Universidade Federal
de São João del-Rei

TRABALHO PRÁTICO 3

Lucas Gonçalves Nojiri
Arthur Antunes Santos Silva

Este trabalho prático tem por objetivo exercitar conceitos e práticas para encontrar casamentos de caracteres utilizando 3 ou mais estrategias. Baseado no código em C para a disciplina de AEDS3 do curso de Ciência da Computação da Universidade Federal de São João del Rei.

São João del Rei

Junho de 2023

Sumário

1 Introdução

Este documento descreve a implementação de algoritmos de casamento de padrões para resolver o problema de identificar se uma sequência de poder está presente em uma pedra mágica. O problema surge em um antigo reino chamado Xulambis, onde a magia era uma realidade e cada habitante recebia uma pedra mágica composta por uma sequência de símbolos coloridos ao atingir a maioridade.

Cada sequência de símbolos na pedra representava um poder único. Por exemplo, uma sequência de símbolos azul, azul, branco, branco, verde, vermelho poderia conceder ao usuário o poder da velocidade. No entanto, nem todas as pedras possuíam poderes, algumas delas eram apenas sequências aleatórias de símbolos.

Recentemente, um grupo de pesquisadores explorou o templo de Xulambis e encontrou um antigo dicionário que relacionava cada sequência de símbolos a um poder específico. Agora, os pesquisadores estão determinados a separar as pedras que realmente têm poderes das que não têm, para comprovar a existência da magia para o mundo.

O desafio é que as pedras podem ter milhares de símbolos, tornando o processo de verificação manual extremamente demorado e propenso a erros humanos. Portanto, o objetivo deste trabalho é implementar e comparar diferentes algoritmos de casamento de padrões que permitam determinar se uma determinada sequência de poder está presente em uma pedra mágica.

Serão propostos, implementados e avaliados pelo menos três algoritmos para resolver esse problema. A análise de complexidade será feita sobre as estratégias implementadas e comparada com os tempos reais de execução para verificar a eficiência de cada algoritmo. A comparação entre a análise de complexidade e os tempos de execução ajudará a determinar qual dos métodos implementados obteve o melhor desempenho na resolução do problema proposto.

1.1 Requisitos

Implementar pelo menos três algoritmos para resolver o problema de encontrar sequências de habilidades em pedras mágicas. Comparar e avaliar o desempenho dos algoritmos implementados, levando em consideração a análise de complexidade feita sobre eles e os tempos reais de execução. O código deve ser escrito na linguagem C. Utilizar estruturas de dados alocadas dinamicamente e modularizar o código usando arquivos .c e .h. Utilizar o utilitário Make para compilar o programa.

O arquivo executável deve ser chamado de "tp3"e deve receber dois parâmetros: o nome do arquivo de entrada e um inteiro representando a estratégia implementada. Ler a entrada a partir de um arquivo de texto, contendo vários casos de teste. Escrever a saída em um arquivo de texto, utilizando o nome do arquivo de entrada com a extensão ".out". Imprimir apenas os tempos de usuário e os tempos de sistema na tela para comparação.

2 Mapeamento do código

2.0.1 estrategia1

A estrategia 1 implementa a busca de padrões utilizando o algoritmo de Força bruta. Ele lê um arquivo de entrada contendo casos de teste e, para cada caso de teste, realiza a busca do padrão em uma determinada sequência de caracteres.

1. **int Forca_bruta(const char *string1, const char *string2)**

   A função Forca_bruta é esta sendo responsável por buscar casamentos entre duas strings utilizando a estratégia de força bruta. Ela recebe como entrada duas strings chamadas de strings 1 e 2. A função começa calculando o tamanho das duas strings usando a função strlen. Apos isso, ela realiza duas iterações para verificar se há casamento nas duas direções existentes, podendo ser de ordem direta e/ou ordem reversa.

