UNIVERSIDADE DO ESTADO DA BAHIA
DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA
CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

MARCUS FILIPE COUTINHO SILVA SOARES

UM ESTUDO SOBRE O USO DE MEMÓRIA DO ALGORITMO DE KNUTH PARA JOGAR MASTERMIND, USANDO PYTHON

SALVADOR
2021

MARCUS FILIPE COUTINHO SILVA SOARES

UM ESTUDO SOBRE O USO DE MEMÓRIA DO ALGORITMO DE KNUTH PARA JOGAR MASTERMIND, USANDO PYTHON

Monografia apresentada ao curso de Sistemas de Informação do Departamento de Ciências Exatas e da Terra da Universidade do Estado da Bahia - UNEB, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação. Área de Concentração: Ciência da Computação

Orientador: Cláudio Alves de Amorim

Coorientador: Ernesto de Souza Massa Neto

SALVADOR

2021

FICHA CATALOGRÁFICA
Sistema de Bibliotecas da UNEB

S676e Soares, Marcus Filipe Coutinho Silva

Um estudo sobre o uso de memória do algoritmo de Knuth para jogar Mastermind, usando Python / Marcus Filipe Coutinho Silva Soares. - Salvador, 2021.
52 fls : il.

Orientador(a): Cláudio Alves de Amorim.
Coorientador(a): Ernesto de Souza Massa Neto.
Inclui Referências
TCC (Graduação - Sistemas de Informação) - Universidade do Estado da Bahia. Departamento de Ciências Exatas e da Terra. Campus I. 2021.

1.Mastermind. 2.Knuth. 3.Algoritmo. 4.Estudo. 5.Python.

CDD: 003

AGRADECIMENTOS

Agradeço ao meus pais, irmãs e sobrinha por todo o carinho e apoio não só ao longo da trajetória deste trabalho, mas por toda a minha vida.

Agradeço à Universidade do Estado da Bahia por possibilitar o acesso a uma educação pública de qualidade.

Por fim, agradeço aos professores e orientadores que me auxiliaram ao longo de toda a minha jornada acadêmica.

RESUMO

Mastermind é um jogo de tabuleiro de informação imperfeita, inventado em 1970 por Modecai Meirowitz, que recebeu popularidade mundial nas últimas décadas. É um jogo de dois jogadores que se assemelha ao jogo de "Bulls and Cows", originalmente jogado com lápis e papel. O objetivo deste trabalho foi realizar um estudo analítico do algoritmo determinístico proposto por Donald Knuth que influenciou diversos outros algoritmos posteriores na solução do jogo de Mastermind, no que diz respeito ao consumo de memória do mesmo durante o processo de execução utlizando a linguagem Python. Com base na documentação disponível sobre a linguagem Python, foi realizado um estudo sobre a forma de como esta realiza a alocação de memória de seus objetos e estruturas de dados, com a finalidade de fundamentar a criação de uma hipótese sobre o crescimento da curva do consumo total de memória esperado. Os resultados medidos empíricamente revelam que o consumo de memória do cabeçalho de objetos na linguagem Python é significativo, mantendo-se em cerca de 7% do consumo total de memória por objeto, mas não afeta a curva de crescimento de memória destes, que mantêm-se exponencial.

Palavras-chave: Mastermind. Knuth. Algoritmo. Estudo. Python.

ABSTRACT

Mastermind is a board game of imperfect information, created in 1970 by Modecai Meirowitz, which has gained worldwide popularity in the last decades. It is a two-player game resembling the game of "Bulls and Cows", originaly played with pencil and paper. The goal of this paper was to carry out an analytical study of the deterministic algorithm proposed by Donald Knuth to solve the game of Mastermind, regarding its memory usage during the execution process using the Python programming language. Based on the documentation available on Python, a study was made on how it allocates memory for its objects and data structures, in order to create an hipothesis on the expected growth of its total memory curve. The empirical results reveal that the memory consumption of an object header is significant in Python, about 7% of the total memory per object, but it doesn't affect the growth of their memory curve, which is kept exponential.

Keywords: Mastermind. Knuth. Algorithm. Study. Python

LISTA DE FIGURAS

Figura 1 – Exemplo do jogo de Mastermind descrito acima. 12
Figura 2 – Representação de um objeto padrão em Python. 19
Figura 3 – Esquema de uma tupla em Python. 20
Figura 4 – Esquema de uma lista de tuplas em Python. 21
Figura 5 – Representação de um objeto padrão em Python. 23
Figura 6 – Modelo da pilha privada no Python e as estruturas que ela contém. 25
Figura 7 – Inicialização das listas de combinações. 30
Figura 8 – Ciclo principal da implementação do algoritmo de Knuth. 31
Figura 9 – Curva esperada de crescimento das permutações possíveis. 33
Figura 10 – Crescimento do cabeçalho de memória de uma lista em relação a quantidade de permutações. 35
Figura 11 – Consumo de memória de uma lista de permutações, em escala logarítimica, pelo algoritmo de Knuth. 37
Figura 12 – Curva de consumo de memória do cabeçalho de uma lista pelo algoritmo de Knuth, na versão 32 bits. 41
Figura 13 – Curva de consumo de memória total de uma lista pelo algoritmo de Knuth, na versão de 32 bits. 42
Figura 14 – Curva de consumo de memória do cabeçalho de uma lista pelo algoritmo de Knuth, na versão 64 bits. 43
Figura 15 – Curva de consumo de memória total de uma lista pelo algoritmo de Knuth, na versão de 64 bits. 44

LISTA DE TABELAS

Tabela 1 – Média de bytes por quantidade de permutações 34
Tabela 2 – Média de bytes por quantidade de permutações. 36
Tabela 3 – Consumo de memória do cabeçalho de cada lista, em bytes, na versão de 32 bits 38
Tabela 4 – Consumo total de memória por cada lista, em bytes, na versão de 32 bits. . . 39
Tabela 5 – Consumo de memória do cabeçalho de cada lista, em bytes, na versão de 64 bits. 39
Tabela 6 – Consumo total de memória por cada lista, em bytes, na versão de 64 bits . . 40

LISTA DE ABREVIATURAS E SIGLAS

GIL *Global Interpreter Lock*

GPU *Graphics Processing Unit*

IA *Inteligência Artificial*

IDE *Integrated Development Environment*

SO *Sistema Operacional*

SSD *Solid State Drive*

SUMÁRIO

1 INTRODUÇÃO **11**
2 ABORDAGENS ALGORÍTIMICAS AO JOGO DE MASTERMIND . . **15**
2.1 Trabalhos Correlatos **16**
3 GERENCIAMENTO DE MEMÓRIA EM PYTHON **18**
3.1 Estruturas de dados **18**
3.1.1 Variáveis 18
3.1.2 Tuplas 20
3.1.3 Listas 21
3.2 Global Interpreter Lock **22**
3.3 Coletor de Lixo **22**
3.4 Pilha **23**
3.4.1 Arenas 23
3.4.2 Pools 24
3.4.3 Blocos 24
4 DESCRIÇÃO DO PROJETO **27**
4.1 Linguagem de Programação **28**
4.2 Ambiente de Desenvolvimento **28**
4.3 Controle de Versão **28**
4.4 Ambiente de Teste **29**
4.5 Implementação dos Algoritmo de Knuth **29**
5 ESTUDO TEÓRICO DO ALGORITMO DE KNUTH **32**
5.1 Estruturas de Dados Críticas **32**
5.2 Consumo de Memória Esperado **33**
6 RESULTADOS **38**
6.1 Versão de 32-bit **38**
6.2 Versão de 64-bit **39**
6.3 Comparação com Resultados Esperados **40**
7 CONSIDERAÇÕES FINAIS **45**

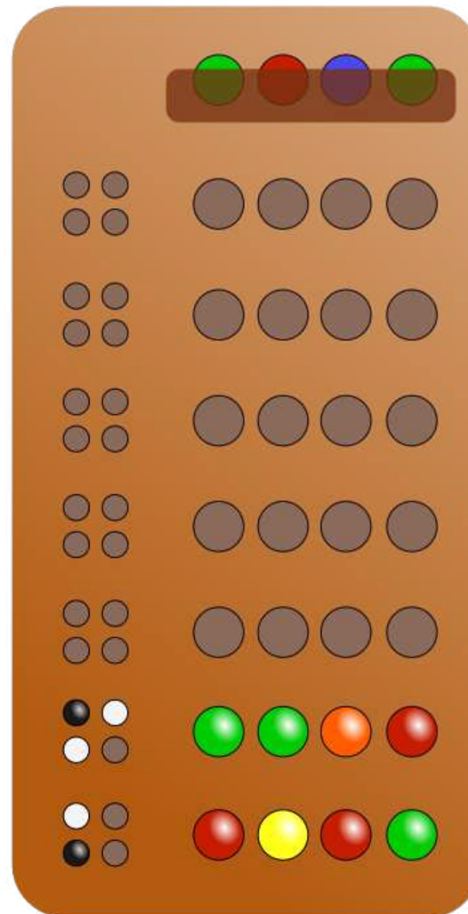
REFERÊNCIAS **46**
APÊNDICES **48**
APÊNDICE A – Código da implementação do algoritmo de Knuth **49**

1 INTRODUÇÃO

O Mastermind é um jogo de tabuleiro, conhecido no Brasil como *Senha*, que foi inventado pelo israelita Mordecai Meirowitz em 1970. Apesar de ser um jogo relativamente recente, sua versão atual é derivada de quebra-cabeças mais tradicionais, tais como o jogo de *bulls and cows*, também conhecido como *MOO*, e originalmente jogado com lápis e papel (FRANCIS, 2010). A empresa Invicta Plastics Ltd. comprou toda a propriedade e os direitos intelectuais e lançou uma versão comercial do jogo em 1971. Na época, tornou-se o jogo mais vendido da história, e é até hoje considerado o produto de maior impacto vendido pela empresa (INVICTA HISTORY(2007). Desde então mais de 55 milhões de cópias do jogo foram vendidas em mais de 80 países.

O jogo de Mastermind clássico é jogado por duas pessoas, no qual um jogador é responsável por criar um código secreto e o outro jogador desempenha o papel de criptoanalista, tentando desvendar a senha. O jogador desafiado deve criar um código que consiste de uma sequência de quatro cores (c1,c2,c3,c4) dentre um conjunto de seis cores possíveis, podendo ocorrer repetições. O jogador desafiante então deve tentar adivinhar o código secreto, realizando diversas tentativas, também na forma de uma sequência de cores (g1,g2,g3,g4). Após cada tentativa, o jogador desafiante recebe uma pista do jogador desafiado sob a forma de dois números (b,w): O número de cores corretas na posição correta (b) e o número de cores que fazem parte do código mas que não estão na posição correta (w). Por exemplo, se o código for (1,2,3,1) e a tentativa realizado pelo desafiante for (2,4,2,1), a resposta do jogador desafiado seria (1,1), visto que o *b* refere-se a uma cor correta e na posição correta, enquanto o *w* refere-se a uma cor correta mas na posição errada. Dessa forma o desafiante continua com as adivinhações até o código correto ser identificado ou até o número máximo de tentativas permitidas for alcançado.

Figura 1 – Exemplo do jogo de Mastermind descrito acima.



