações.
[10] Meios de telecomunicações voltados à prestação dos serviços de internet.
[11] Rede de celulares.
[12] Indicador de taxa de queda de ligações telefônicas.
[13] Intervalos de tempo, ao longo do dia, em que ocorrem os maiores interesses no estabelecimento de chamadas por parte dos usuários. Período diurno das 10h00 às 13h00 (PMM1), período noturno das 18h00 às 21h00 (PMM2).

- A - a contagem das chamadas interrompidas por queda de ligação, na rede da prestadora, após o completamento, em cada PMM, no mês, por CCC existente [14];
- B - a contagem de todas as chamadas completadas, na rede da prestadora, em cada PMM, no mês, por CCC existente [15];
- os dados para obtenção deste indicador devem ser extraídos dos arquivos de tarifação ou de tráfego [16] das CCC, sendo a coleta efetuada de acordo com o Calendário Anual.

A Resolução continua especificando que os dados relativos a este indicador devem ser apresentados contendo informações, mês a mês, por CCC, para cada área definida pelo Código Nacional contida na Área de Prestação. Neste ponto, a própria resolução já coloca duas condições relativas aos processos estocásticos: condição espacial (... por CCC ... para cada área definida pelo Código Nacional contida na Área de Prestação); condição temporal (... mês a mês ...).

As CCCs, no registro de um CDR, cria um arquivo na sua memória secundária tendo um tempo de vida para inserção de dados das ligações que estão em curso, este tempo de vida é definido por cada prestadora. Entretanto, é pequeno, na ordem de minuto. Isto porque, se um arquivo for corrompido a prestadora não perde muitas tarifações das ligações que estavam no arquivo.

Este arquivo, através do seu próprio nome, já indica a qual CCC pertence, e por lógica a qual Código Nacional da área de prestação. Se se deseja a separação pelo Código Nacional na Área de Prestação, estes arquivos já estão ordenados na memória secundária. Basta que sejam separados pelos prefixos no nome do arquivo. Isto já torna fácil a separação espacial no processo estocástico.

Em prosseguimento, os arquivos de CDRs têm a data e horário em que foram criados - através dos seus metadados. E dentro de cada arquivo de CDR as CCCs colocam de forma ordenada, no tempo, hora/minuto/segundo relativos às ocorrências das ligações. Por esta condição de escrita, todos os registros dentro de cada arquivo de CDR já estão ordenados de forma crescente no tempo. Isto, também, ja torna fácil a separação temporal no processo estocástico.

Os arquivos de CDRs são lacrados pela estrutura de escrita *Abstract Syntax Notation One*[17] (ASN1) o que os torna estáticos, não sendo inserido ou retirado nenhum registro após a lacração. Cria-se, a partir deste instante, um novo arquivo de CDR para dar prosseguimento aos registros das ligações. Esta característica é fundamental para o uso da BSCL, a condição estática dos dados.

[14] Queda de ligação provocada por problemas na rede.
[15] Só se computa para a variável B as ligações que foram completadas, os dois usuários firmaram a comunicação.
[16] Estão nos arquivos de CDRs das CCCs.
[17] Padrão para a escrita de dados seriais em registros de telecomunicação.

Pelo exposto, as condições primordiais para o uso da BSCL, nas bases de dados dos CDRs, são satisfeitas: processo estocástico, ordenamentos espacial/temporal e dados estáticos. Bastando, para o cálculo do indicador de SMP7, o processamento dos dados nos registros, como se segue.

Há somente três condições de desligamento de uma ligação telefônica: pelo usuário U1 (que fez a ligação), pelo usuário U2 (que recebeu a ligação) e pela rede da prestadora - por problemas técnicos. As três informações de desligamentos estão contidas no campo *Disconnecting Party*[18] do CDR, em cada registro de ligação completada. Através deste campo, é possível contar a quantidade de ligações interrompidas pela rede da prestadora.