   Na iteração de ordem direta, a função utiliza dois índices, i para percorrer a string2 e j para percorrer a string1. Ela compara os caracteres das duas strings nas posições correspondentes e avança os índices. Se todos os caracteres das duas strings casarem, ou seja, j chegar ao final da string1, a função retorna a posição do casamento, que é representada por i.

   Na iteração de ordem reversa, a função utiliza os mesmos índices, mas começa a partir do último caractere da string2. Ela percorre a string2 em ordem reversa, comparando seus caracteres com os da string1. A posição do casamento é calculada de forma diferente para garantir que a posição retorne a última posição correta ou a posição na ordem reversa.

   Se a função não encontrar nenhum casamento em ambas as iterações, ela retorna -1 indicando que nenhum casamento foi encontrado.

2. **void Casamento_forca_bruta(No *cabeca)**

   A função Casamento_forca_bruta esta sendo responsável por percorrer a lista ligada de pares de strings e chamar a função Forca_bruta para cada par. Ela itera sobre os nós da lista, começando pelo nó inicial (cabeca) e avançando para o próximo nó até retornar ao nó inicial. Para cada par de strings, a função chama Forca_bruta passando as strings como argumentos e armazena o resultado na variável posicao_casamento. Em seguida, ela verifica se houve casamento ou não.

   Se posicao_casamento for diferente de -1, significa que houve um casamento entre as strings. A função imprime a posição do casamento acrescida de 1 e os caracteres correspondentes na string2, utilizando o valor da posição do casamento e o tamanho da string1. Se posicao_casamento for igual a -1, significa que não houve casamento, e a função imprime uma mensagem informando isso.

   Após imprimir os resultados para um par de strings, a função avança para o próximo nó da lista e repete o processo até percorrer todos os nós.

2.0.2 estrategia2

A estrategia 2 implementa a busca de padrões utilizando o algoritmo de casamento de padrões KMP (Knuth-Morris-Pratt). Ele lê um arquivo de entrada contendo casos de teste e, para cada caso de teste, realiza a busca do padrão em uma determinada sequência de caracteres.

1. **void Casamento_KMP(No *cabeca, FILE *output))**

   Essa função realiza o processo de casamento de padrões usando o algoritmo KMP para uma lista circular de nós (No). Ela recebe como parâmetros o ponteiro para a cabeça da lista (cabeca) e o ponteiro para o arquivo de saída (output). A função itera sobre os nós da lista até chegar novamente à cabeça. Para cada nó, a função calcula o tamanho das strings (string1 e string2) e cria uma sequência circular duplicando a sequência original (txtCircular). Em seguida, chama as funções KMP_Busca e KMP_Reverso para buscar o padrão na sequência circular de caracteres e obter as posições de casamento. Após o casamento, a função escreve "S"seguido pela posição onde o padrão começa (levando em conta a duplicação circular) se ele for encontrado. Caso contrário, escreve "N".A função libera a memória alocada para a sequência circular de caracteres e passa para o próximo nó da lista. O processo é repetido até que o nó atual seja novamente a cabeça da lista.

2. **int KMP_Reverso(char *padrao, char *txt)**

   Essa função realiza a busca reversa do padrão (padrao) na sequência de caracteres (txt). Ela retorna a posição de casamento (se encontrado) ou -1 (se não encontrado). A função começa percorrendo a sequência de caracteres de txt a partir do último caractere (len2 - 1) até o primeiro caractere. Para cada posição, ela compara os caracteres de txt com os caracteres do padrão (padrao) em ordem reversa. Se houver um casamento completo, ou seja, todos os caracteres do padrão correspondem aos caracteres de txt em ordem reversa, a função retorna a posição correta (levando em conta a reversão). A função utiliza a operação de módulo % para fazer uma indexação circular, ou seja, quando chega ao início da sequência de caracteres, ela volta ao final.Caso nenhum casamento seja encontrado, a função retorna -1.