Fonte: Steiner (2006)

Mastermind também proporciona uma vasta gama de possibilidades em relação a formulação de estratégias para solucioná-lo. Desta forma, é um jogo que tem atraído a atenção de diversos pesquisadores e cientistas da computação ao longo dos últimos anos, que se debruçaram sobre ele na tentativa de criar diversos tipos de algoritmos para resolvê-lo.

Desenvolver e analisar soluções para o jogo de Mastermind traz grandes contribuições acadêmicas e científicas para a área da computação.

> *MasterMind is a challenging problem from the algorithmic point of view, but it is also interesting since any solution to MasterMind also yields insight on the solution of problems called generically black box or oracle problems, where some code has to be discovered by asking questions to a black box or oracle and getting hints as a result.(SALCEDO-SANZ; GUERVóS, 2014, p.2).*

Dessa forma, Mastermind é em essência um problema de pesquisa na qual um conjunto de símbolos que é mantido em segredo deve ser desvendado ao produzir sequencialmente conjuntos que usem o mesmo alfabeto, e utilizar as respostas que indicam o quão perto estes

novos conjuntos estão do código secreto como pistas para desvendá-lo. Exemplos práticos são facilmente encontrados na literatura, como o de Focardi (FOCARDI E LUCCIO, 2010), que demonstra a quebra de códigos PIN utilizado em caixas-eletrônicos, e o trabalho realizado pelo Goodrich (2009) que identifica unicamente uma pessoa a partir de um conjunto de caracteres resultante de solicitações a um banco de dados com informações genéticas. Além disso, Devido ao fato de Mastermind ser um quebra-cabeça NP-completo (STUCKMAN E ZHANG, 2006), qualquer solução utilizada para decifrá-lo pode também ser utilizada para resolver outros problemas da mesma natureza. Isto torna a busca por soluções e formulações de estratégias para solucionar o jogo extremamente atraentes.

Dentre estas diversas soluções, algumas estratégias se destacam por sua simplicidade e eficiência. São elas a estratégia MiniMax proposta por Knuth (1976), a estratégia da entropia proposta por Neuwirth (1981), a estratégia de Irving (1979), a solução em Prolog proposta por Shapiro (1983) e as estratégias Simples e Maior Partes propostas por Kooi (2005).

Durante a análise dos estudos acima, ficou claro que as principais métricas utilizadas para medir a eficiência dessa classe de algoritmos são: número máximo de tentativas para solucionar cada jogo, média e desvio padrão de tentativas para solucionar um jogo, primeira tentativa recomendada por um algoritmo e tempo médio de resolução.

Não foi encontrado, porém, nenhum foco na utilização de memória dos algoritmos utilizados. Visto que alguns dos algoritmos citados acima armazenam uma grande quantidade de dados em memória durante sua execução, é possível conjecturar que estas soluções sejam ineficientes quanto à escalabilidade. Percebeu-se também que a principal diferença entre estes algoritmos, ditos determinísticos, refere-se a estratégia utilizada para encontrar a próxima permutação a ser jogada, enquanto a forma com a qual as permutações são armazenadas na memória mantém-se a mesma do algoritmo originalmente proposto por Knuth (1976). Dessa forma, este algoritmo foi escolhido como representante da sua classe de algoritmos para o estudo do uso de memória, com relação à escalabilidade destes.

No caso de um simples jogo, não há necessidade de se pensar em escalabilidade, porém ao utilizar os conceitos relacionados ao Mastermind em outras aplicações, percebe-se que existem cenários onde a utilização de um grande número de cores e posições é inevitável. Por exemplo, o conceito de identificar a posição correta de uma cor, ao ser traduzida para pixels em uma tela, pode ser utilizado na reconstrução de imagens (BLAIR *et al.*, 2018).

Além disso, de acordo com o índice TIOBE (TIOBE Index BV, 2021), a linguagem Python é atualmente a linguagem mais popular de acordo com buscas realizadas por usuários nos maiores motores de busca. Python também é uma das linguagens mais utilizadas para abordar diversos problemas que requerem processamento de grandes massas de dados, sendo a linguagem chave em variadas aplicações no ramo da astronomia,*Inteligência Artificial* (IA), aprendizado de máquina, biologia, neurociência e modelagem molecular (RASHED E AHSAN,2012).

Assim, o presente trabalho visa realizar um estudo sobre a forma como a linguagem Python gerencia a alocação de memória, determinando a curva caracteristica do uso de memória de suas estruturas de dados e de seus respectivos cabeçalhos de memória, através do algoritmo proposto por Knuth (1976) para solucionar o jogo de Mastermind.

O restante do trabalho está estruturado da seguinte forma: A próxima seção traz diferentes abordagens algorítmicas, se aprofundando mais nos conceitos do jogo e apresentando trabalhos relacionados. O Capítulo 3 discute a forma como a linguagem Python aloca memória para seus objetos. A metodologia de pesquisa utilizada e seus conceitos são introduzidas no capítulo 4, descrevendo toda a trajetória do desenvolvimento do trabalho, e as decisões tomadas que definiram o seu formato final. O capítulo 5 consiste em um estudo preliminar do algoritmo, utilizando os conceitos introduzidos para prever matematicamente o consumo de memória do algoritmo. Em seguida, no capítulo 6, é feita uma descrição de como o algoritmo foi implementado em Python. No capítulo 7, são apresentados os resultados empíricos, e é feita uma comparação com os resultados esperados. Por fim, no capítulo 8 são apresentadas as considerações finais, conclusões e futuras linhas de pesquisa.

2 ABORDAGENS ALGORÍTIMICAS AO JOGO DE MASTERMIND

Berghman *et al.* (2009) divide as diferentes classes de algoritmos existentes para solucionar o jogo de Mastermind em três tipos:

- Enumeração completa: Compara todas os códigos possíveis e retêm aquela com a maior pontuação de acordo com uma função objetiva.
- Regra de Ouro: Constrói uma lista em ordem aleatória e contínua da última posição examinada até achar um código elegível.
- Meta-Heurísticas: Utiliza um código de ordem aleatória e conhecimento histórico para escolher o próximo código.

A maioria das soluções, principalmente as de enumeração completa, usam o conceito de códigos possíveis, ou seja, aqueles que de acordo com a resposta do jogador desafiado ainda podem ser o código secreto. Dessa forma, é possível diminuir o espaço de busca, porém, a capacidade de redução desse espaço depende de uma pontuação (Guervós *et al.*, 2013).

O cálculo dessas pontuações leva em consideração a resposta que um código irá gerar ao ser comparada com todos os outros códigos no conjunto. Ou seja, considera-se a quantidade de códigos que ainda serão possíveis e como eles estarão agrupados caso aquele código seja utilizado na próxima tentativa de desvendar a senha. Dessa forma, várias partições são criadas, baseadas em termos de cores e posições em comum em relação a cada uma.

A distribuição destas partições são então utilizadas para resolver o problema utilizando vários métodos diferentes. Dentre eles, a mais famosa é provavelmente a proposta feita por Knuth (1976), que sempre consegue descobrir o código secreto em cinco tentativas ou menos. Nesta estratégia os códigos são escolhidos em relação ao tamanho da partição, quanto maior, pior. Outros métodos, como o de Most Parts, proposta por Kooi (2005) somente leva em consideração as partições que contenham algum elemento, enquanto a estratégia da Entropia proposta por Neuwirth (1981) calcula a entropia de cada partição e escolhe a maior (Guervós *et al.*, 2013).

Experimentos realizados na literatura demonstram que as melhores estratégias de enumeração completa são a de Most Parts e Entropy, para problemas de todos os tamanhos, e

que a diferença entre os resultados delas são insignificantes (Guervós *et al.*, 2012).

Atualmente, a maior parte dos artigos publicados em relação ao jogo de Mastermind foca em algoritmos meta-heurísticos, principalmente algoritmos genéticos. Uma possível explicação deve-se ao fato de que Koyama(1994) encontrou a estratégia ótima para solucionar o jogo de Mastermind clássico utilizando uma estratégia de busca em profundidade. E como a complexidade de um algoritmo de enumeração cresce exponencialmente ao aumentar-se o número de cores e de posições, visto que este tipo de algoritmo leva em consideração todas as combinações possíveis, formulações de estratégias matemáticas para resolver o jogo de Mastermind com esta classe de algoritmos tornou-se menos atrativa. De qualquer forma, estudos com um maior número de cores e posições ainda são relevantes, visto que a solução ótima para maiores instâncias do problema ainda não foram encontradas.

Já em relação a meta-heurísticas, o algoritmo proposto por Berghman *et al.*, (2009) conseguiu o melhor número médio de jogadas para todos os casos testados e é considerado como o melhor algoritmo genético.

2.1 TRABALHOS CORRELATOS

Distintos artigos apresentam comparações entre os diferentes algoritmos e soluções para o jogo de MasterMind, porém durante o processo de revisão de literatura, não foi encontrado nenhum artigo que desse importância quanto ao uso de memória utilizado durante a execução destes algoritmos.

O artigo de Dowell (2009) apresenta uma comparação entre diversos algoritmos em relação ao número de jogadas, de média, de desvio padrão e da quantidade máxima de jogadas para resolver o problema.

Já Berghman *et al.*, (2009) introduz um novo algoritmo genético e faz uma comparação com outros algoritmos no que diz respeito ao número médio de jogadas e de tempo computacional. Não implementou os mesmos e os números utilizados foram meramente copiados da literatura.

Kooi (2005) apresenta uma explicação de estratégias utilizadas por outros autores, as implementa e compara com seus algoritmos propostos, encontrando inclusive resultados diferentes do apresentado no artigo do Irving (1979).

O artigo de Rybár (2014) implementa um algoritmo evolutivo em duas linguagens diferentes e os compara com outros algoritmos, embora não os tenha implementado, utilizando dados disponíveis na literatura.

Em todos os casos, como citado anteriormente, é perceptível a falta de atenção para o problema do crescimento do uso de memória nestes algoritmos e a necessidade da realização de estudos na área.

3 GERENCIAMENTO DE MEMÓRIA EM PYTHON

O interpretador da linguagem Python, conhecido como CPython, é por padrão implementado na linguagem C, assim como o de diversas outras linguagens de alto nível (ROCKYB, 2020). Existem também outras implementações da linguagem Python, e o gerenciamento de memória entre elas varia, dentre elas as mais utilizadas são:

- IronPython: Implementação Python em .NET.
- Jython: Implementação Python utilizando uma Java Virtual Machine.
- PyPy: Implementação Python utilizando um compilador JIT, com foco em velocidade de processamento.

Em CPython, um alocador de memória garante que existe espaço suficiente para para que todos os objetos e estruturas de dados relacionados ao Python possam ser armazenados em uma pilha privada, ao interagir com o gerenciador de memória do sistema operacional. Além disso, alocadores de memória específicos para cada objeto operam sobre a pilha, implementando políticas de gerenciamento de memória distintas de acordo com o tipo de cada um deles. Por fim, o Coletor de Lixo é responsável por liberar objetos da pilha que não estejam sendo utilizados (Python Software Foundation, 2021).