Sejam os registros de CDRs extraídos e agrupados no tempo e no espaço da análise - variável B da fórmula SMP7 -, e ordenados pelos códigos do campo *Disconnecting Party*. Para o algoritmo BSCL, basta encontrar o índice do registro inicial de cada uma das três faixas dos códigos de desligamento, e através de uma simples subtração de índices achar a quantidade de registros de ligações que foram interrompidas pela rede - variável A. Após, é só aplicar a fórmula do SMP7 com os valores de A e B encontrados.

A título de exemplo, seja a Tabela 10 representando uma base de dados de CDRs - ordenado pelo campo *Disconnecting Party* - baseado em ligações que foram completadas. Se a base de dados é de ligações completadas, a variável B da fórmula SMP7 já é automática, é a quantidade de registros da base, $B = 8$.

Com o BSCL em funcionamento, procura-se na base o início das faixas [19] cujos o campo *Disconnecting Party* sejam (00, 01 e 02), as únicas condições possíveis para este campo.

Tabela 10 – Contagem BSCL por faixa (SMP7) - *Disconnecting Party*

Registro	*Disconnecting Party*
1	00
2	00
3	00
4	01
5	01
6	01
7	02
8	02

Como o campo *Disconnecting Party* está ordenado, temos que o início da faixa de chave para o código "00" (de desligamento pela rede) está no registro "1", e o da chave para o próximo código "01" está no registro "4". Então, para calcularmos a variável A da

[18] Para a tecnologia Ericsson, os códigos do *Disconnecting Party* são: 00 (desligamento pela rede); 01 (desligamento pelo usuário U1); 02 (desligamento pelo usuário U2)..

[19] Representadas em cinza na base de dados.

fórmula SMP7 basta fazer a subtração dos registros de início de chaves, $A = 4 - 1 = 3$. Com isto, num total de apenas $2 \times 1,37 = 2,74$ iterações numa base, **independentemente do número de registros**[20], teremos todas as informações para contagem de desligamentos pela rede, ou por qualquer um dos usuários. Neste modelo, o indicador SMP7 é calculado em um tempo muito pequeno.

O procedimento de contagem para os cálculos é simples, o problema é a quantidade de registros que pode chegar a centenas de milhões de chamadas. As EAQs, pela sua grande capacidade computacional, usam Sistema de Gerenciamento de Banco de Dados (SGBD) - por contagem - para a aferição dos indicadores de qualidade dos RGQs do SMP e SCM. Entretanto, justamente pelo grande volume de dados, estes processos de aferição podem demorar nesta forma de processamento na SQL.

A maioria dos indicadores de qualidade contida nos RGQs constitui, basicamente, de contagens. Neste ambiente, a BSCL não necessita de grande capacidade computacional. Com 1,37 iterações à base de dados - que pode estar em memória secundária - chegaria ao início dos índices de cada uma das faixas de dados, que satisfaçam determinada condição. Dai, os indicadores de qualidade seriam gerados com simples operações matemáticas envolvendo os índices das faixas[21].

Este algoritmo pode propiciar, à Anatel, subsídios para colocar em prática um novo modelo de processamento computacional de forma que, as atribuições designadas pela lei nº 9.472, de 16 de julho de 1997 sejam alcançadas com maior eficiência e eficácia por esta Agência Reguladora.

> Art. 38. A atividade da Agência será juridicamente condicionada pelos princípios da legalidade, **celeridade**[22], finalidade, razoabilidade, proporcionalidade, impessoalidade, igualdade, devido processo legal, publicidade e moralidade.
> . (Lei nº 9.472, de 16 de julho de 1997.)

[20] Vide subseção 3.3.2.
[21] Este procedimento de cálculo dos indicadores de qualidade dos RGQs, através da BSCL, só é possível porque as bases de dados são ordenadas.
[22] Grifo nosso.

