FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

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

A representação de informação digital 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 comparar e explorar as diferenças entre dois formatos de representação de informação, JSON e MessagePack, de acordo com alguns parâmetros: complexidade de programação, tamanho de serialização, e velocidades de serialização e desserialização.

1.1 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 em aplicações web.

1.2 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 inicial.

2 Ambiente da experiência

Nesta secção descreveremos as condições do ambiente da experiência realizada, como as estruturas de dados e ferramentas utilizadas, e as especificações dos computadores que executaram o código e recolheram os dados.

2.1 Estruturas de dados

De forma a proceder a uma comparação entre estes dois formatos de representação de dados, desenvolvemos uma estrutura *Pet-Owner* com *many-to-one relationship*, que é composta por duas entidades *Pet* e *Owner*, que contém atributos básicos como identificadores, nomes, datas de nascimento, etc.

Foram então as criadas as duas classes da seguinte forma:

```
6   Class Owner:
7       def __init__(id, name, birth, phone, address)
8           self.id = id
9           self.name = name
10          self.birth = birth
11          self.phone = phone
12          self.address = address
13
15  Class Pet:
16      def __init__(id, name, species, gender, weight, birth, description, owner)
17          self.id = id
```

```
18          self.name = name
19          self. species = species
20          self. gender = gender
21          self. weight = weight
22          self. birth = birth
23          self. description = description
24          self. owner = owner
```

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

2.2 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

3 Código fonte

Nesta secção iremos explicar as diferentes partes do código desenvolvido, como a geração de dados, a serialização e desserialização dos dados, e a medição do tempo das mesmas.

Para reduzir o impacto de componentes aleatórias da performance dos computadores, as operações a estudar e a sua cronometração foram parametrizadas com o número de repetições a serem feitas.

3.1 Geração de dados

Para automatizar e parametrizar a geração de dados foram implementadas 3 funções: “gen_owners”, “gen_pets” e “gen_data”.

```
111 def gen_owners(n, owners):
112     for i in range (1, n + 1):
113         name = "Owner " + str(i)
114         owners.append(Owner(i, name, "19/10/2000", "912345678", "Coimbra"))
115
117 def gen_pets(n, pets):
118     for i in range(1, n + 1):
119         name = "Pet" + str(i)
120         species = "Species " + str(i)
121         description = name + " that belongs to " + species
122         if i % 2 == 0:
123             pets.append(Pet(i, name, species, "Male", 8, "10/04/2005", description, i))
124         else:
125             pets.append(Pet(i, name, species, "Female", 4, "10/04/2005", description, i))
126
128 def gen_data(n):
129     data = [ ]
130     gen_owners(n, data)
131     gen_pets(n, data)
132     return data
```

Figura 1 - Funções de geração de dados

A partir da chamada da função "gen_data" com o número desejado de objetos de cada classe a serem gerados ("n"), são chamadas as funções "gen_owners" e "gen_pets", que os geram respetivamente, com parâmetros simplificados e por vezes iguais, visto que o foco da experiência é a comparação dos dois formatos e não dos dados em si.

3.2 Serialização e desserialização

JSON:

Para realizar a serialização dos dados com o formato *JSON* foi importada a biblioteca "*json*", e usadas as suas funções de serialização e desserialização.

Primeiramente, foi necessário criar uma classe para substituir a que mapeia a codificação dos tipos de objetos da função de serialização "*dump*", pois o tipo de objetos a ser utilizado não é serializável.

```
28 class Encoder(json.JSONEncoder):
29     def default(self, o):
30         return o.__dict__
```

Figura 2 - Classe de codificação do formato JSON

Enviando então esta classe como parâmetro "*cls*" na função "*dump*" é possível serializar os objetos pois estes são transformados em dicionários do *Python* antes da serialização.