3.1 ESTRUTURAS DE DADOS

3.1.1 Variáveis

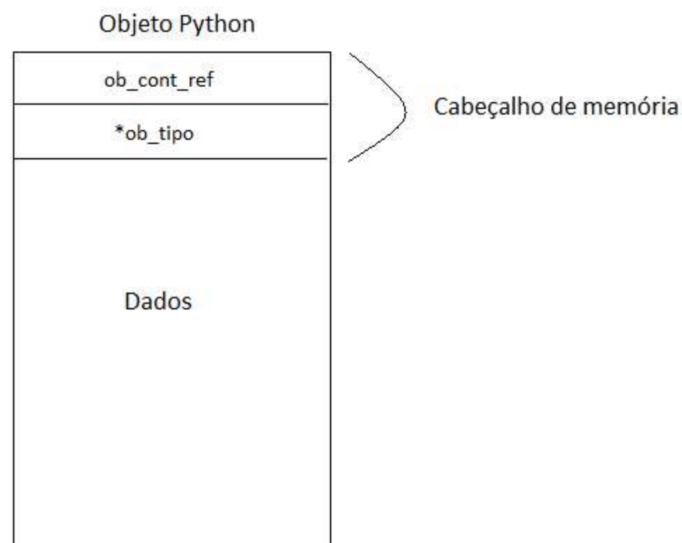
O que consideramos uma variável em Python é diferente do conceito de variável na linguagem C, por exemplo. Em Python todas as variáveis e estruturas são tratadas como objetos.

Em relação a estes objetos, qualquer objeto python inclui ao menos: Um contador de referências, para saber quantas referências existem para aquele objeto, um ponteiro para o tipo do objeto em questão (inteiro, string, etc),e os demais dados necessários para guardar o objeto em si (Python Software Foundation, 2021b). Dessa forma, não importa que o tamanho do dados guardado seja pequeno, o cabeçalho de memória sempre irá existir. Em alguns casos, o cabeçalho de memória pode ser inclusive maior do que o dado relevante.

Por fim, os ponteiros de memória ainda podem ocupar mais espaço, dependendo do *Sistema Operacional* (SO). Em um SO de 32 bits, o padrão são 2 itens para um total de 8 bytes por cabeçalho de memória para cada objeto. Para 64 bits, esse valor aumenta para 16 bytes.

A figura abaixo ilustra um objeto padrão na linguagem Python e seu cabeçalho de memória.

Figura 2 – Representação de um objeto padrão em Python.



Fonte - O autor.

Existem diversas formas possíveis de se representar as cores e posições em um algoritmo para jogar Mastermind. Tais como uma cadeia de caracteres *AABB*, um inteiro longo *1122* ou um vetor de inteiros *[1,1,2,2]*. Ao escolher a representação utilizando um vetor de inteiros, por exemplo, uma das estruturas que poderiam ser utilizadas para armazenar esses dados na linguagem C seria uma matriz de inteiros. Esta escolha deve funcionar bem para um pequeno número de cores e posições, com um baixo consumo de memória, sendo uma relação direta com o tamanho de um inteiro. Porém, matrizes alocadas estaticamente não podem ser redimensionadas, e como o algoritmo de Knuth (1976) executa diversas operações de remoção de elementos da matriz, isto causaria um impacto significativo na performance do algoritmo com um número elevado de cores e posições.

Uma alternativa seria usar uma lista ligada, onde somente as referências para cada estrutura teriam que ser atualizada após cada remoção. Uma desvantagem dessa abordagem é aumento do consumo de memória devido à adição destes ponteiros, cujo cálculo deixa de ser tão

simples uma vez que a função *malloc()* contém um pequeno overhead para auxiliar na liberação de memória.

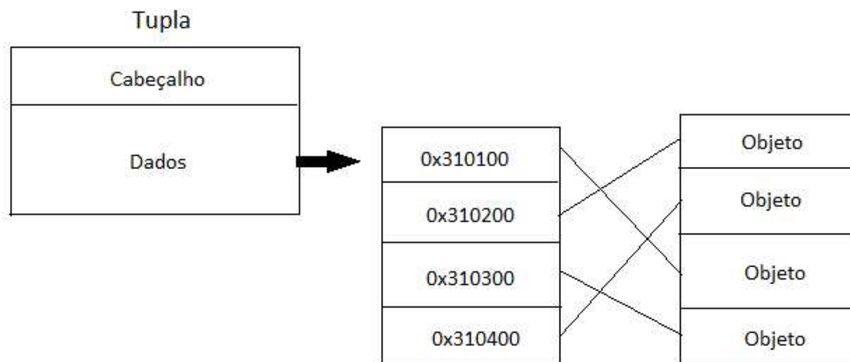
De volta à linguagem Python, uma estrutura de dados equivalente às discutidas acima é uma lista de tuplas, que é bem parecida com uma matriz de inteiros em sua estrutura, porém, contém funções embutidas para a execução de operações sobre ela.

3.1.2 Tuplas

Tuplas são sequências imutáveis, geralmente usadas para guardar coleções de dados heterogêneos. Tuplas também são usadas em casos onde uma sequência imutável de dados é necessária, tais como permitindo armazenamento em uma instância de um *set* ou *dict* (Python Software Foundation, 2021a).

Como são imutáveis, tuplas guardam uma sequência (*array*) de referências para os elementos que ela contém diretamente dentro de sua estrutura como mostra a figura 3. Dessa forma, o tamanho de uma tupla será uma função linear do número de elementos pertencentes à ela. (Python Software Foundation, 2021c)

Figura 3 – Esquema de uma tupla em Python.



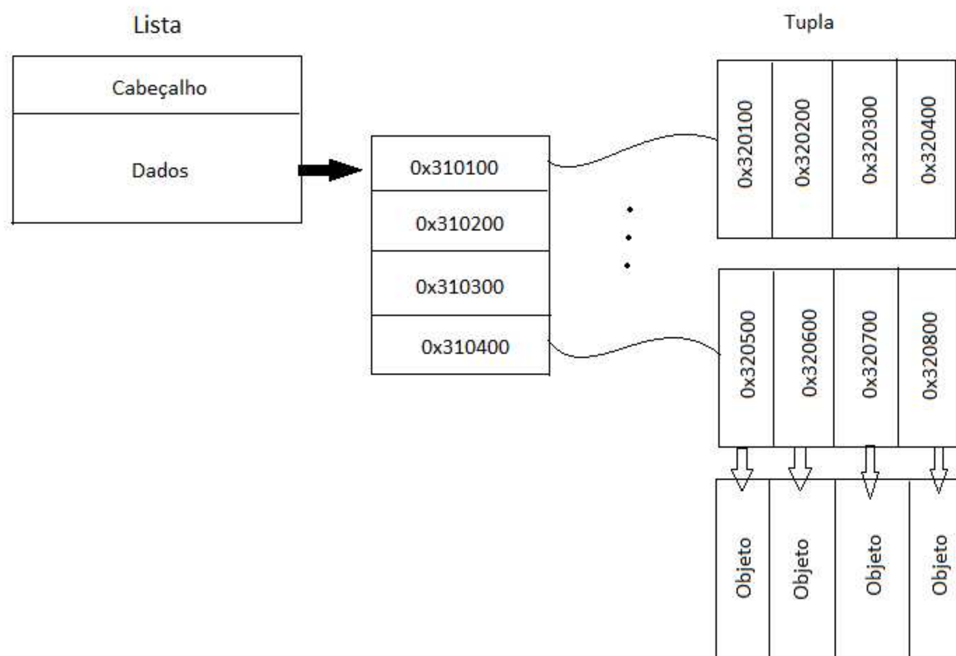
Fonte - O autor.

3.1.3 Listas

Listas são sequências mutáveis, tipicamente usadas para guardar coleções de itens homogêneos (onde o grau de similaridade varia por aplicação) (Python Software Foundation, 2021a)

Diferente de uma tupla, uma lista guarda um ponteiro para uma sequência (*array*) de referências para cada elemento que ela contém. Dessa forma, uma lista nada mais é do que uma coleção de referências para um objeto original. A figura 4 ilustra uma lista de tuplas:

Figura 4 – Esquema de uma lista de tuplas em Python.



Fonte - O autor.

Python aloca espaço para listas de forma que um espaço amortizado constante seja estabelecido quando elementos forem adicionados à lista. Desta forma listas reservam espaço para mais objetos do que elas contém, na maioria dos casos. Este tamanho varia de como a lista foi criada, além do seu histórico de apêndices e remoções (Python Software Foundation, 2021a).

Isso também significa que cada lista precisa guardar e gerenciar o tamanho alocado, adicionando um novo campo em seu cabeçalho de memória.

3.2 GLOBAL INTERPRETER LOCK

O *Global Interpreter Lock* (GIL) é uma flag que protege objetos Python, prevenindo que multiplas threads executem bytecodes Python ao mesmo tempo. Isto se faz necessário pois o gerenciamento interno de memória do interpretador Python não é thread-safe. Ou seja, o interpretador CPython que reserva memóra para a criação de objetos e estruturas de dados, como tuplas ou listas, usa o GIL para bloquear o interpretador por completo, garantindo que outra thread não irá deixá-las em um estado inconsistente (Python Software Foundation, 2021).

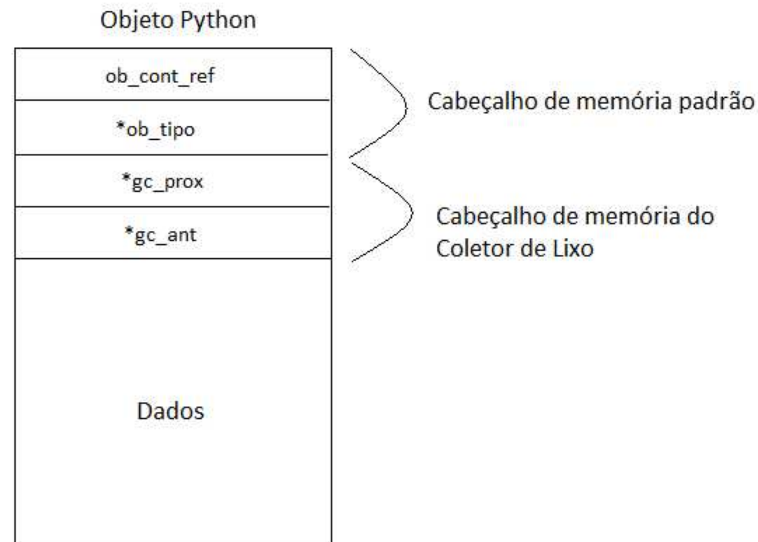
O GIL também é responsável por gerir as estruturas de dados tais como listas, dicionários e frames de execução, garantindo seu funcionamento em um código que utilize várias threads. (ERIKY, 2020)

3.3 COLETOR DE LIXO

O coletor de lixo é um processo no qual o interpretador libera memória quando esta não está sendo utilizada, para que possa ser usada por outros objetos. A liberação de memória é, na maioria das vezes, tratada contando as referências para cada objeto. Ou seja, o Python mantém uma tabela de quantas referências existem para um determinado objeto e o coletor de lixo automaticamente o remove da pilha caso esse valor seja zero (Python Software Foundation, 2021b).