3. **int KMP_Busca(char *padrao, char *txt)**

   Essa função implementa o algoritmo KMP para realizar a busca do padrão (padrao) na sequência de caracteres (txt).Ela retorna a posição de casamento (se encontrado) ou -1 (se não encontrado).A função começa obtendo o tamanho do padrão (M) e da sequência de caracteres (N). Em seguida, aloca memória para o array lps (longest proper prefix which is also suffix) com tamanho M.A função chama a função Contagem_Caracteres para calcular os valores do array lps. A busca é feita percorrendo os caracteres de txt e padrao ao mesmo tempo, comparando os caracteres correspondentes. Se houver um casamento parcial, a função atualiza as posições de txt e padrao para continuar a comparação a partir do próximo caractere possível. Se houver um casamento completo (ou seja, j atingir o valor de M), a função retorna a posição correta na sequência de caracteres. Antes de retornar, a função libera a memória alocada para o array lps.

4. **void Contagem_Caracteres(char *padrao, int M, int *lps)**

   Essa função é responsável pelo pré-processamento do padrão (padrão) para construir o array lps que foi baseado no algoritmo (longest proper prefix which is also suffix). Ela recebe como parâmetros o padrão (padrao), o tamanho do padrão (M) e o array lps. O objetivo do pré-processamento é calcular para cada posição i do padrão o comprimento do maior prefixo próprio que também é sufixo em padrão[0...i]. O array lps é preenchido com os valores calculados.A função percorre o padrão a partir da segunda posição até a última, comparando cada caractere com o caractere do prefixo anterior e atualizando o valor correspondente no array lps. Se o caractere atual não corresponder ao caractere do prefixo anterior, a função ajusta a posição do prefixo anterior para recomeçar a comparação com o novo caractere atual. Caso o comprimento do prefixo seja 0, significa que não há nenhum prefixo próprio que também é sufixo, então o valor correspondente no array lps é definido como 0.

2.0.3 estrategia3

A estrategia 3 implementa a busca de padrões utilizando o algoritmo de casamento de padrões ShiftAnd. Ele lê um arquivo de entrada contendo casos de teste e, para cada caso de teste, realiza a busca do padrão em uma determinada sequência de caracteres.

1. **int ShiftAnd(char *txt, char *padrao)**

   Esta função implementa a estratégia Shift-And para encontrar casamentos exatos do padrão em um texto. Ela recebe como parâmetros uma string txt representando o texto e uma string padrao representando o padrão a ser buscado. A função retorna a posição do casamento do padrão no texto ou -1 caso não haja casamento. Esta função utiliza manipulação de bits para realizar a busca de forma eficiente. O algoritmo Shift-And faz uso de uma tabela de caracteres, chamada de "máscara", que é inicializada com todos os elementos definidos como 0. Percorre cada caractere do padrão e define a posição correspondente na máscara como 1, marcando a presença do caractere no padrão. Se houver um casamento, a função calcula a posição do casamento e retorna esse valor. Caso contrário, retorna -1.

2. **int ShiftAnd_Reverso(char *txt, char *padrao)**

   Esta função é uma variação da função ShiftAnd, porém busca casamentos reversos do padrão. Ela recebe os mesmos parâmetros que a função ShiftAnd sendo eles uma string txt representando o texto e uma string padrão representando o padrão a ser buscado. A função retorna a posição do casamento reverso do padrão no texto ou -1 caso não haja casamento.

3. **void Casamento_ShiftAnd(No *cabeca, FILE *output)**

   Esta função recebe um ponteiro para a cabeça de uma lista encadeada de nós e um ponteiro para um arquivo de saída. Implementando a função principal do programa, que realiza a busca de casamentos exatos do padrão em cada nó da lista encadeada usando a estratégia Shift-And. Essa função é utilizada em conjunto com as outras duas funções para buscar casamentos em uma estrutura de dados específica e registrar os resultados em um arquivo.