Para armazenar os dados serializados é criado um ficheiro "json_serialized.txt" e enviado o seu ponteiro como parâmetro na função "*dump*".

```
41 json.dump(data, f, cls=Encoder)
```

Figura 3 - Utilização da função dump da biblioteca json

Para a desserialização, apenas é chamada a função "*load*" com o ponteiro do ficheiro mencionado acima como parâmetro.

```
46 json.load(f)
```

Figura 4 - Utilização da função load como forma de desserialização

MessagePack:

O código desenvolvido para o formato *MessagePack* foi bastante semelhante ao do *JSON*, mas com a biblioteca "*msgpack*".

Para contornar a impossibilidade de serializar os objetos com a função "pack", foi criada a função "*encoder_msgpack*" e enviada como parâmetro "*default*" para a função "pack", cumprindo o mesmo fim que a classe "*Encoder*" criada para o *JSON*.

```
107  def encoder_msgpack(o):
108      return o.__dict__
```

Figura 5 - Função de Encode do formato MessagePack

Também é criado um ficheiro para guardar os dados serializados, "*msgpack_serialized.txt*", e enviado o seu ponteiro como parâmetro na função "*pack*".

```
78  msgpack.pack(data, f, default=encoder_msgpack)
```

Figura 6 - Método de serialização do formato MessagePack

A desserialização é igual à do JSON, mas com a função "*unpack*".

```
78  msgpack.unpack(f)
```

Figura 7 - Método de desserialização do formato MessagePack

3.3 Medição do tempo

Foram importadas duas bibliotecas, a "*time*" para medir o tempo, e a "*statistics*" para calcular a média dos tempos e o seu desvio padrão.

Em ambos os formatos usou-se o mesmo código para realizar o mencionado. Primeiramente criam-se dois ficheiros, um para guardar os tempos de serialização ("*json_time_s.txt*" / "*msgpack_time_s.txt*"), e outro para os de desserialização ("json_time_d.txt" / "msgpack_time_d.txt"). É então usada a função "*perf_counter*", e guardado o seu valor, antes e depois da serialização e desserialização, e a sua diferença (depois - antes) corresponde ao tempo em segundos que demorou a operação.

```
77  startS = time.perf_counter()
78  msgpack.pack(data, f, default=encoder_msgpack)
79  endS = time.perf_counter()
```

Figura 8 - Exemplo de contagem do tempo

A cada execução é escrito no ficheiro correspondente o tempo demorado, e guardado numa lista. No fim da última execução, temos duas listas (uma da serialização e outra da desserialização) com os tempos obtidos, e é usada a função "*mean*" para obter a média de cada lista, e a função "*pstdev*" para calcular o desvio padrão. Estes dois últimos valores também são escritos no ficheiro correspondente.

4 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;

	JSON					MSG PACK				
Número de Gerações	Serialization(s)	STDEV(s)	Deserialization(s)	STDEV(s)	Tamanho(MB)	Serialization(s)	STDEV(s)	Deserialization(s)	STDEV(s)	Tamanho(MB)
50000	1,02	0,025	0,16	0,0037	15,7	0,07	0,0025	0,15	0,0027	10,8
100000	2,08	0,14	0,35	0,020	30,4	0,15	0,15	0,29	0,0026	21,9
500000	10,24	0,16	1,80	0,0052	167,8	0,74	0,011	1,48	0,012	113,1
1000000	21,72	0,49	3,82	0,16	299,1	1,54	0,063	3,19	0,113	221,1
10000000	222,32	10,85	52,86	3,39	3070	28,91	0,43	39,13	0,73	2260