Um problema com esse método é a existência de referências circulares, por exemplo, quando um objeto A faz referência a um objeto B que faz referência ao objeto A novamente. Para isso, o Coletor de Lixo adiciona referências de controle nos cabeçalhos de memória de objetos que podem conter referências a um ou mais objetos, tais como listas e dicionários, conforme figura abaixo (SALGADO, 2021).

Figura 5 – Representação de um objeto padrão em Python.



Fonte - O autor.

De maneira resumida, o ciclo de coleta de lixo em Python funciona da seguinte forma:

- O Python inicializa uma lista para objetos não utilizados.
- É executado um algoritmo para detectar referências circulares.
- Caso o objeto não tenha referências externas, é adicionado a lista.
- A memória alocada para os objetos da lista é liberada.

3.4 PILHA

O gerenciamento de memória na linguagem Python envolve a utilização de uma pilha privada contendo todos os objetos e estruturas de dados existentes (Python Software Foundation, 2021b). Essa pilha permite que os objetos sejam acessados globalmente por todos os métodos do programa. O gerenciamento da pilha é controlado pelo próprio interpretador e o usuário não possui controle sobre ele, mesmo que este manipule referências de objetos para blocos de memória dentro da pilha. (ZEHRA et al., 2020)

3.4.1 Arenas

Arenas representam a maior estrutura de memória na linguagem Python, com um tamanho fixo de 256 KiB e são os objetos que compõem a pilha. Mais especificamente, arenas são mapeamentos de memória usadas pelo alocador Python *pymalloc*, otimizado para pequenos

objetos (igual ou menor que 512 bytes). Arenas são as responsáveis por alocar memória, e estruturas subsequentes não o fazem (Python Software Foundation, 2021).

Uma arena pode ser dividida em 64 pools, que representa a próxima maior estrutura de memória no Python. A memória só pode ser liberada para o *SO* na forma de uma arena, ou seja, somente quando uma arena estiver vazia. Da mesma forma, o tamanho mínimo de memória que poder ser alocado também é uma arena (ZEHRA et al., 2020).

3.4.2 Pools

Pools são de tamanho fixo de 4 KB cada. Cada Pool mantém uma lista ligada dupla para outras pools da mesma arena (ZEHRA et al., 2020). Dessa forma, o Python pode facilmente encontrar espaço disponível mesmo dentre diferentes pools. Os alocadores de memória Python também mantém listas de referências para auxiliar no gerenciamento de memória. A lista *usedpools* guarda todos os pools que possuem blocos disponíveis de acordo com o tamanho destes, enquanto a *freepools* guarda todos os pools vazios (Python Software Foundation, 2021).

Pools precisam em um de três estados disponíveis:

- Vazio: O pool está vazio e disponível para alocação.
- Usado: O pool contém blocos não alocados.
- Alocado: O pool está completamente cheio, e não possui espaço para alocação.

Um pool pode ser então dividido em vários blocos, que são a menor estrutura de memória.

3.4.3 Blocos

O tamanho de um bloco não é fixo, podendo variar de 8 até 512 bytes, em múltiplos de oito (ZEHRA et al., 2020). Cada bloco somente guarda um objeto Python e possui três estados possíveis:

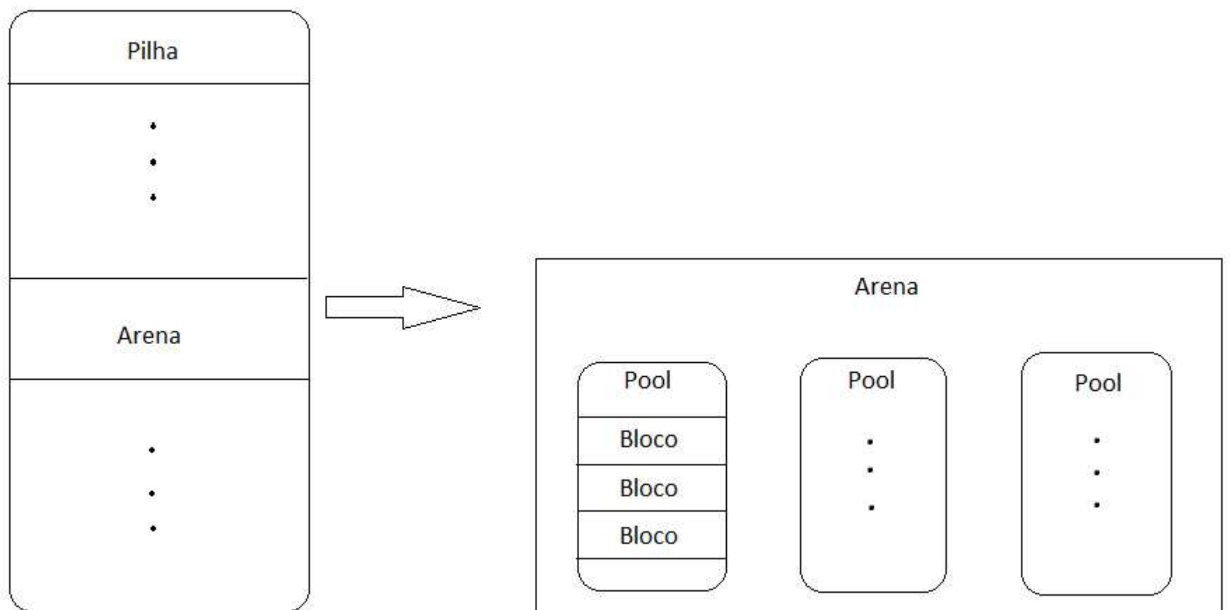
- Vazio: O bloco está vazio e disponível para alocação.
- Liberado: O bloco foi alocado, mas liberado novamente para alocação.
- Alocado: O bloco está alocado.

Os alocadores de memória do Python não necessariamente liberam a memória de

volta para o *SO*. No caso de pools e blocos, a memória alocada e devolvida ao interpretador Python para ser utilizada novamente. Dessa forma, o Python sempre possui uma pequena reserva de memória não alocada durante a execução (Python Software Foundation, 2021).

A figura 6 ilustra a forma como está organizada a pilha privada e suas estruturas: Arenas, pools e blocos.

Figura 6 – Modelo da pilha privada no Python e as estruturas que ela contém.



Fonte - O autor.

É importante perceber que arenas, pools e blocos são especificamente para pequenos objetos em Python. Objetos maiores são gerenciados pelo alocador CPython padrão (Python Software Foundation, 2021).

4 DESCRIÇÃO DO PROJETO

Como visto anteriormente, existe uma lacuna no que diz respeito ao estudo do uso de memória das soluções propostas para resolver o jogo de Mastermind. Durante este estudo, percebeu-se que os algoritmos de enumeração completa (determinísticos) utilizam a mesma estratégia de alocar todas as combinações possíveis em memória que o algoritmo de Knuth (1976), pioneiro na área, mudando somente a forma com a qual a próxima combinação é escolhida. Foi decidido que um estudo aprofundado do uso de memória deste algoritmo refletiria com os resultados de todos os outros algoritmos da mesma classe.

Após a definição que o algoritmo de Knuth (1976) seria utilizado, foi realizado um estudo teórico do mesmo, a fim de elaborar pressupostos quanto ao uso de memória esperado. Nesta etapa foi necessário definir com clareza quais são as estruturas de dados críticas deste algoritmo, para que durante a implementação fosse possível medir a utilização de memória do mesmo.

A segunda fase tratou-se da extração do pseudocódigo do artigos relevante e implementação dos algoritmo em questão. Nessa etapa do processo o algoritmo foi codificado, e então submetido a testes em relação a qualidade do código e validação de erros.

A próxima etapa foi a coleta dos resultados. Esta etapa consistiu basicamente na execução sistemática do algoritmo implementado e registro do consumo de memória durante cada execução.

Em seguida, foram deduzidas algebricamente as funções características, relacionando o tamanho da instância do problema (número de pinos e cores) com a quantidade de memória utilizada. Foram traçadas as curvas características do uso de memória a partir da implementação e testes e apresentados na forma de gráficos para facilidade de visualização.

Ao final desse processo, foram respondidas as questões pertinentes quanto à escalabilidade dos algoritmos, pautadas nos resultados adquiridos, suprindo assim o objetivo geral da pesquisa.

4.1 LINGUAGEM DE PROGRAMAÇÃO

A linguagem Python é uma linguagem que vem ganhando popularidade ao longo dos anos, sendo muito utilizada no desenvolvimento de softwares complexos (RASHED E AHSAN,2012), por isso foi escolhida como a linguagem de programação alvo deste estudo e utilizada para implementar o algoritmo em questão. Python é uma linguagem gratuita de alto nível, com tipagem dinâmica e com uma vasta gama de bibliotecas disponíveis voltadas à análises matemáticas, criando assim uma vantagem sobre outras linguagens de programação como Java ou C++.

Python disponibiliza fácil criação e manipulação de matrizes e objetos, componentes extremamente importantes, visto que o jogo de Mastermind também utiliza conceitos parecidos com o paradigma de orientação a objetos. Python também é uma linguagem extremamente flexível, onde não é necessário modificar o código para executá-lo em outra plataforma, tornando-a uma linguagem prática para utilização.

4.2 AMBIENTE DE DESENVOLVIMENTO

Para qualquer desenvolvimento de software é necessário a existência de um ambiente de desenvolvimento, onde o código em questão será produzido. Em um *Integrated Development Environment* (IDE) todas as ferramentas que um desenvolvedor pode utilizar ao longo do desenvolvimento estão presentes em um único ambiente. Ferramentas que completam nomes e variáveis automaticamente, detectam erros em tempo real e facilitam a navegação no código, auxiliam no processo de desenvolvimento, poupando tempo e esforço do programador.

Para este projeto, o IDE escolhido foi o *Visual Studio Code*, por ser um IDE versátil e de fácil utilização. Neste ponto, foi levado também em consideração a proficiência com o IDE em questão, visto que é necessário o conhecimento sobre o ambiente de desenvolvimento para que as vantagens que ele traz possam ser utilizadas ao máximo.

4.3 CONTROLE DE VERSÃO

Versionamento de software é um item necessário durante o desenvolvimento de qualquer projeto. Pois só assim pode-se estabelecer uma forma simples de manter e de acompanhar o código implementado, facilitando a correção de erros e evitando a perda de informações em

caso de acidentes. O controle de versão permite restaurar o código para um estado anterior, bem como detalha cada modificação feita ao longo do tempo.

Foi importante também escolher um software de controle de versão de fácil utilização e que pudesse ser integrado ao ambiente de desenvolvimento. O Git é um software de controle de versão extremamente simples de se utilizar, gratuito e que pode ser facilmente integrado ao Visual Studio Code, mostrando em tempo real as mudanças no código e permitindo que os comandos relacionados ao versionamento pudessem ser feitos pela própria ferramenta. Dessa forma, foi assim escolhido como software a ser utilizado.

4.4 AMBIENTE DE TESTE

Durante um processo de comparação entre diferentes algoritmos, a criação de um ambiente de testes é fundamental. Diversos fatores externos ao algoritmo sendo testado podem impactar na performance do mesmo, tal como a execução de um processo em segundo plano pelo sistema operacional, gerando assim resultados inconsistentes.