Conclusão

Em resposta à hipótese de pesquisa levantada na Introdução, em que se questiona se é possível uma busca sequencial - baseada nas faixas de colisões sob a distribuição probabilística de *Poisson* - ser mais eficiente do que uma tabela *hash* perfeito nos recursos do sistema: conclui-se, através das deduções matemáticas na análise de memória necessária para o funcionamento dos algoritmos - e validada pelos resultados experimentais -, que sim (na ordem de 3 vezes superior).

No tocante à hipótese da BSCL ser mais eficiente no tempo de processamento, se comparado ao *hash* perfeito: demonstrou-se que o resultado foi significativo (na ordem de 3,5 vezes superior). Entretanto, cabe aqui uma ressalva: o algoritmo BSCL foi projetado para trabalhar com colunas de chaves uniformemente distribuídas e ordenadas, isto implica - segundo a distribuição probabilística de *Poisson* - que as faixas de colisões regidas por uma equação de 1º grau terão poucos elementos em colisão. Esta característica propiciou, usando a busca sequencial, uma eficiência bem superior para a BSCL.

Se as colunas de chaves não forem uniformemente distribuídas, haverá faixas com muitos elementos em colisão. Com isto, degradará este desempenho de processamento - pois a busca sequencial seria de $\mathcal{O}(n)$ em faixas de colisões onde n seria muito grande.

Diante do exposto, o algoritmo apresentado - BSCL - demonstrou, para o conjunto das métricas analisadas, ser superior ao de referência - Busca *Hash* Perfeito -, mas somente validado dentro das características dos dados analisados. Disto, tem-se que: se as colunas de chaves forem uniformemente distribuídas e ordenadas a BSCL será superior ao *hash* perfeito em recursos de memória e tempo de processamento.

Com base nos trabalhos estudados, e apresentados na Revisão da Literatura, verificou-se que as variantes do algoritmo de tabela *hash* no intuito de obterem maior desempenho - como no *hash* perfeito - se baseiam em algoritmos mais complexos do que o original. No mesmo caminho, as variantes da busca binária pura - e mesmo assim continuando com o número de iterações máximo na ordem de $\mathcal{O}(\log n)$. No oposto deste paradigma, a BSCL teve como foco principal trabalhar com rotinas de menor complexidade visando um ponto ótimo de custo *versus* benefício.[23]

Por se tratar de um algoritmo voltado a processos estocásticos, a BSCL pode vir a ser utilizado em diversas estruturas de índices - que referenciam dados estocásticos - como motor de busca[24]. Foi possível verificar bons resultados, mas melhorias também podem vir

[23] Nota-se, baseado nos pseudo-códigos apresentados na dissertação, que as rotinas são bem simples - visando a eficiência de processamento.

[24] Com excelente desempenho de processamento, e recursos de memória, se as chaves forem ordenadas e uniformemente distribuídas.

a ser realizadas de forma a aperfeiçoar as rotinas da BSCL visando diminuir as colisões para colunas de chaves desuniformemente distribuídas .

Dentre as contribuições oferecidas por este trabalho, destacam-se:

- A validação de que rotinas simples podem obter bons resultados de custo *versus* benefício.
- Um algoritmo de busca - em bases estáticas de processos estocásticos - com bom desempenho computacional usando poucos recursos de memória.

Os trabalhos futuros, voltados à eficiência do algoritmo e maiores funcionalidades, podem ser desenvolvidos a partir desta dissertação. Entre eles, destacam-se:

- Estudos mais detalhados quanto ao ganho de eficiência devido ao armazenamento em *cache* dos dados dos vetores.
- Análise de desempenho e robustez do algoritmo quando trabalhar com dados armazenados em memória secundária.
- Desenvolvimento de índices que se adaptem às características do algoritmo.
- Pesquisas voltadas ao desempenho de processamento e ao número de iterações quando os dados de colisões forem guardados em matrizes Δ com o número de registros maior do que as matrizes de dados.
- Adaptar o pré-processamento para fazer o ordenamento dos elementos do campo chave, na matriz de dados, durante a criação da matriz Δ.

