Uma imagem com texto

Descrição gerada automaticamente

Universidade de Coimbra

Faculdade de Ciências e Tecnologia

Departamento de Engenharia Informática

Integração de Sistemas

**Trabalho prático Nº1**

Diogo Jordão Filipe - uc2018288391@student.uc.pt

José Miguel Silva Gomes - uc2018286225@student.uc.pt

Coimbra, 8 de outubro de 2021

Índice

[1 Introdução 3](#_Toc84436599)

[2 JSON 3](#_Toc84436600)

[3 MessagePack 3](#_Toc84436601)

[4 JSON vs MessagePack 3](#_Toc84436602)

[4.1 Estruturas de dados utilizadas 4](#_Toc84436603)

[4.2 Descrição do problema 5](#_Toc84436604)

[4.3 Condições experimentais 5](#_Toc84436605)

[4.4 Resultados experimentais 6](#_Toc84436606)

[5 Análise do código fonte 11](#_Toc84436607)

[6 Conclusão 13](#_Toc84436608)

# Introdução

A representação de informação é algo importante no mundo da informática e, como tal, existem diversas maneiras de a representar nomeadamente em formato de texto e formato binário.

Este documento visa explicar a diferença entre dois formatos de representação de informação: JSON *vs* MessagePack.

# JSON

O JSON é um formato simples e rápido de serialização de dados para texto, derivado do JavaScript, que funciona via pares de nomes/valores e utiliza linguagem legível por humanos. É frequentemente usado para enviar dados do servidor ao cliente numa aplicação web.

# MessagePack

O MessagePack é um formato eficiente de serialização binária, que também permite a transmissão de dados entre sistemas como o JSON. Este formato codifica inteiros de pequenas dimensões para apenas um byte, e texto, também de pequenas dimensões, é codificado com um byte mais o tamanho em si.

# JSON *vs* MessagePack

De forma a proceder a uma comparação entre estes dois formatos de representação, seguimos a estrutura *Pet-Owner* *many-to-one relationship*. De seguida foram elaboradas as diferentes estruturas de dados.

## Estruturas de dados utilizadas

Para o problema acima descrito, foram criadas duas classes (*Pet, Owner*) da seguinte maneira:

|  |  |  |  |
| --- | --- | --- | --- |
| **1** | **Class** Owner: | | |
| **2** |  | **def** \_init\_\_(id, name, birth, phone, address) | |
| **3** |  |  | self.id = id |
| **4** |  |  | self.name = name |
| **5** |  |  | self.birth = birth |
| **6** |  |  | self.phone = phone |
| **7** |  |  | self.address = address |

|  |  |  |  |
| --- | --- | --- | --- |
| **1** | **Class** Pet: | | |
| **2** |  | **def** \_init\_\_(id, name, species, gender, weight, birth, description, owner) | |
| **3** |  |  | self.id = id |
| **4** |  |  | self.name = name |
| **5** |  |  | self. species = species |
| **6** |  |  | self. gender = gender |
| **7** |  |  | self. weight = weight |
| **8** |  |  | self. birth = birth |
| **9** |  |  | self. description = description |
| **10** |  |  | self. owner = owner |

Estas classes foram criadas tendo em conta o enunciado disponibilizado e, como se trata de uma relação *many-to-one*, o objeto *Pet* tem um identificador para o seu *Owner* de forma poder ser identificado.

## Descrição do problema

De forma a proceder a uma comparação entre os dois formatos de dados (JSON e MessagePack), procedeu-se à criação de métodos para a geração de um grande volume de dados (*Pets* e *Owners*), funções para serializar e desserializar informação para cada um dos formatos, contagem do tempo de cada formato de dados para cada operação (serialização e desserialização) e tamanho do ficheiro de cada formato após se elaborar a serialização dos dados.

## Condições experimentais

IDE: *PyCharm* 2021

Linguagem de Programação: Python 3.9

Especificações do Computador de José Miguel

Processador: 2,6 GHz Intel Core i7

Placa Gráfica: AMD Radeon Pro 5300M 4 GB

Memória: 16 GB 2667 MHz DDR4

Disco: 512GB SSD

Sistema Operativo: MacOS BigSur

Especificações do Computador de Diogo Filipe

Processador: Intel Core i7-7500U CPU 2.70-2.90 GHz

Placa Gráfica: AMD Radeon R7 M340

Memória: 8 GB

Disco: Samsung SSD 860 EVO 500GB

Sistema Operativo: Windows 10 Home

## Resultados experimentais

De modo a estabelecer uma comparação entre os dois formatos de dados foram realizadas experiências tendo por base os seguintes princípios:

* Número de dados gerados;
* Média do tempo ocorrido no processo de serialização/desserialização;
* Desvio padrão;
* Tamanho do ficheiro gerado após o processo de serialização para os dois formatos;

Figura 1 - Tabela com os dados gerados pelo aluno José Miguel

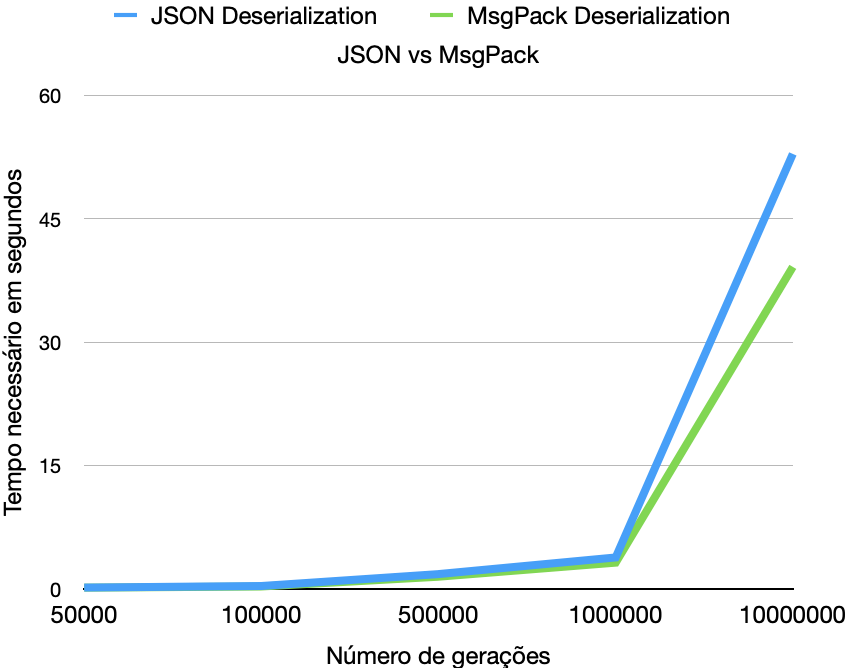
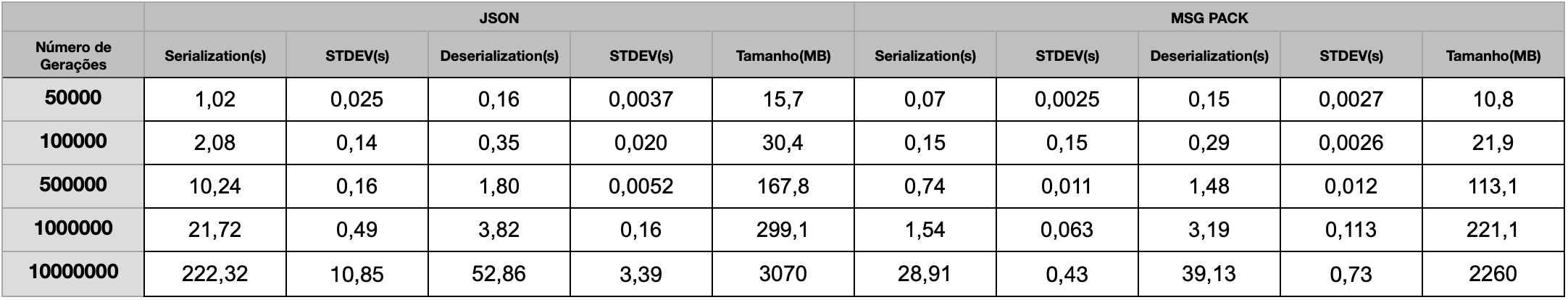