A criação de um ambiente de teste estéril não é um processo tão simples, e adicionaria uma complexidade desnecessária ao desenvolvimento do trabalho. Para contornar este problema, serão utilizadas bibliotecas internas da própria linguagem Python e suas funções, tal como a *sys.getsizeof(object)* para medir o uso de memória utilizado durante as execuções dos algoritmos.

As especificações do computador utilizado na análise são:

- Sistema Operacional: Windows 10 Home 64 Bit
- Memória Principal: 16 GB DDR3
- Processador: Intel Core i7 - 4770 @3.40 GHz
- *Graphics Processing Unit* (GPU): Nvidia GTX 1660 Super, 6 GB, GDDR6
- *Solid State Drive* (SSD): Kingston A400, 120GB, SATA, Leitura 500MB/s, Gravação 320MB/s

4.5 IMPLEMENTAÇÃO DOS ALGORITMO DE KNUTH

Com base nos princípios estabelecidos anteriormente, e com as especificações disponíveis nos artigos relevantes, foi realizada a implementação do algoritmo e os detalhes da codificação deles serão discutidos a seguir.

No caso do algoritmo proposto por Knuth (1976) a lista de combinações *S* e a lista de todas as combinações *codigo* foram definidas como uma lista de tuplas, e as cores foram representadas por números, variando de 1 até o tamanho de cores definido. Ambas as listas são iniciadas igualmente, contendo todas as combinações possíveis para aquela quantidade de cores e posições, a lista *S*, porém, terá elementos removidos com o tempo, enquanto a lista *codigo* se mantém estática, e será usada para garantir a vitória com cinco jogadas (no modo clássico) de acordo com a estratégia de *minimax*.

Figura 7 – Inicialização das listas de combinações.

```
S = [tuple(x) for x in product(range(1,len(CORES)+1), repeat=QTD_POSICOES)]
codigo = [tuple(x) for x in product(range(1,len(CORES)+1), repeat=QTD_POSICOES)]
```

Fonte - O autor.

De acordo com a estratégia definida no artigo, a tentativa inicial sempre é feita com a combinação *(1,1,2,2)*. O ciclo principal do algoritmo é executado enquanto o número do *feedback[0]* (número de cores na posição correta) for diferente do número de posições. Neste ciclo, uma cópia auxiliar de *S* é feita para permitir a iteração da lista *S* sem extrapolar os limites desta, e durante esta iteração, todos os códigos cujo número de pinos pretos e brancos (feedback), ao comparados com a tentativa, forem diferentes do número de pinos recebidos durante a comparação com o código secreto, são removidos da lista *S*.

Uma função *minimax* visa minimizar a perda caso o pior caso aconteça. Neste algoritmo, o objetivo desta função é dar um score para todas as combinações possíveis, inclusive as que não pertencem a *S*, buscando adquirir o máximo de informação. Knuth (1976) dá um exemplo onde pode ocorrer de o algoritmo fazer uma jogada que não pode ser o código secreto (não está em *S*) com a finalidade de garantir a solução em 5 tentativas para o modo clássico com seis cores e quatro posições. Esta função é então responsável por escolher a próxima combinação a ser jogada.

Figura 8 – Ciclo principal da implementação do algoritmo de Knuth.

```
combinacao=(1,1,2,2)

while feedback[0] != QTD_POSICOES:

    aux = S.copy()
    attempts += 1
    for i in range (0,len(S)):
        if calculaFeedback(S[i],combinacao) != feedback:
            aux.remove(S[i])
    S = aux

    combinacao = minMax()

    feedback = calculaFeedback(combinacao,codigoSecreto)
```

Fonte - O autor.

5 ESTUDO TEÓRICO DO ALGORITMO DE KNUTH

Nesta seção foi realizado um estudo do algoritmo de Knuth (1976), um algoritmo de enumeração completa, com a finalidade de melhor compreendê-lo e, de acordo com as especificações da linguagem Python, formular uma hipótese matématica sobre a curva de crescimento do uso de memória de uma implementação deste algoritmo.

5.1 ESTRUTURAS DE DADOS CRÍTICAS

Diz respeito às estruturas de dados que irão variar de tamanho de acordo com a escalabilidade do problema. A identificação e acompanhamento destas estruturas se faz necessária para identificar o consumo de memória utilizado durante o processo de execução dos algoritmos.

O algoritmo proposto por Knuth (1976) cria uma lista *S* com todas as permutações possíveis e ao longo da execução percorre esta lista removendo todas as permutações que não podem ser o código secreto. O algoritmo também mantém uma lista com todas as permutações possíveis que não é alterada, utilizada durante a estratégia de *MinMax* para garantir a vitória com no máximo cinco jogadas.

Desta forma as estruturas de dados críticas identificadas no algoritmo de Knuth são:

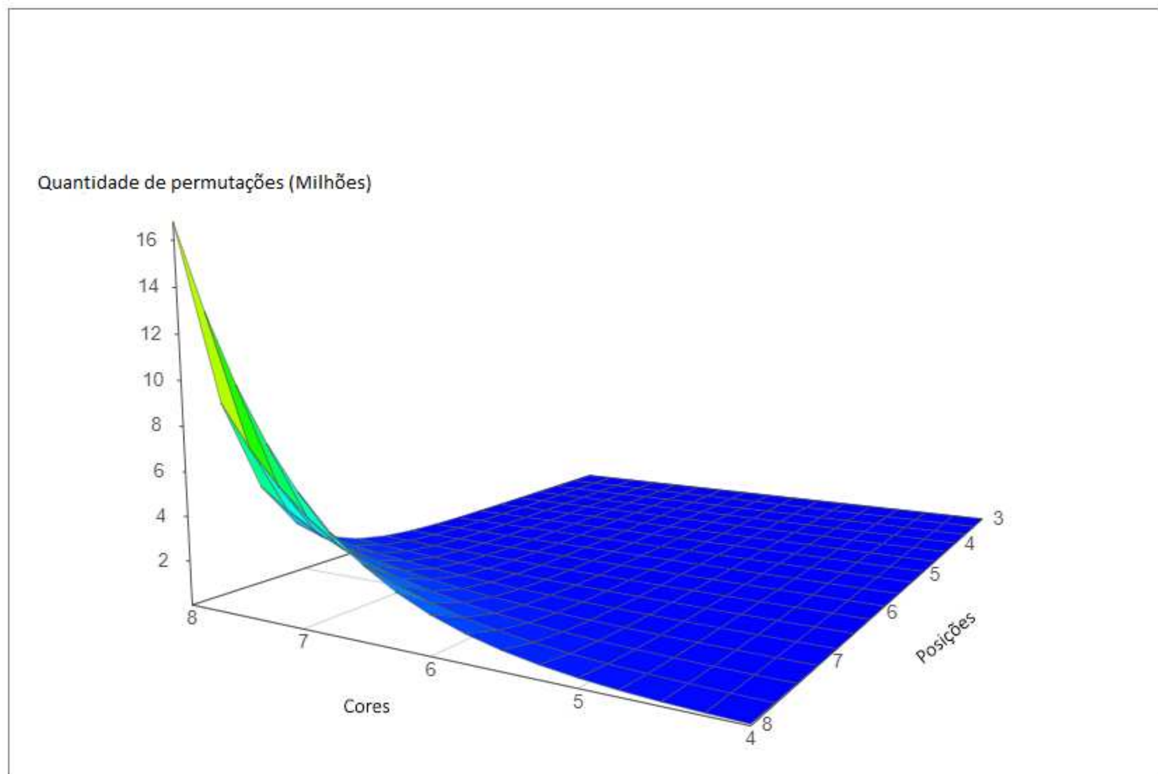
- Lista de permutações: Lista com todas as permutações possíveis.
- S: Lista de todas as permutações possíveis, reduzida ao longo da execução.
- Lista Auxiliar: Cópia de S, utilizada temporariamente para percorrer a lista S e remover elementos.

A estrutura acima diz respeito a forma original de como o algoritmo foi descrito no artigo e implementada. Porém, esta estrutura pode ser otimizada para melhorar o consumo de memória. Em uma implementação utilizando vetores, por exemplo, as permutações excluídas da lista *S* seriam enviadas para o fim do vetor, e uma variável contendo o índice limite para buscas naquele vetor seria atualizado. Desta forma, a lista *Lista de permutações* se tornaria redundante, visto que a lista *S* ainda iria conter todas as permutações possíveis.

5.2 CONSUMO DE MEMÓRIA ESPERADO

O algoritmo proposto por Knuth (1976) utiliza o número máximo de permutações possíveis, este número é dado por C^P onde C é o número de cores e P é o número de posições. Desta forma, é fácil presumir que a função resultante do consumo de memória seja proporcional a esta função exponencial, conforme mostra a figura 9.

Figura 9 – Curva esperada de crescimento das permutações possíveis.



Fonte - O autor.

Como explicado anteriormente, o cabeçalho de memória é um fator decisivo no consumo total de memória de qualquer aplicação Python. Conjectura-se, porém, que a medida que o tamanho do número de pinos e cores aumente, o tamanho deste cabeçalho torne-se insignificante, e que a curva de crescimento de memória deve manter-se exponencial.

Para fundamentar essas expectativas, foram realizados uma série de testes utilizando a versão 3.7.4 do Python de 32 bits, com a finalidade de encontrar valores mais concretos em relação aos resultados esperados. Isso foi possível pois percebeu-se que durante estes testes, o uso de memória para a mesma instância de problema permanece a mesma dentre diferentes execuções.

A função nativa *sys.getsizeof(object)* retorna somente o consumo de memória diretamente atribuído ao objeto passado (tamanho do objeto em si e tamanho da sequência de ponteiros), mas não dos objetos referidos por ele (Python Software Foundation, 2021d).

Assim, utilizando a função *sys.getsizeof()* para medir o cabeçalho de memória, foram encontrados os valores a seguir:

Tabela 1 – Média de bytes por quantidade de permutações

Permutações	Bytes	Média	Permutações	Bytes	Média
4	52	13	7.776	34584	4
8	68	9	15.625	70296	4
16	100	6	16.384	70296	4
27	176	7	16.807	70296	4
32	176	6	32.768	142696	4
64	324	5	46.656	203252	4
125	540	4	65.536	289472	4
216	968	4	78.125	325680	4
256	1112	4	117.649	521784	4
343	1452	4	262.144	1057980	4
512	2140	4	279.936	1190248	4
625	2752	4	390.625	1694784	4
1.024	4516	4	823.543	3436000	4
1.296	5764	4	1.679.616	6965944	4
2.401	10528	4	2.097.152	8816320	4
3.125	13372	4	5.764.801	25448716	4
4.096	16968	4	16.777.216	73458252	4

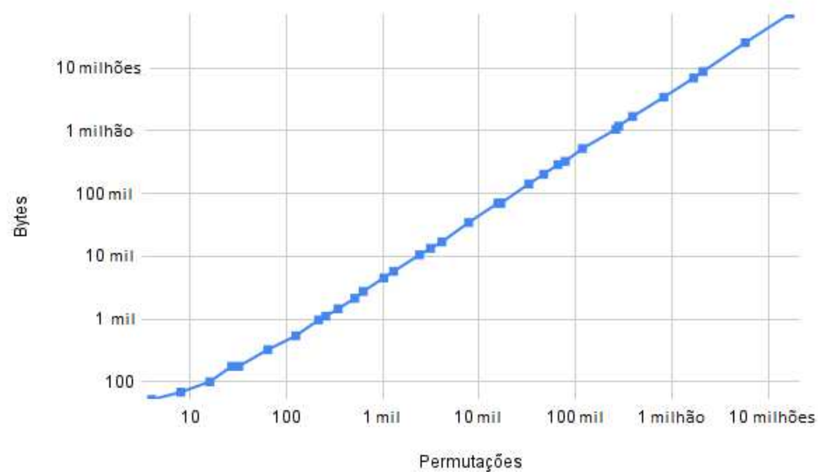
Fonte - O autor.