No mais, o caminho seguido de projetar rotinas mais simples, mas não simplórios, nos projetos dos algoritmos - como a busca sequencial adotada no BSCL - mostrou ser válida quando se procura obter resultados mais expressivos no processamento.

“Make everything as simple as possible,
but not simpler”
(Einstein, Albert)

Referências

BAER, J.-L.; CHEN, T.-F. An effective on-chip preloading scheme to reduce data access penalty. In: IEEE. *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*. [S.l.], 1991. p. 176–186. Citado na página 15.

BENTLEY, J. L.; YAO, A. C.-C. An almost optimal algorithm for unbounded searching. *Information processing letters*, Elsevier, v. 5, n. 3, p. 82–87, 1976. Citado na página 20.

CELIS, P.; LARSON, P.-A.; MUNRO, J. I. Robin hood hashing. In: IEEE. *Foundations of Computer Science, 1985., 26th Annual Symposium on*. [S.l.], 1985. p. 281–288. Citado na página 23.

CHAZELLE, B.; GUIBAS, L. J. Fractional cascading: I. a data structuring technique. *Algorithmica*, Springer, v. 1, n. 1, p. 133–162, 1986. Citado na página 20.

CHAZELLE, B.; LIU, D. Lower bounds for intersection searching and fractional cascading in higher dimension. In: ACM. *Proceedings of the thirty-third annual ACM symposium on Theory of computing*. [S.l.], 2001. p. 322–329. Citado na página 20.

CORMEN, T. H. et al. *Introduction to algorithms*. [S.l.]: MIT press Cambridge, 2009. Citado 7 vezes nas páginas 16, 21, 24, 42, 46, 47 e 50.

ERLANG, A. K. The theory of probabilities and telephone conversations. *Nyt Tidsskrift for Matematik B*, v. 20, n. 6, p. 87–98, 1909. Citado na página 53.

ERLANG, A. K. Solution of some problems in the theory of probabilities of significance in automatic telephone exchanges. *Transactions of the Danish Academy of Technical Sciences*, v. 2, p. 138–155, 1948. Citado na página 53.

FLORES, I.; MADPIS, G. Average binary search length for dense ordered lists. *Communications of the ACM*, ACM, v. 14, n. 9, p. 602–603, 1971. Citado na página 19.

FREDMAN, M. L.; KOMLÓS, J.; SZEMERÉDI, E. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, ACM, v. 31, n. 3, p. 538–544, 1984. Citado na página 15.

GEEM, Z. W.; KIM, J. H.; LOGANATHAN, G. V. A new heuristic optimization algorithm: harmony search. *simulation*, Sage Publications Sage CA: Thousand Oaks, CA, v. 76, n. 2, p. 60–68, 2001. Citado na página 15.

HASSIN, R. On maximizing functions by fibonacci search. *Fibonacci Quart*, v. 19, p. 347–351, 1981. Citado na página 20.

HILLER, F.; LIEBERMAN, G. Introduction to operations research sixth edition. *to*, v. 274, p. 81, 1995. Citado 2 vezes nas páginas 26 e 53.

HOROWITZ, E.; SAHNI, S. *Fundamentals of computer algorithms*. [S.l.]: Computer Science Press, 1978. Citado na página 15.

KNUTH, D. E. *The art of computer programming: sorting and searching*. [S.l.]: Pearson Education, 1998. Citado 4 vezes nas páginas 16, 18, 21 e 36.

LEHMER, D. H. Teaching combinatorial tricks to a computer. In: *Proc. Sympos. Appl. Math. Combinatorial Analysis*. [S.l.: s.n.], 1960. v. 10, p. 179–193. Citado na página 18.

MOFFAT, A.; TURPIN, A. *Compression and coding algorithms*. [S.l.]: Springer Science & Business Media, 2002. Citado na página 20.

PAGH, R.; RODLER, F. F. Cuckoo hashing. In: SPRINGER. *European Symposium on Algorithms*. [S.l.], 2001. p. 121–133. Citado na página 23.