3 Complexidade

Em todas as estratégias, é necessário percorrer todos os nós da lista circular, o que leva a uma complexidade adicional de O(T), onde T é o número de nós na lista.

Portanto, a complexidade geral do programa depende da estratégia escolhida, podendo variar entre O(N*M * T), O((N+M) * T) e O((N+M) * T), onde N é o tamanho da segunda string, M é o tamanho da primeira string e T é o número de nós na lista circular.

3.0.1 estrategia1

1. **int Forca_bruta(const char *string1, const char *string2)**

   A função executa duas iterações, uma para a ordem direta e outra para a ordem reversa das strings. Para cada iteração, são realizadas comparações de caracteres até que ocorra um casamento completo ou que a iteração seja concluída.

   A complexidade dessa função é O(len1 * len2), onde len1 é o comprimento da string1 e len2 é o comprimento da string2.

2. **void Casamento_forca_bruta(No *cabeca)**

   A função itera sobre a lista ligada aos pares de strings. Para cada par de strings, chama a função Forca_bruta, que possui a complexidade O(len1 * len2) também. Desta forma, a complexidade da função Casamento_forca_bruta depende do número de pares de strings e do tamanho das strings em cada par. Se o número de pares de strings for N e o comprimento médio das strings for M, a complexidade de tempo será $O(N * M^2)$.

3.0.2 estrategia2

1. **void Casamento_KMP(No *cabeca, FILE *output))**

   A função itera sobre os nós da lista até chegar novamente à cabeça. O número de iterações é igual ao número de nós na lista, que denotaremos por n. Para cada nó, a função realiza operações de cópia de strings, cuja complexidade é proporcional ao tamanho das strings (len1 e len2). Em seguida, chama as funções KMP_Busca e KMP_Reverso, cujas complexidades foram calculadas anteriormente como O(len2 * len1) e O(M + N), respectivamente. Portanto, a complexidade total da função é O(n * (len1 + len2 + len2 * len1 + M + N)).

2. **int KMP_Reverso(char *padrao, char *txt)**

   A função utiliza um loop for para percorrer os caracteres de txt, resultando em uma complexidade de O(len2), onde len2 é o tamanho da sequência txt. Dentro do loop, há um loop while que compara os caracteres de txt com os caracteres de padrao. O número máximo de iterações desse loop é o tamanho do padrão len1. Portanto, a complexidade total da função é O(len2 * len1).

3. **int KMP_Busca(char *padrao, char *txt)**

   A função começa calculando o tamanho do padrão (M) e o tamanho da sequência txt (N), o que tem complexidade O(1). Em seguida, é alocado espaço para o array

lps de tamanho M, o que também tem complexidade O(1). A função chama a função Contagem_Caracteres, cuja complexidade depende do tamanho do padrão M. O loop while percorre os caracteres de txt e padrao, e o número máximo de iterações desse loop é N. No interior do loop, há uma condição if que pode levar a uma atualização de j e i ou a uma atualização de j com base no valor de lps[j - 1]. Portanto, a complexidade do loop while é O(N). Considerando que a função Contagem_Caracteres tem complexidade O(M), a complexidade total da função é O(M + N).

4. **void Contagem_Caracteres(char *padrao, int M, int *lps)**

   A função contém um loop while que percorre o padrão a partir da segunda posição até a última. O número máximo de iterações desse loop é M - 1, onde M é o tamanho do padrão. Dentro do loop, há uma condição if que pode levar a uma atualização de len ou a uma atualização de len com base no valor de lps[len - 1]. Portanto, a complexidade do loop while é O(M). No geral, a complexidade da função é O(M).

3.0.3 estrategia3

A estrategia 3 possui complexidade determinada pelas funções ShiftAnd() e ShiftAnd_Reverso(), que são O(n + m) e implementa a busca de padrões utilizando o algoritmo de casamento de padrões ShiftAnd. Ele lê um arquivo de entrada contendo casos de teste e, para cada caso de teste, realiza a busca do padrão em uma determinada sequência de caracteres.