É possível perceber na tabela 1 que para poucas permutações, o cabeçalho de

memória ocupa espaço considerável. Isto deve-se ao fato de o cabeçalho de memória de uma lista vazia ocupar 36 bytes. Porém, de acordo com a forma que listas são implementadas em Python, o maior crescimento deve-se a adição de ponteiros para cada objeto, com cerca de 4 bytes cada na versão de 32 bits, e uma alocação adicional feita para manter o espaço amortizado constante. Assim, é possível observar que o crescimento do cabeçalho de memória se estabiliza em cerca de 4 bytes por objeto adicionado a medida em que o problema é escalado.

A figura 10 traz a curva de crescimento do cabeçalho de memória, em escala logarítimica, de uma lista de permutações de acordo com os dados apresentados acima.

Figura 10 – Crescimento do cabeçalho de memória de uma lista em relação a quantidade de permutações.



Fonte - O autor.

Da mesma forma, é possível estimar o consumo total de memória ao executar instâncias menores do problema e utilizar a média entre quantidade de itens de memória por espaço consumido.

Fez-se uso da biblioteca *pympler*, que aplica uma implementação recursiva da função *sys.getsizeof()*, para obter o consumo total de memória de cada objeto (BROUWERS et al., 2021).

De acordo com esta função, foram encontrados os valores a seguir:

Tabela 2 – Média de bytes por quantidade de permutações.

Permutações	Bytes	Média	Permutações	Bytes	Média
4	248	62	7.776	407.928	52
8	424	53	15.625	945.376	61
16	904	57	16.384	987.864	60
27	1.304	48	16.807	877.144	52
32	1.744	55	32.768	1.715.688	52
64	2.952	46	46.656	2.816.088	60
125	5.624	45	65.536	4.483.840	68
216	9.704	45	78.125	4.700.760	60
256	13.464	53	117.649	7.110.240	60
343	15.288	45	262.144	15.738.176	60
512	22.752	44	279.936	16.866.760	60
625	32.832	53	390.625	26.694.864	68
1.024	53.736	52	823.543	49.554.520	60
1.296	68.072	53	1.679.616	114.461.464	68
2.401	125.888	52	2.097.152	126.256.960	60
3.125	163.456	52	5.764.801	394.396.096	68
4.096	246.408	60			

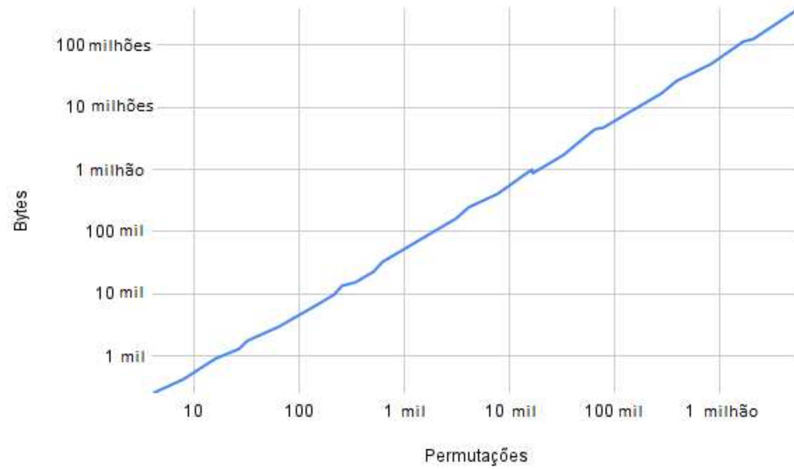
Fonte - O autor.

Na tabela 2, é possível perceber que o consumo em bytes de uma lista de tuplas $[i_1, i_2, ..., i_n]$ é, em média, $C^P \cdot 56$. Então, o cálculo final do consumo de memória máximo, em bytes, deste algoritmo será aproximadamente $L(40 + C^P \cdot 56)$, onde L representa a quantidade de listas utilizadas. Isso ocorre pois, em algum momento da execução, todas as três listas irão conter todas as permutações possíveis.

O motivo desta média não ser sempre igual deve-se ao Python alocar mais memória do que o necessário para uma lista, como explicado anteriormente, com a finalidade de manter um tempo amortizado constante durante a execução de operações sobre ela.

A figura 11 mostra a curva de crescimento total do consumo de memória, em escala logarítimica, de uma lista de permutações de acordo com os dados da tabela 2.

Figura 11 – Consumo de memória de uma lista de permutações, em escala logarítimica, pelo algoritmo de Knuth.



Fonte - O autor.

6 RESULTADOS

A execução do programa foi realizada com duas versões do Python distintas, de 32 e 64 bits, para entender como a linguagem Python se comporta em relação à alocação de memória entre elas.

6.1 VERSÃO DE 32-BIT

A tabela a seguir detalha a quantidade de memória consumida pelo cabeçalho de memória de acordo com a quantidade de cores e posições, medida com a função *sys.getsizeof* para uma lista de permutações.

Tabela 3 – Consumo de memória do cabeçalho de cada lista, em bytes, na versão de 32 bits

Posições/Cores	4	5	6	7	8
3	324	540	968	1452	2.140
4	1.112	2.752	5.764	10.528	16.968
5	4.516	13.372	34.584	70.296	142.696
6	16.968	70.296	203.252	521.784	~1 milhão
7	70.296	325.680	~1,2 milhão	~3,4 milhões	~8,8 milhões
8	289.472	~1,7 milhões	~7 milhões	~25,5 milhões	~73,5 milhões

Fonte - O autor.

De acordo com a tabela 3, é possível perceber que a linguagem Python alocou uma quantidade significativa de memória para fazer a gerência de seus objetos durante o experimento. Dessa forma, conclui-se que uma implementação utilizando lista de tuplas não é atrativa para escalar algoritmos que façam uso intensivo de objetos em memória. É possível notar também que a medida em que o problema é escalado, a média de bytes adicionado ao cabeçalho de memória é de cerca de 4 bytes, de acordo com as especificações do Python 32 bits.

Em seguida, a tabela 4 mostra o consumo total de memória máximo de uma lista, também de acordo com o número de cores e posições:

Tabela 4 – Consumo total de memória por cada lista, em bytes, na versão de 32 bits.

Posições/Cores	4	5	6	7	8
3	2.952	5.624	9.704	15.288	22.752
4	13.464	32.832	68.072	125.888	213.704
5	53.736	163.456	407.928	877.144	~1,7 milhão
6	246.408	945.376	~2,8 milhões	~7,1 milhões	~15,7 milhões
7	987.864	~4,7 milhões	~16,9 milhões	~49,6 milhões	~123,3 milhões
8	~4,5 milhões	~26,7 milhões	~114,5 milhões	~394,4 milhões	MemoryError

Fonte - O autor.

É importante notar que na tabela 4, o algoritmo apresentou um *MemoryError* com 8 cores e 8 posições. De acordo com a documentação do Python, isso acontece após o programa tentar alocar uma quantidade de memória maior do que o Python tem acesso. Neste caso, a versão usada de 32-bits só consegue alocar, no máximo, cerca de 4GB de RAM. De acordo com os cálculos realizados, o consumo de memória foi de aproximadamente 1 GB por lista.

6.2 VERSÃO DE 64-BIT

A tabela 5 traz o consumo de memória em bytes, do cabeçalho de memória de uma lista na versão de 64 bits do Python, também medida com a função *sys.getsizeof*.

Tabela 5 – Consumo de memória do cabeçalho de cada lista, em bytes, na versão de 64 bits.

Posições/Cores	4	5	6	7	8
3	568	1.080	1.912	2.872	4.216
4	2.200	5.432	11.288	20.536	33.048
5	8.856	26.040	67.224	136.632	277.336
6	33.048	136.632	394.968	~1 milhão	~2,3 milhões
7	136.632	632.824	~2,3 milhões	~6,7 milhões	~17,1 milhões
8	562.488	~3,3 milhões	~13,5 milhões	~49,4 milhões	~142,7 milhões

Fonte - O autor.

Comparado aos valores encontrados para o cabeçalho de memória na versão de 32 bits, o valor do consumo de memória quase dobra na versão de 64 bits, conforme tabela 5. Isso deve-se ao fato de o tamanho do um ponteiro passar de 4 para 8 bytes, sendo estes os principais componentes do cabeçalho de memória da lista.

Tabela 6 – Consumo total de memória por cada lista, em bytes, na versão de 64 bits

Posições/Cores	4	5	6	7	8
3	4.792	9.240	15.928	25.048	37.240
4	20.760	50.592	104.792	193.632	328.216
5	90.904	276.200	689.496	~1,5 milhão	~2,9 milhões
6	393.624	~1,5 milhão	~4,5 milhões	~11,4 milhões	~25,3 milhões
7	~1,7 milhão	~8,1 milhões	~29,2 milhões	~85,7 milhões	~218,5 milhões
8	~7,4 milhões	~44,9 milhões	~188,2 milhões	~648,9 milhões	~1,9 bilhão

Fonte - O autor.