Figura 2 - Diferença entre o tempo de desserialização do formato JSON e MsgPack

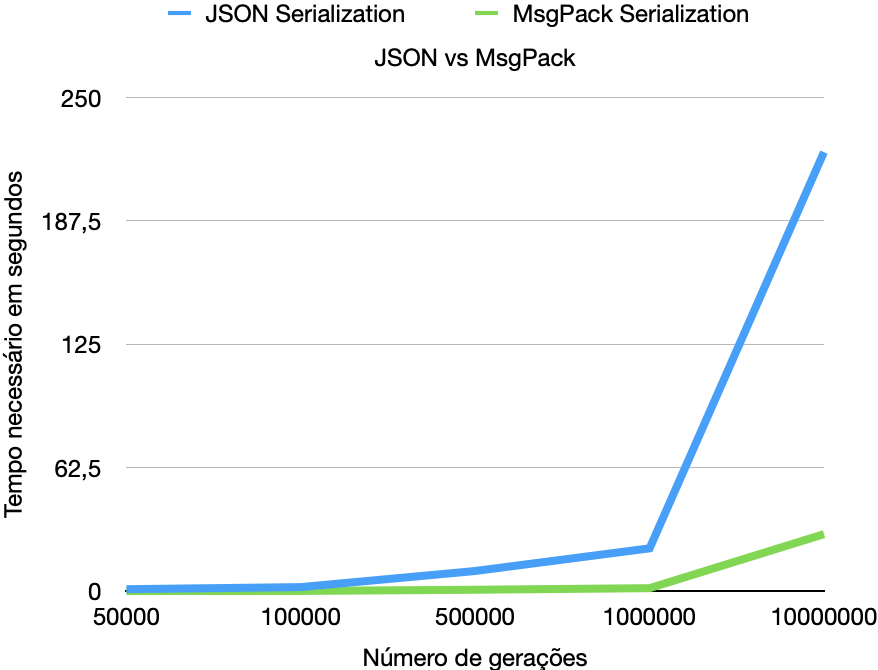


Figura 3 - Diferença entre o tempo de serialização do formato JSON e MsgPack

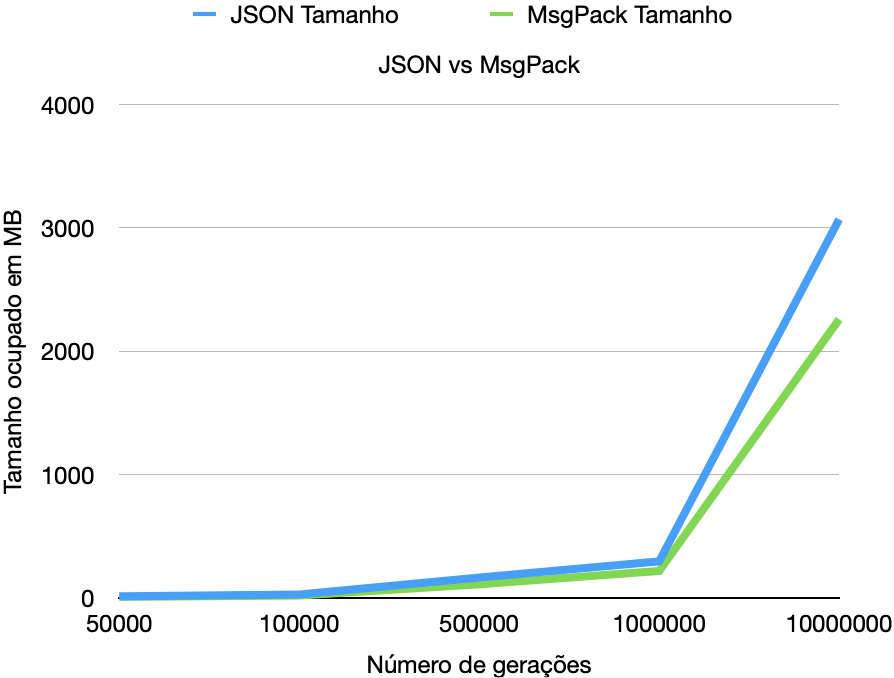


Figura 4 - Diferença do tamanho ocupado em disco pelos ficheiros serializados no formato JSON e MsgPack