Figura 9 - Tabela com os dados gerados pelo computador de José Miguel

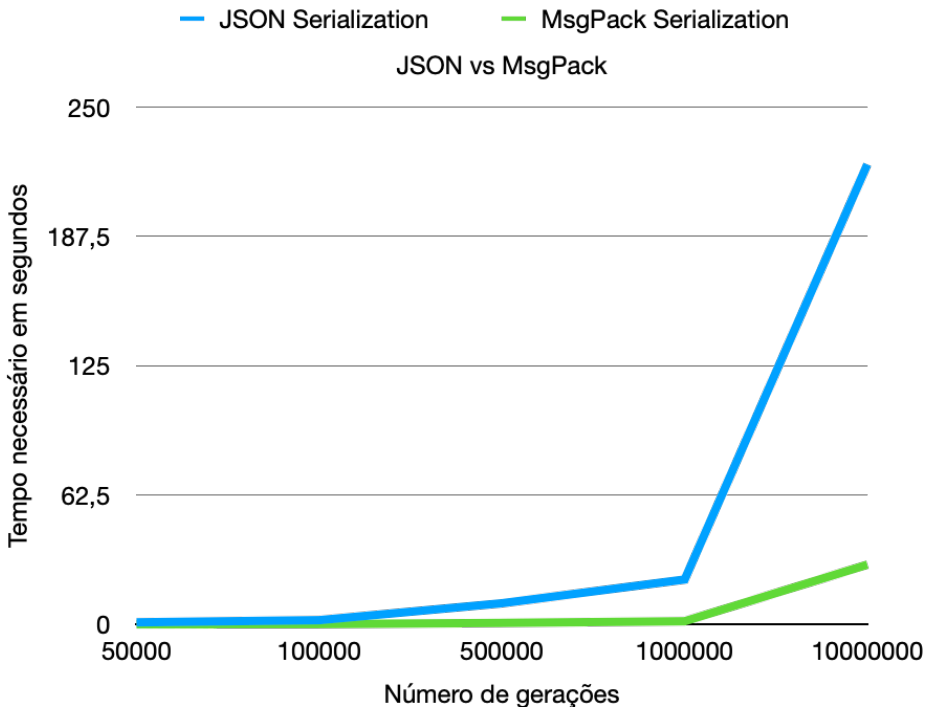

Figura 10 - Diferença entre o tempo de serialização do formato JSON e MsgPack

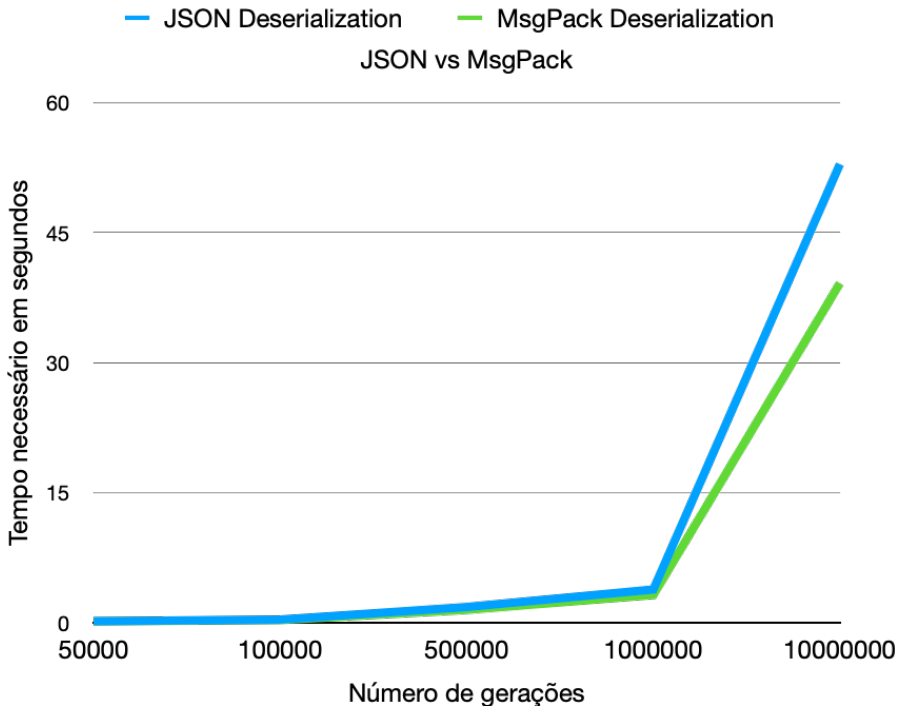

Figura 11 - Diferença entre o tempo de desserialização do formato JSON e MsgPack

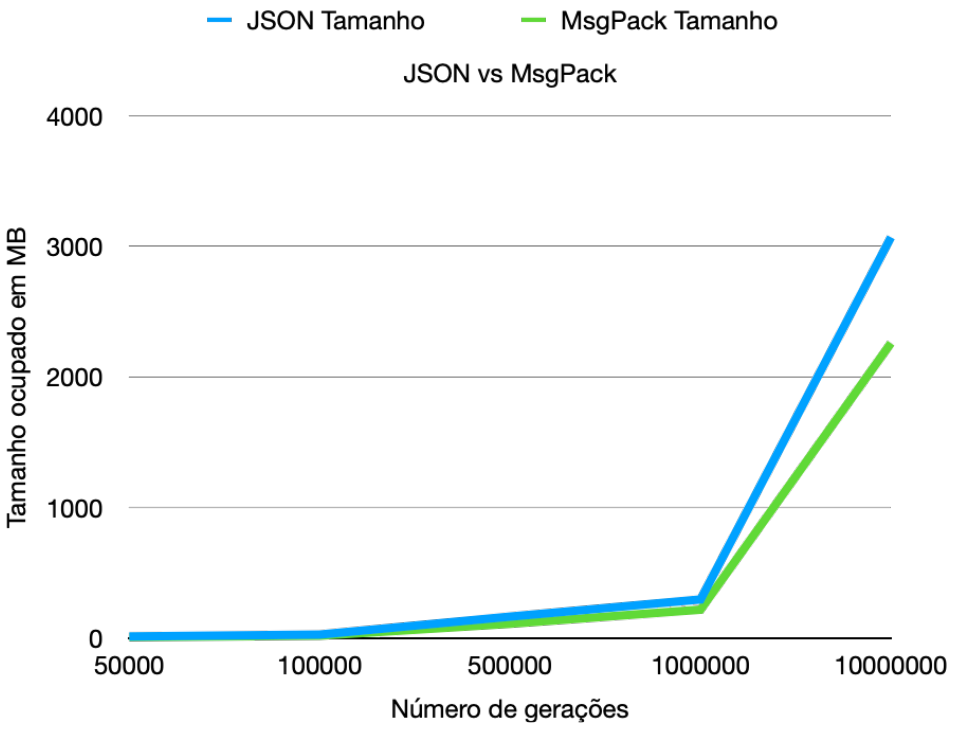

Figura 12 - Diferença do tamanho ocupado em disco pelos ficheiros serializados no formato JSON e MsgPack

	JSON					MSG PACK				
Número de Gerações	Serialization(s)	STDEV(s)	Deserialization(s)	STDEV(s)	Tamanho(MB)	Serialization(s)	STDEV(s)	Deserialization(s)	STDEV(s)	Tamanho(MB)
1000	0,04	0,007	0,004	0,0006	0,276	0,002	0,0008	0,003	0,001	0,204
5000	0,19	0,02	0,02	0,002	1,37	0,008	0,0009	0,01	0,001	1
10000	0,40	0,04	0,04	0,002	2,75	0,02	0,005	0,03	0,007	2,01
50000	1,98	0,14	0,22	0,02	14,1	0,09	0,01	0,15	0,02	10,2
100000	4,04	0,21	0,44	0,04	28,3	0,18	0,01	0,29	0,02	20,8
500000	20,94	0,74	2,17	0,10	145	1,07	0,05	1,44	0,11	107
1000000	44,76	4,26	5,11	0,80	290	2,36	0,11	3,11	0,24	216