PERL, Y.; ITAI, A.; AVNI, H. Interpolation search—a log log n search. *Communications of the ACM*, ACM, v. 21, n. 7, p. 550–553, 1978. Citado na página 19.

PRENEEL, B. Cryptographic hash functions. *Transactions on Emerging Telecommunications Technologies*, Wiley Online Library, v. 5, n. 4, p. 431–448, 1994. Citado na página 23.

RICHA, A. W.; MITZENMACHER, M.; SITARAMAN, R. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, v. 9, p. 255–304, 2001. Citado na página 23.

ROSS, S. *Probabilidade: um curso moderno com aplicações*. [S.l.]: Bookman Editora, 2009. Citado 4 vezes nas páginas 16, 34, 40 e 53.

SEDGEWICK, R.; WAYNE, K. *Algorithms*. [S.l.]: Addison-Wesley Professional, 2011. Citado na página 19.

SPIVAK, M. *Calculus. Houston: Publish or Perish*. [S.l.]: Inc, 1994. Citado na página 62.

TORGO, L. Functional models for regression tree leaves. In: CITESEER. *ICML*. [S.l.], 1997. v. 97, p. 385–393. Citado na página 18.

WADA, T.; RAJAN, S.; PRZYBYLSKI, S. A. An analytical access time model for on-chip cache memories. *IEEE Journal of solid-state circuits*, IEEE, v. 27, n. 8, p. 1147–1156, 1992. Citado na página 15.

WANG, J. Computer network routing configuration based on intelligent algorithm. In: IEEE. *Future Generation Communication Technologies (FGCT), 2017 Sixth International Conference on*. [S.l.], 2017. p. 1–4. Citado na página 16.

WANG, X. et al. Collisions for hash functions md4, md5, haval-128 and ripemd. *IACR Cryptology ePrint Archive*, v. 2004, p. 199, 2004. Citado na página 23.

WILTON, S. J.; JOUPPI, N. P. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, IEEE, v. 31, n. 5, p. 677–688, 1996. Citado na página 15.

APÊNDICE A – Fórmulas

Este apêndice tem a função de deduzir fórmulas, específicas, usadas na seção 3.3.

A.1 Fórmula F.1

Seja a fórmula[1]:

$$\lim_{n\to\infty} \sum_{x=0}^{n} \frac{1}{x!}$$

Segundo (SPIVAK, 1994), o teorema de Taylor para uma função $f(x)$ em que existem n-derivadas em x_0, com $n \to \infty$, pode ser escrito como:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)(x - x_0)^n}{n!}$$

Onde $f^{(n)}(x_0)$ é a n-ésima derivada calculada em $x = x_0$.

No caso específico da função $f(x) = e^x$, $f^{(1)}(x_0) = f^{(2)}(x_0) = ... = f^{(n)}(x_0) = e^{x_0}$, pois a derivada da exponencial é ela mesma (SPIVAK, 1994).

Assim, fazendo $x_0 = 0$ chega-se em:

$$f(x) = \sum_{n=0}^{\infty} \frac{e^0(x-0)^n}{n!} = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Sendo $f(x) = e^x$ tem-se: $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$

Para $x = 1$, tem-se: $\sum_{n=0}^{\infty} \frac{1^n}{n!} = e^1$

Substituindo o n do fatorial e da somatória por x, com $n \to \infty$, tem-se:

$$\lim_{n\to\infty} \sum_{x=0}^{n} \frac{1}{x!} = e \qquad \text{(A.1)}$$

A.2 Fórmula F.2

Seja a fórmula[2]:

$$\lim_{n\to\infty} \sum_{x=1}^{n} \frac{x}{x!}$$

Sendo $x! = x(x-1)!$, tem-se:

$$\lim_{n\to\infty} \sum_{x=1}^{n} \frac{1}{(x-1)!}$$

[1] Usada na dedução da Equação 3.7.
[2] Usada na dedução da Equação 3.8 e Equação 3.20.