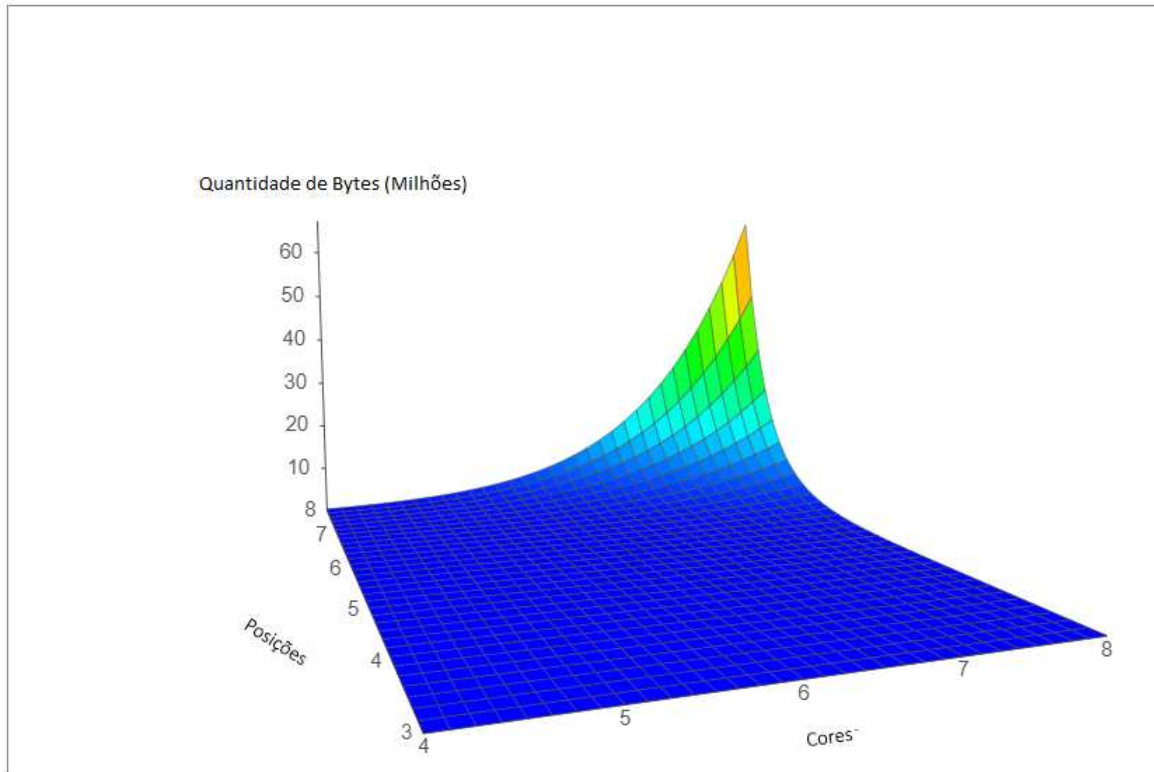
Na tabela 6, é possível perceber a mesma tendência de crescimento que o cabeçalho de memória entre as versões de 32 e 64 bits. Os valores encontrados são praticamente o dobro dos valores da tabela 4. Após investigação, foi medido que um inteiro na versão de 32 bits ocupa 12 bytes de memória, e este passa a ocupar 24 bytes na versão de 64 bits. Como a lista é composta de tuplas de inteiros, e o valor dos ponteiros de referências destas também dobram de tamanho, é fácil perceber o motivo destes valores tendo sido obtidos.

6.3 COMPARAÇÃO COM RESULTADOS ESPERADOS

Percebe-se que embora o crescimento da curva do consumo de memória tenha se mantido exponencial, como esperado, a conjectura inicial de que o tamanho do cabeçalho deixa de impactar significativamente a utilização total de memória a medida em que o problema é escalado é falsa. Foi constatado que o cabeçalho de memória representa cerca de 7% da utilização de toda a memória de um objeto.

A figura a seguir mostra a curva do crescimento do cabeçalho da lista S, conforme registrado na tabela 3.

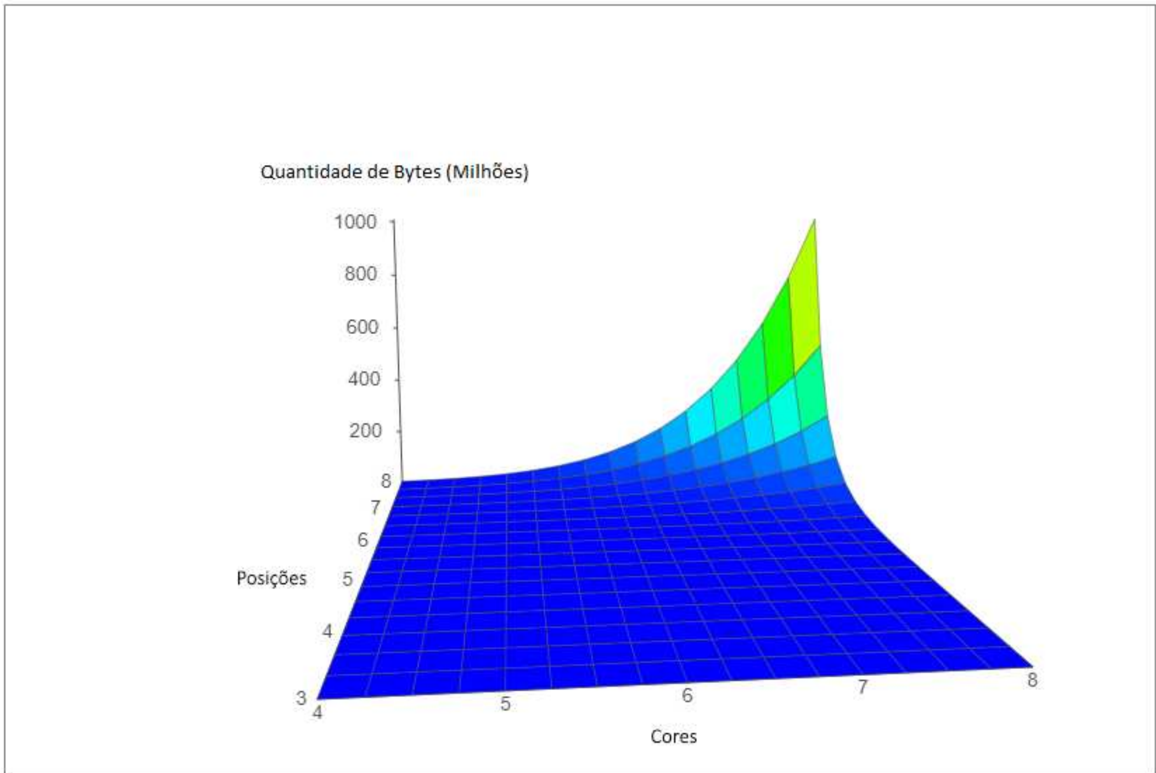
Figura 12 – Curva de consumo de memória do cabeçalho de uma lista pelo algoritmo de Knuth, na versão 32 bits.



Fonte - O autor.

De forma similar, a figura abaixo ilustra a curva de crescimento do consumo de memória geral da lista *S*, conforme apresentado na tabela 4. É possível observar o alto consumo de memória registrado por uma lista de tuplas durante a execução do programa. É possível perceber que a medida em que o problema é escalado a curva tende a se estabilizar em cerca de 60 bytes por permutação, uma diferença de cerca de 4 bytes a mais da média calculada no estudo teórico. Essa variação tem um impacto significativo no consumo total de memória, visto que com 8 cores e 8 posições, temos mais de 16 milhões de permutações.

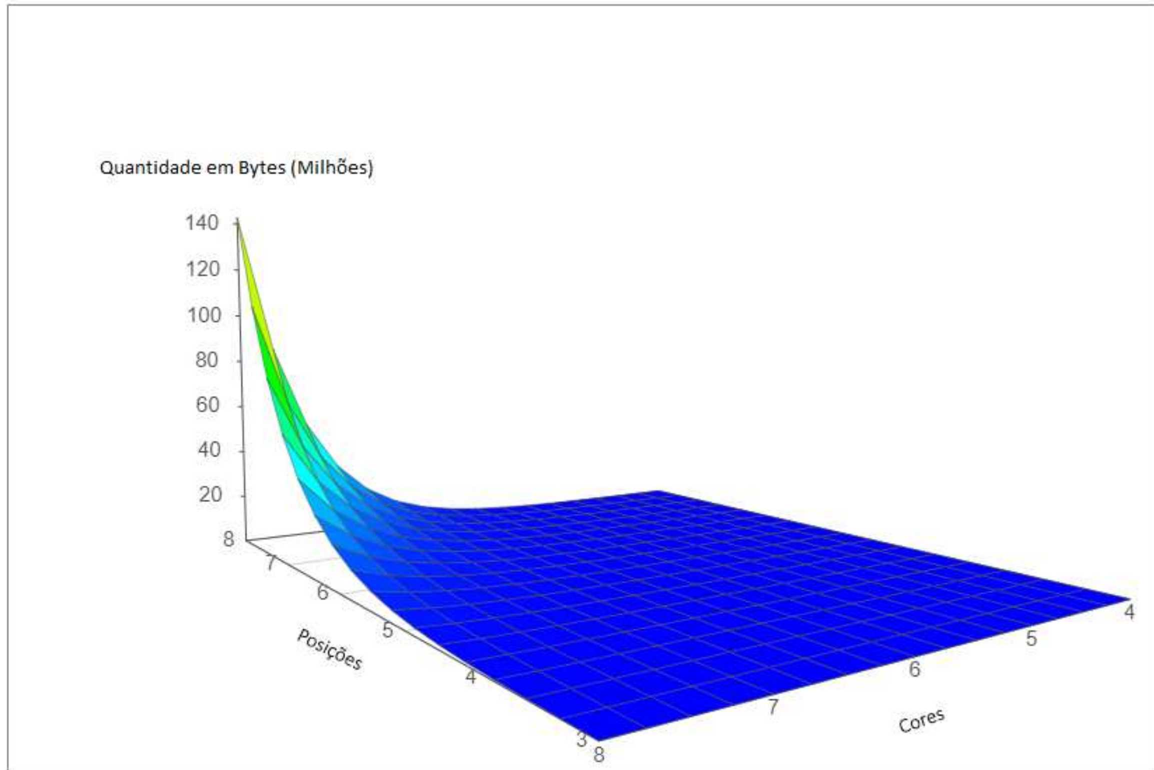
Figura 13 – Curva de consumo de memória total de uma lista pelo algoritmo de Knuth, na versão de 32 bits.



Fonte - O autor.

Já em relação a versão de 64 bits, a curva de crescimento do cabeçalho de memória é aproximadamente o dobro da curva de crescimento da versão de 32 bits, como ilustra a figura 14.

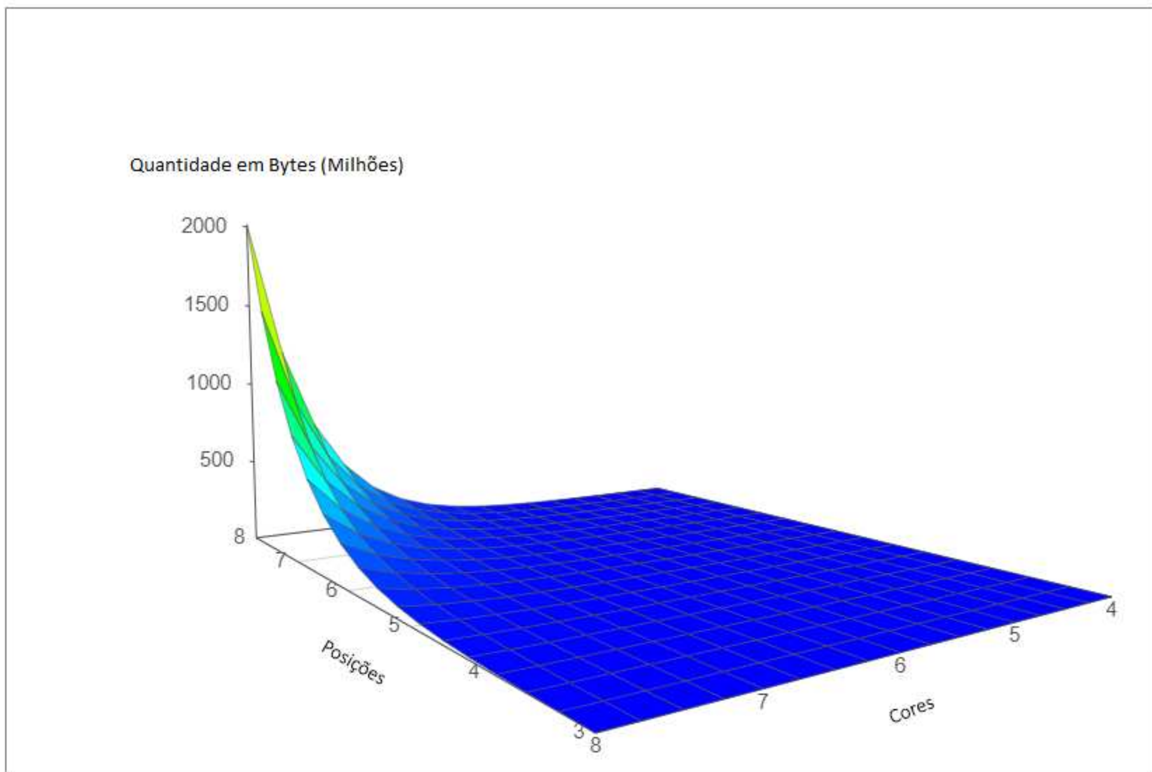
Figura 14 – Curva de consumo de memória do cabeçalho de uma lista pelo algoritmo de Knuth, na versão 64 bits.