1. **int ShiftAnd(char *txt, char *padrao)**

   A função ShiftAnd() utiliza um algoritmo de busca chamado Shift-And, que possui complexidade O(n + m), onde n é o tamanho do texto e m é o tamanho do padrão. O txt é representado pela variável txt_circ e o padrão é representado pela variável padrão. Portanto, a complexidade de ShiftAnd() é O((2 * txt_tam) + padrão_tam), onde txt_tam é o tamanho do texto e padrao_tam é o tamanho do padrão.

2. **int ShiftAnd_Reverso(char *txt, char *padrao)**

   A função ShiftAnd_Reverso() chama a função ShiftAnd() passando o texto invertido como argumento. Portanto, a complexidade de ShiftAnd_Reverso() é a mesma de ShiftAnd()

3. **void Casamento_ShiftAnd(No *cabeca, FILE *output)**

   A função Casamento_ShiftAnd() percorre a lista circular de nós e chama as funções ShiftAnd() e ShiftAnd_Reverso() para cada par de strings. Portanto, a complexidade do Casamento_ShiftAnd() depende do número de nós na lista e do tamanho das strings.

4 Testes

Os testes realizados usaram os parâmetros de entrada dado pelo professor, sendo eles:
4
ava av
patapon npatapatapatapo
isitfriday ohnoitisnt
haskell lleksah

Além de todas as execuções para a análise foram usadas o comando valgrind para verificar vazamento de memória.

4.1 Performance e Resultados

Foram testados 3 algoritmos, sendo eles um Força Bruta, um KMP e um Shift-And. Durante a implementação do programa também foram testados outros métodos como o Casamento Exato e Boyer-Moore porém não conseguimos resultados corretos.

Para realizar os testes adotamos a seguinte rotina: os códigos foram executados 5 vezes para cada estratégia e em seguida foram comparados o tempo com o uso de memória durante a execução.

A seguir serão mostrados uma imagem do terminal durante a execução do algoritmo e um gráfico de comparação entre tempo de execução e uso de memória em cada uma das estratégias.