Definindo $z = x - 1$ e substituindo na fórmula acima:

$$\lim_{n\to\infty} \sum_{z=0}^{n-1} \frac{1}{z!}$$

No limite $n \to \infty$ a fórmula pode ser escrita como:

$$\lim_{n\to\infty} \sum_{z=0}^{n} \frac{1}{z!}$$

Como demonstrado na Equação A.1 tem-se que:

$$\lim_{n\to\infty} \sum_{z=0}^{n} \frac{1}{z!} = \lim_{n\to\infty} \sum_{x=0}^{n} \frac{1}{x!} = e$$

Portanto:

$$\lim_{n\to\infty} \sum_{x=1}^{n} \frac{x}{x!} = e \tag{A.2}$$

A.3 Fórmula F.3

Seja a fórmula[3]:

$$\lim_{n\to\infty} \sum_{x=0}^{n} \frac{x^2}{x!}$$

A fórmula acima pode ser escrita como:

$$\lim_{n\to\infty} \sum_{x=1}^{n} \frac{x^2}{x!}$$

O termo da somatória para $x = 0$ não influencia no resultado.

Sendo $x! = x(x-1)!$ e substituindo na fórmula acima, tem-se:

$$\lim_{n\to\infty} \sum_{x=1}^{n} \frac{x}{(x-1)!}$$

Esta pode ser escrita como:

$$\lim_{n\to\infty} \sum_{x=1}^{n} \frac{[(x-1)+1]}{(x-1)!}$$

Baseado nas propriedades de somatórias tem-se:

$$\lim_{n\to\infty} \sum_{x=0}^{n} \frac{x^2}{x!} = \lim_{n\to\infty} \sum_{x=1}^{n} \frac{(x-1)}{(x-1)!} + \lim_{n\to\infty} \sum_{x=1}^{n} \frac{1}{(x-1)!} \tag{A.3}$$

Sejam:

$$D = \lim_{n\to\infty} \sum_{x=1}^{n} \frac{(x-1)}{(x-1)!} \qquad E = \lim_{n\to\infty} \sum_{x=1}^{n} \frac{1}{(x-1)!}$$

[3] Usada na dedução da Equação 3.2.

Cálculo de D :

$$\lim_{n\to\infty}\sum_{x=1}^{n}\frac{(x-1)}{(x-1)!}=\lim_{n\to\infty}\sum_{x=2}^{n}\frac{(x-1)}{(x-1)!}$$

Pois o termo para $x=1$ não interfere na parcela da somatória - o numerador é 0.

Definindo $z=x-2$ e utilizando que:

$$\frac{(x-1)}{(x-1)!}=\frac{(x-1)}{(x-1)(x-2)!}=\frac{1}{(x-2)!}$$

Tem-se:

$$\lim_{n\to\infty}\sum_{z=0}^{n-2}\frac{1}{z!}$$

No limite $n\to\infty$ a fórmula pode ser escrita como:

$$\lim_{n\to\infty}\sum_{z=0}^{n}\frac{1}{z!}$$

Como na Equação A.1:

$$D=e \tag{A.4}$$

Cálculo de E :

$$\lim_{n\to\infty}\sum_{x=1}^{n}\frac{1}{(x-1)!}$$

Seja $z=x-1$ e substituindo na fórmula acima:

$$\lim_{n\to\infty}\sum_{z=0}^{n-1}\frac{1}{z!}$$

No limite $n\to\infty$ a fórmula pode ser escrita como:

$$\lim_{n\to\infty}\sum_{z=0}^{n}\frac{1}{z!}$$

Como na Equação A.1:

$$\lim_{n\to\infty}\sum_{z=0}^{n}\frac{1}{z!}=e$$

Logo:

$$E=e \tag{A.5}$$

Substituindo a Equação A.4 e a Equação A.5 na Equação A.3 tem-se:

$$\lim_{n\to\infty}\sum_{x=0}^{n}\frac{x^2}{x!}=2e \tag{A.6}$$