Uma imagem com texto

Descrição gerada automaticamente

Figura 5 - Tabela com os dados gerados pelo aluno Diogo Filipe

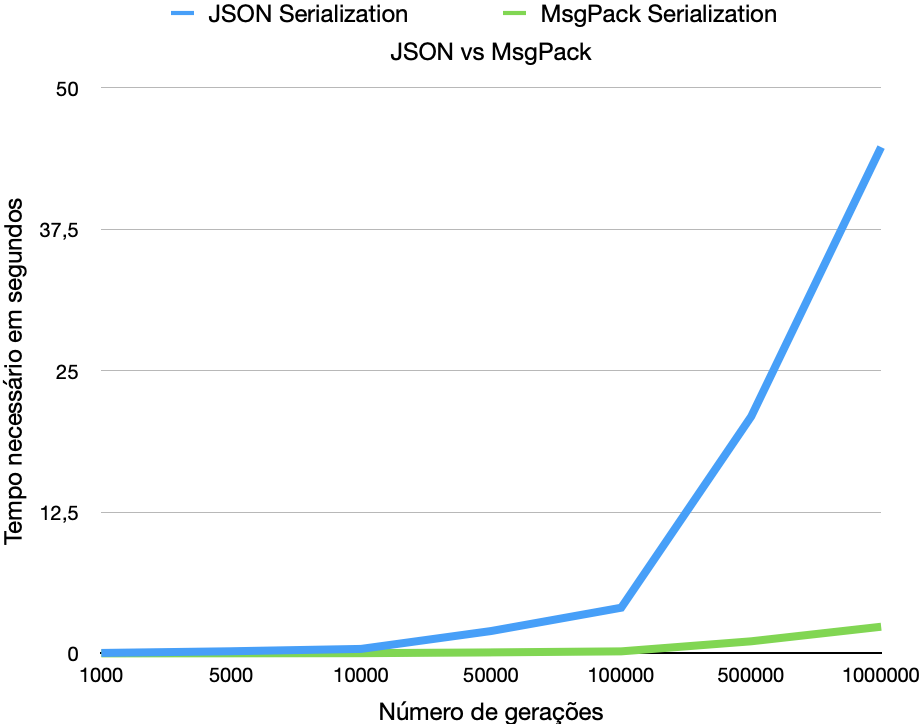


Figura 7 - Diferença entre o tempo de serialização do formato JSON e MsgPack

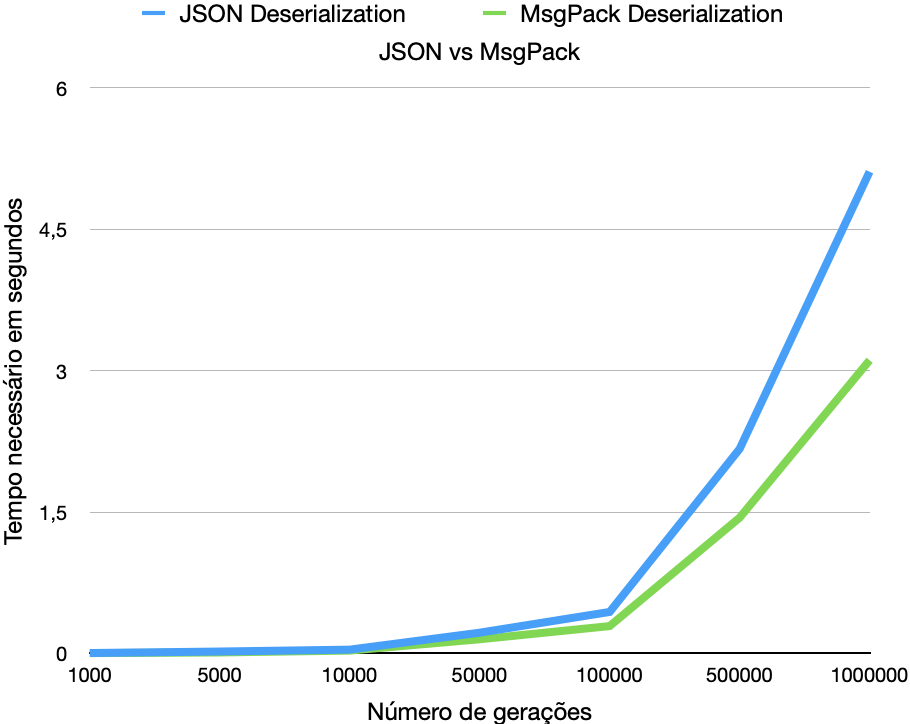


Figura 6 - Diferença entre o tempo de desserialização do formato JSON e MsgPack

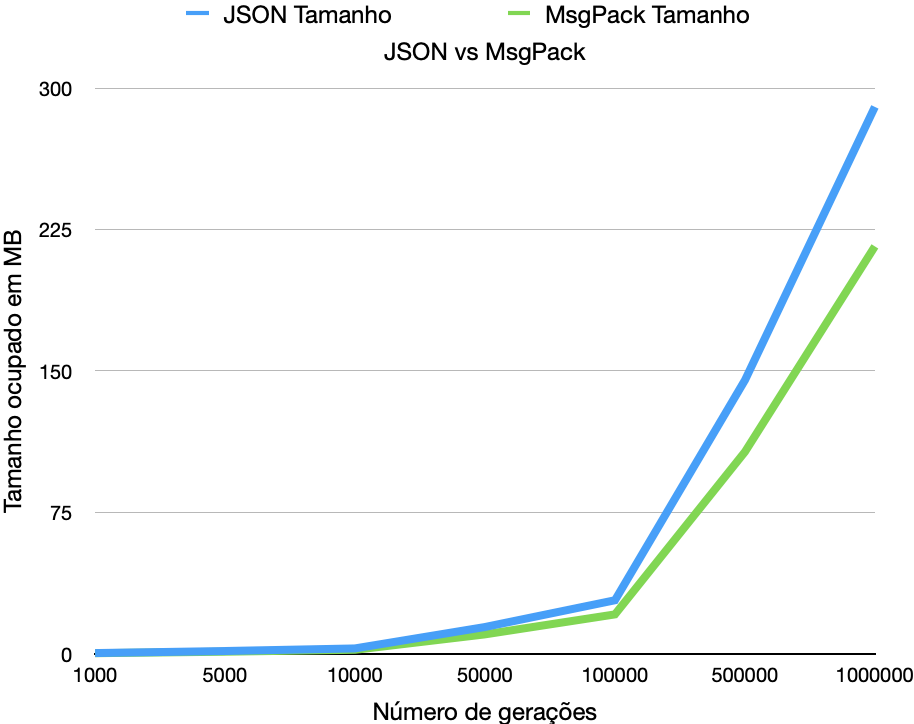


Figura 8 - Diferença do tamanho ocupado em disco pelos ficheiros serializados no formato JSON e MsgPack

# Análise do código fonte

Neste projeto, começámos por elaborar as classes *Pet* e *Owner* descritas na secção **4.1**. De seguida, começámos por elaborar o processo de serialização do *JSON* através da função criada nas linhas 32-67. Nesta função são criados dois ficheiros de texto “json\_time\_s.txt” e “json\_time\_d.txt” que servirão para guardar os tempos de serialização e de desserialização, respetivamente, durante as várias repetições dessas mesmas operações de modo a calcular a média e o desvio padrão desse processo.

Seguidamente, cria-se outro ficheiro de texto (“json\_serialized.txt”) que guardará a informação *JSON* serializada de forma a obter o espaço que ocupa em disco. Através do método **json.dump(data, f, cls=Encoder**), é serializado o objeto “*data*” para um *file* *pointer* **f** usando como *Enconder* a função especificada no parâmetro **cls** e, através do método **json.load(f)** ocorre a desserialização do *file pointer* **f** que contém um documento JSON para um objeto Python.

Através das variáveis *“startS”, “endS”, “startD”, “endD”,* é possível determinar o tempo que demorou a fazer o processo de serialização (linha 49) e de desserialização (linha 50). Finalmente, após as várias medições, são calculadas a média e o desvio padrão das execuções e escritas para o ficheiro de texto (linhas 61-64).