Fonte - O autor.

Já a figura 15 traz a curva do consumo total de memória de uma lista de permutações, de acordo com os dados apresentados na tabela 6. Novamente é possível perceber que o consumo total de memória praticamente dobra em relação à curva de memória de 32 bits na figura 13.

Figura 15 – Curva de consumo de memória total de uma lista pelo algoritmo de Knuth, na versão de 64 bits.



Fonte - O autor.

7 CONSIDERAÇÕES FINAIS

Após o desenvolvimento deste trabalho, podemos perceber a importância que o jogo de Mastermind, superficialmente apenas um jogo inocente de tabuleiro, possui para a área de computação e afins, e que o estudo deste problema NP-Completo pode trazer grandes avanços em aplicações práticas.

Percebe-se que apesar da grande quantidade de artigos disponíveis sobre o tema, ainda há uma grande carência em relação à estudos quanto a escalabilidade destes algoritmos, principalmente no que diz respeito ao uso de memória.

Fica claro também que, embora seja uma linguagem de fácil programação, devido a forma que a memória é alocada, o alto consumo de memória pelo cabeçalho dos objetos pode criar um empecilho para máquinas de configurações mais modestas. Dessa forma, é salientada a significância deste estudo em determinar como a linguagem Python se comporta em relação à alocação de memória, auxiliando outros possíveis estudos utilizando a mesma.

É importante ressaltar que devido a limitações das bibliotecas Python utilizadas, o estudo teórico realizado sobre o algoritmo de Knuth não se aprofundou na forma de como o Python faz a gerência de memória utilizando arenas, pools e blocos. É importante um estudo mais detalhado dessas estruturas de memória e sobre a forma como elas se comportam.

Por fim, existe também a necessidade de se realizar em trabalhos futuros, um estudo mais aprofundado em relação ao consumo de memória do Python, principalmente utilizando outras estruturas de dados, para entender se o elevado consumo de memória do cabeçalho na linguagem Python é algo inerente de uma lista de tuplas ou de todos os objetos.

REFERÊNCIAS

BERGHMAN, L.; GOOSSENS, D.; LEUS, R. Efficient solutions for mastermind using genetic algorithms. **Computers and Operations Research**, v. 36, 2009.

BLAIR, N.; KARNATI, A.; LEE, D.; LEE, J. Mastering mastermind with mcmcs. **EECS 126 - Probability and Random Processes**, 2018.

BROUWERS, J.; HAEHNE, L.; SCHUPPENIES, R. **pympler.asizeof**. 2021. <https://pympler.readthedocs.io/en/latest/library/asizeof.html>.

BV, T. S. **TIOBE Index for November 2021**. 2021. <https://www.tiobe.com/tiobe-index/>.

DOWELL, J. **Defeating Mastermind**. 2009. <http://mercury.webster.edu/aleshunas/Support%20Materials/Analysis/Dowelll%20-%20Mastermind%20v2-0.pdf>.

ERIKY. **GlobalInterpreterLockor**. 2020. <https://wiki.python.org/moin/GlobalInterpreterLock>.

FOCARDI, R.; LUCCIO, F. Cracking bank pins by playing mastermind. **Fun with Algorithms, Vol. 6099 of Lecture Notes in Computer Science**, 2010.

FOUNDATION, P. S. **Memory Management**. 2021. <https://docs.python.org/3/c-api/memory.html>.

FRANCIS, J. **Strategies for playing MOO, or "Bulls and Cows"**. 2010. <http://slovesnov.users.sourceforge.net/bullscows/bulls_and_cows.pdf>.

GOODRICH, M. T. The mastermind attack on genomic data. **2009 30th IEEE Symposium on Security and Privacy**, 2009.

GUERVóS, J. J. M.; CASTILLO; MORA, A.; COTTA, C. An experimental study of exhaustive solutions for the mastermind puzzle. **.**, 2012.

GUERVóS, J. J. M.; CASTILLO; MORA, A.; COTTA, C. A search for scalable evolutionary solutions to the game of mastermind. **IEEE Congress**, 2013.

INVICTA Toys and Games. 2007. <https://web.archive.org/web/20070812104420/http://dspace.dial.pipex.com/town/road/gbd76/toys.htm>.

IRVING, R. W. Towards an optimum mastermind strategy. **Journal of Recreational Mathematics 11**, 1979.

KNUTH, D. E. The computer as a master mind. **Journal of Recreational Mathematics, 9:1–6**, 1976.

KOOI, B. Yet another mastermind strategy. **ICGA Journal, 28, no. 1, 13–20**, 2005.

KOYOMA, K.; LAI, W. An optimal mastermind strategy. **Journal of Recreational Mathematics, no. 25, pp. 251–256**, 1994.

NEUWIRTH, E. Some strategies for mastermind. **Zeitschrift far Operations Research, Band 26**, 1981.

PYTHON; SOFTWARE; FOUNDATION. **Built-in Types**. 2021. <https://docs.python.org/3/library/stdtypes.html>.

PYTHON; SOFTWARE; FOUNDATION. **Common Object Structures**. 2021. <https://docs.python.org/3/c-api/structures.html>.

PYTHON; SOFTWARE; FOUNDATION. **Data Structures**. 2021. <https://docs.python.org/3/tutorial/datastructures.html>.

PYTHON; SOFTWARE; FOUNDATION. **Sys**. 2021. <https://docs.python.org/3/library/sys.html>.

RASHED, G.; AHSAN, R. Python in computational science: Applications and possibilities. **International Journal of Computer Applications**, v. 46, 2012.

ROCKYB. **PythonImplementations**. 2020. <https://wiki.python.org/moin/PythonImplementations>.

RYBÁR. Evolutionary solution of the game mastermind. **Thesis**, Slovak University of Technology in Bratislava, 2014.

SALCEDO-SANZ, S.; GUERVóS, J. M. New solver and optimal anticipation strategies design based on evolutionary computation for the game of mastermind. **Evolutionary Intelligence 6**, v. 4, 2014.

SALGADO, P. G. **Design of CPython's Garbage Collector**. 2021. <https://devguide.python.org/garbage_collector/>.

SHAPIRO, E. Playing mastermind logically. **ACM SIGART Bulletin.**, 1983.

STEINER, T. **Mastermind board after 2 moves**. 2006. <https://upload.wikimedia.org/wikipedia/commons/f/f5/Mastermind_beispiel.svg>.

STUCKMAN, J.; ZHANG, G.-Q. Mastermind is np-complete. **INFOCOMP Journal of Computer Science**, v. 5, p. 25–28, 2006.

ZEHRA, F.; DARAKHSHAN; JAVED, M.; PASHA, M. Comparative analysis of c++ and python in terms of memory and time. **Preprints**, 2020.

APÊNDICES

APÊNDICE A – Código da implementação do algoritmo de Knuth

```
from random import randint
from collections import Counter
from itertools import product

#Escalabilidade
QTD_POSICOES = 4
CORES = [1,2,3,4,5,6]

#Listas Globais
combinacao=(1,1,2,2)
listaCombinacao =[combinacao]
codigoSecreto = (randint(1,len(CORES)),randint(1,len(CORES)),randint(1,len(CORES)),randint(1,len(CORES)))
S = [tuple(x) for x in product(range(1,len(CORES)+1), repeat=QTD_POSICOES)]
codigo = [tuple(x) for x in product(range(1,len(CORES)+1), repeat=QTD_POSICOES)]
resultados = [[certo, errado] for certo in range(5) for errado in range(5 - certo) if not (certo == 3 and errado == 1)]

def calculaFeedback(combinacao, segredo):
    '''
    Essa funcao compara duas sequencias e retorna a quantidade de numeros iguais
    na mesma posiÃğÃčo (pretos) e numeros iguais em posicoes diferentes (brancos).
    '''

```

```
23      brancos = sum((Counter(segredo) & Counter(combinacao)).
           values())
24      pretos = sum(c == g for c, g in zip(segredo, combinacao
           ))
25      return [pretos, brancos - pretos]
26
27
28  def minMax():
29
30      '''
31      Calcula a proxima jogada de acordo com a estrategia de
           minimax.
32      '''
33
34      global combinacao
35      global S
36
37      cScore = []*len(codigo)
38      for item in codigo:
39          if item not in listaCombinacao:
40              hitCount = [0]*len(resultados)
41              for s in S:
42                  hitCount[resultados.index(calculaFeedback(s
                       , item))] += 1
43              # calcula o score para a as combinaÃğÃţes nÃčo
                   utilizadas
44              cScore.append(len(S)-max(hitCount))
45          else:
46              cScore.append(0)
47      # pega todos os indices com o score mÃąximo
48      maxScore = max(cScore)
```

```
49      indices = [i for i, x in enumerate(cScore) if x ==
            maxScore]
50      # se alguma combinaÃğÃčo corresponde correspondente aos
             indices pertence a S, a usa como proxima jogada
51      change = False
52      for i in range(len(indices)):
53          if codigo[indices[i]] in S:
54              combinacao = codigo[indices[i]]
55              change = True
56              break
57      # se nao, uma a menor combinacao como proxima jogada
58      if change == False:
59          combinacao = codigo[indices[0]]
60      listaCombinacao.append(combinacao)
61
62  def main():
63
64      attempts = 1
65      feedback = calculaFeedback(combinacao,codigoSecreto)
66      global S
67
68
69      print('CÃşdigo Secreto = '+str(codigoSecreto))
70      print('\n')
71      print('combinaÃğÃčo = '+str(combinacao))
72
73      while feedback[0] != QTD_POSICOES:
74
75          aux = S.copy()
76          attempts += 1
77          for i in range (0,len(S)):
```

```
78              if calculaFeedback(S[i],combinacao) != feedback
                    :
79                  aux.remove(S[i])
80          S = aux
81
82          minMax()
83
84          feedback = calculaFeedback(combinacao,codigoSecreto
               )
85          print('combinaÃğÃčo = '+str(combinacao))
86
87      print('\n')
88      print('tentativas = ' +str(attempts))
89
90
91  if __name__ == "__main__":
92      main()
```