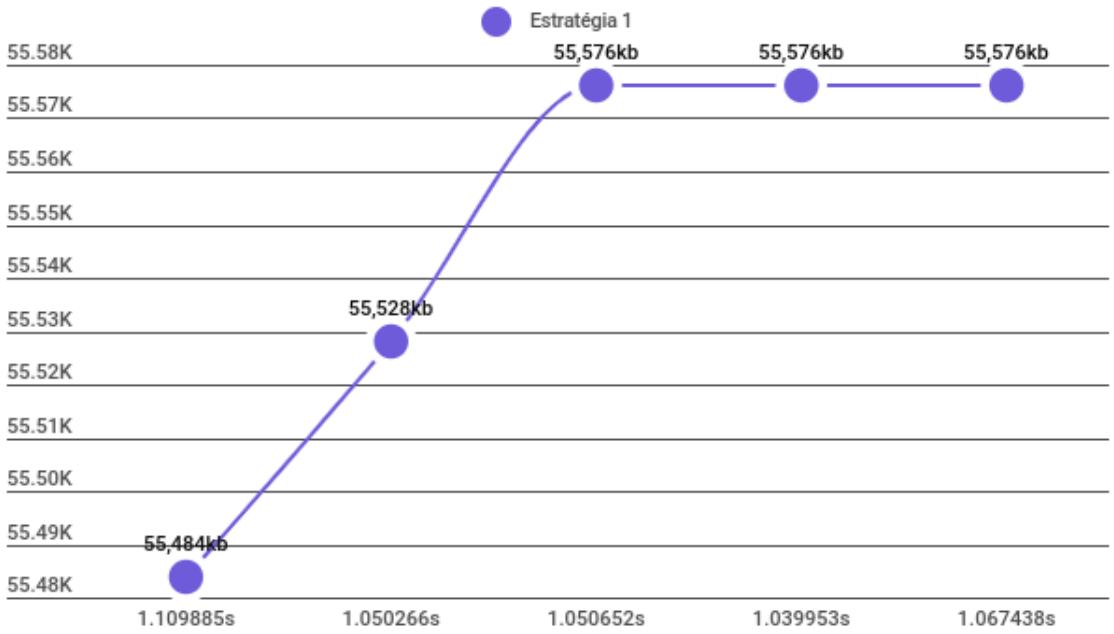


Figura 1: Estrategia 1

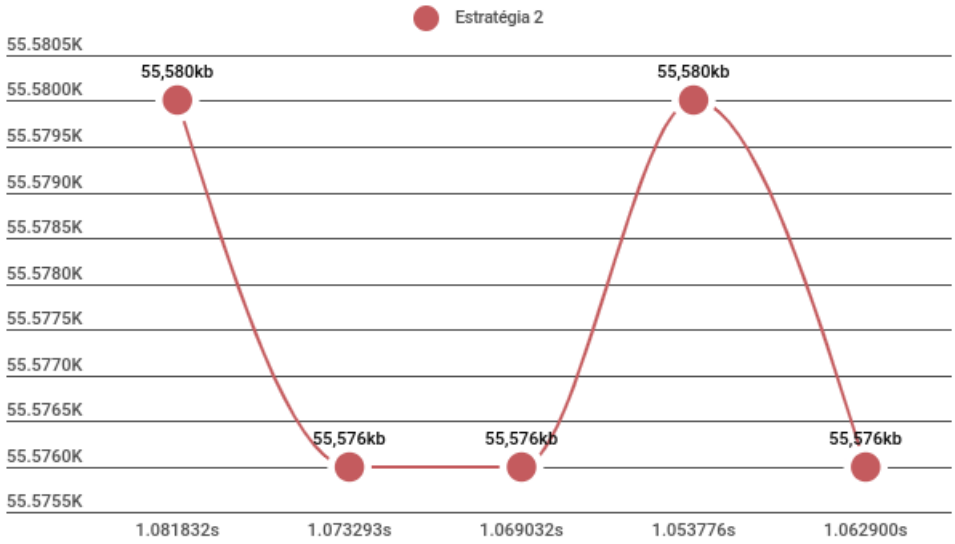


Figura 2: Estrategia 2

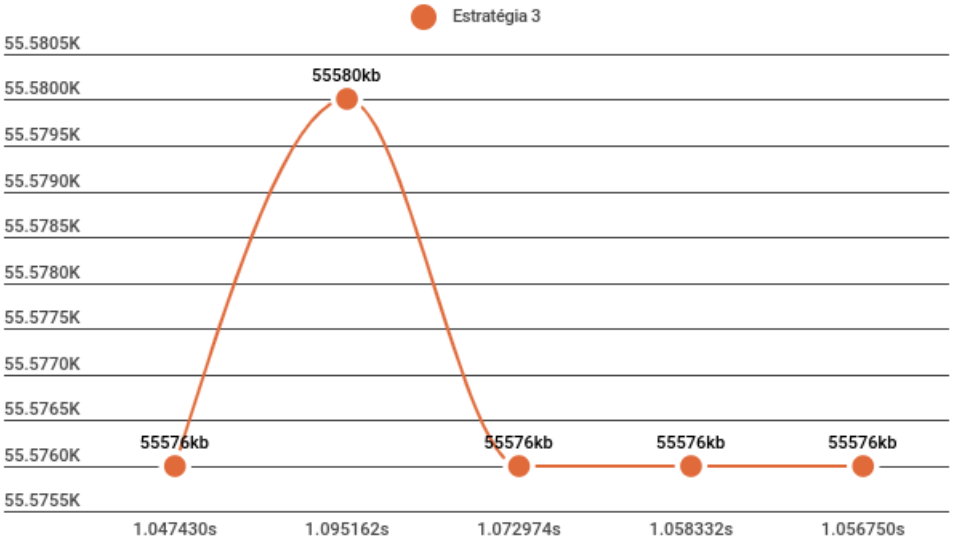
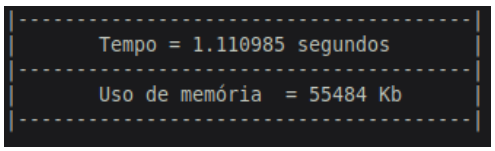
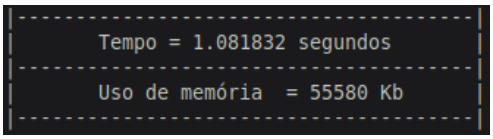