De igual modo, para o formato MessagePack, são criados dois ficheiros (“msgpack\_time\_s.txt” e “msgpack\_time\_d.txt”) de forma a guardar os valores de cada execução para posterior análise. É também criado um ficheiro de texto “msgpack\_serialized.txt” de forma a guardar a informação serializada do MessagePack. Dado que como o MessagePack se trata de um processo de serialização binária, o seu ficheiro de texto será aberto no modo ‘wb’ ao invés de ‘w’ como no formato JSON.

Por recurso à função **msgpack.pack(data, f, default=encoder\_msgpack)** (linha 78), é possível serializar a informação descrita por “data” e escrevê-la no ficheiro dado pelo *file pointer* **f** e, fazendo uso da função **msgpack.unpack(f)** (linha83) é possível desserializar a informação que está contida no ficheiro dado por **f**. De forma a calcular o tempo, este segue o mesmo princípio que o JSON: fazendo uso das variáveis *“startS”, “endS”, “startD”, “endD”* (linhas 86-87) é possível calcular o tempo decorrido em cada operação e proceder ao cálculo da sua média e desvio padrão tendo em conta o número de repetições da experiência

De modo a gerar dados de forma massiva, foram implementadas as funções “gen\_owners” e “gen\_pets” (linhas 111 e 117) que, tendo em conta o valor de **n** (número de gerações), este cria n *Pets* e n *Owners*. Estas duas funções estão englobadas no método **gen\_data(n)** que retorna os dados criados nas duas funções acima descritas.

Finalmente, na linha 136 podemos gerar uma grande quantidade de dados especificando um valor para o argumento da função e também denominamos um valor para o número de repetições de forma a reduzir o impacto de componentes aleatórias na performance do programa.

# Conclusão