Figura 3: Estrategia 3

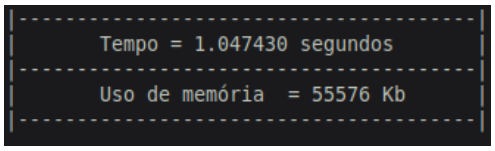
4.1.1 Resultados



(a) Estrategia 1



(b) Estrategia 2



(c) Estrategia 3

Figura 4

4.1.2 Comparação geral

De acordo com os resultados obtidos, pode-se dizer que os algoritmos de Força Bruta e Shift-And obtiveram os melhores resultados, como pôde ser visto no gráficos apresentados para cada estratégia. Na estratégia 1 foi obtido menor uso de memória, porém um tempo de execução maior, e pode ser visto um aumento crescente na memória até chegar ao valor 55576kb e não apresentar mais mudanças.

O algoritmo da estratégia 2, apesar de apresentar resultados positivos, não manteve uma performance estável, sendo possível notar picos em relação ao gasto de memória como foi visto no gráfico referente a essa estratégia.

Na estratégia 3, é visto que o algoritmo manteve um gasto de memória estável durante os testes realizados, com exceção de um dos testes, onde foi visto alto gasto de memória e tempo de execução alto. Porém pode-se dizer que obteve o resultado esperado.

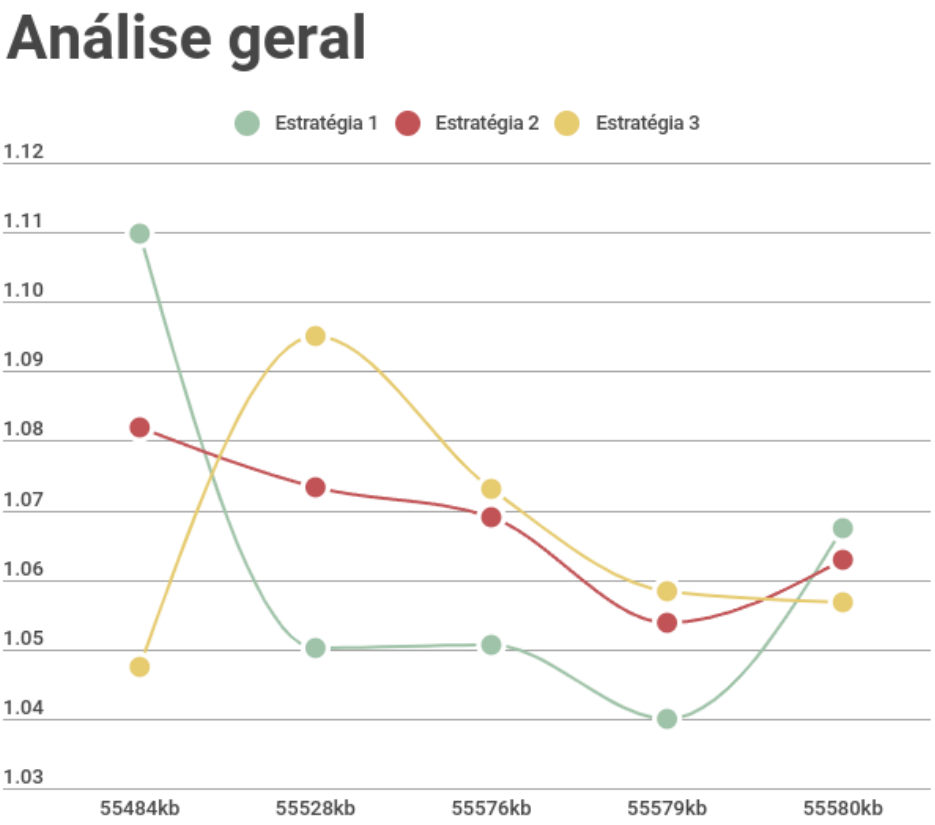


Figura 5: Gráfico geral de uso de memória por tempo de execução

5 Conclusão

O objetivo do trabalho foi buscar casamentos de padrões em pedras mágicas usando diferentes algoritmos. Foram implementados e comparados três algoritmos: Força Bruta, KMP e Shift-And. O problema surgiu no antigo reino de Xulambis, onde cada sequência de símbolos representava um poder único. Os pesquisadores buscaram separar as pedras mágicas das comuns usando um dicionário antigo encontrado no templo de Xulambis. A busca manual seria demorada e suscetível a erros, então os algoritmos foram implementados para determinar se uma sequência de poder estava presente em uma pedra mágica.

A estratégia 1 utilizou o algoritmo de Força Bruta para buscar casamentos de padrões. Essa estratégia percorreu as sequências de caracteres e comparou as strings utilizando dois índices. Foram realizadas iterações de ordem direta e reversa para verificar os casamentos. A função Casamento_forca_bruta percorreu a lista de pares de strings e chamou a função Forca_bruta para cada par, verificando se houve casamento ou não.

A estratégia 2 implementou o algoritmo KMP (Knuth-Morris-Pratt) para a busca de padrões. Esse algoritmo utilizou o pré-processamento do padrão para construir o array lps, que foi utilizado durante a busca para otimizar o processo. A função KMP_Busca realizou a busca direta do padrão no texto, enquanto a função KMP_Reverso realizou a busca reversa. A função Casamento_KMP percorreu a lista circular de nós e chamou as funções KMP_Busca e KMP_Reverso para buscar o padrão em cada nó, registrando os resultados.

A estratégia 3 utilizou o algoritmo Shift-And para buscar casamentos exatos do padrão no texto. Essa estratégia fez uso de manipulação de bits e de uma tabela de caracteres, chamada de "máscara", para realizar a busca de forma eficiente. Foram implementadas as funções ShiftAnd e ShiftAnd_Reverso para buscar casamentos diretos e reversos, respectivamente. A função Casamento_ShiftAnd percorreu a lista encadeada de nós e chamou as funções ShiftAnd e ShiftAnd_Reverso para buscar o padrão em cada nó.

De forma geral apesar ser encontrado dificuldades na implementação das estratégias, foi possível notar a importância dos algoritmos de casamento de padrões desempenham na computação, tendo um importante papel em diversas áreas. Entre elas estão o processamento de imagens e visão computacional, segurança cibernética, biometria e as áreas que foram o tema deste trabalho o reconhecimento de padrões em dados e processamento de linguagem natural.

6 Referências

https://www.geeksforgeeks.org/longest-prefix-also-suffix/

https://tungmphung.com/shift-and-algorithm-for-exact-pattern-matching/

https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

NivioZiviani.ProjetodeAlgoritmoscomimplementaÃğÃµesemJavaeC++.ThomsonLearning, 2007.

ThomasH.Cormen,CharlesE.Leiserson,RonaldL.Rivest,CliffordStein.Algoritmos: TeoriaePrÃątica.3aediÃğÃčo.Elsevier,2012.