Figura 13 - Tabela com os dados gerados pelo computador Diogo Filipe

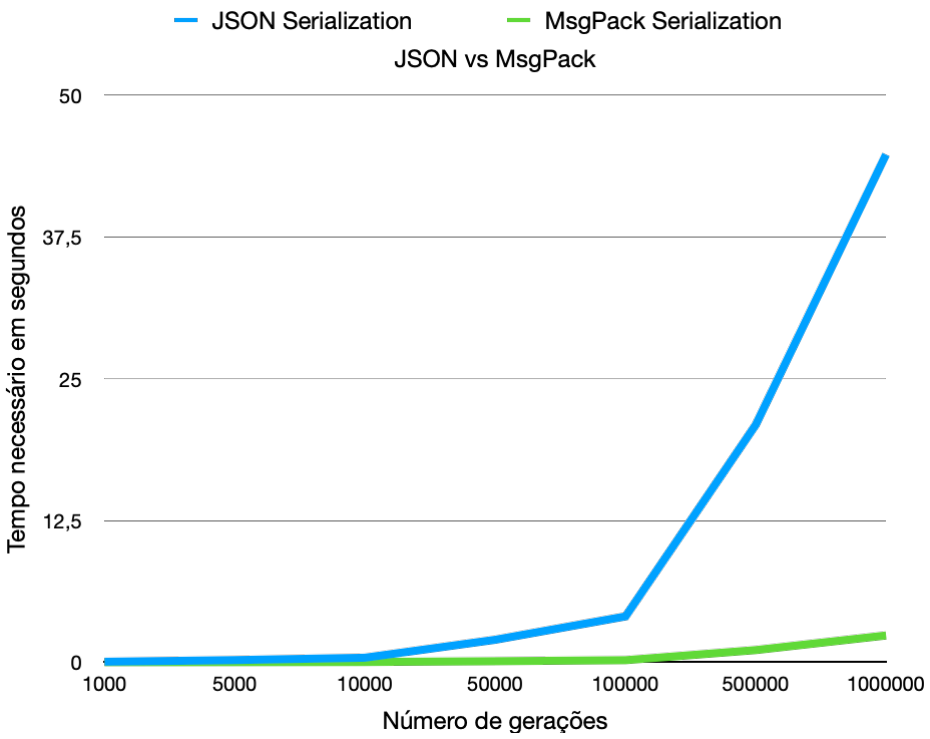

Figura 15 - Diferença entre o tempo de serialização do formato JSON e MsgPack

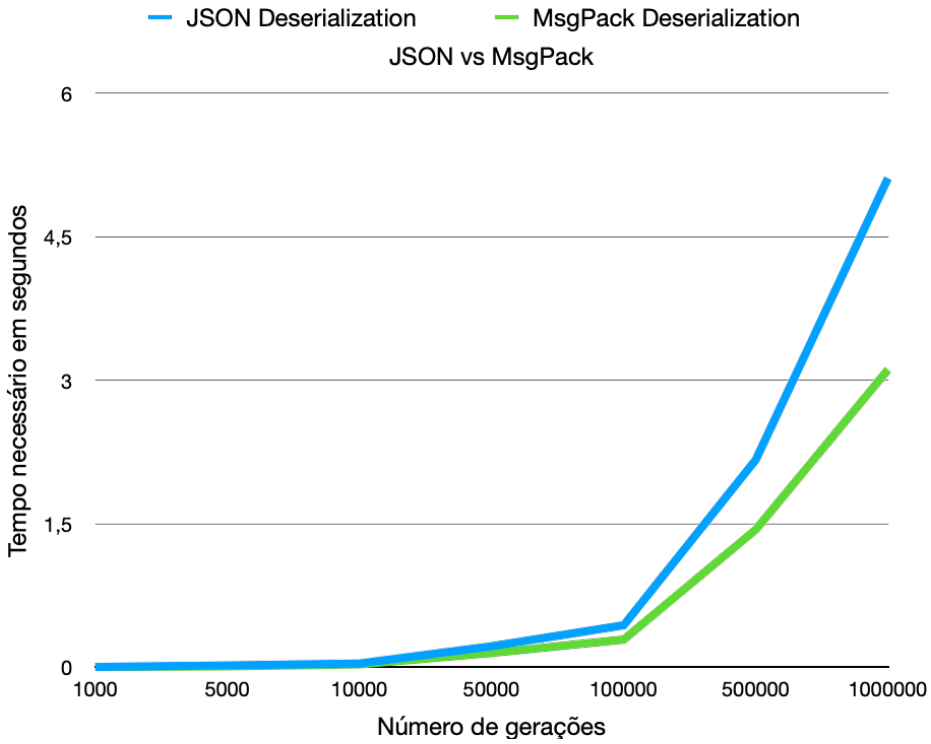

Figura 14 - Diferença entre o tempo de desserialização do formato JSON e MsgPack

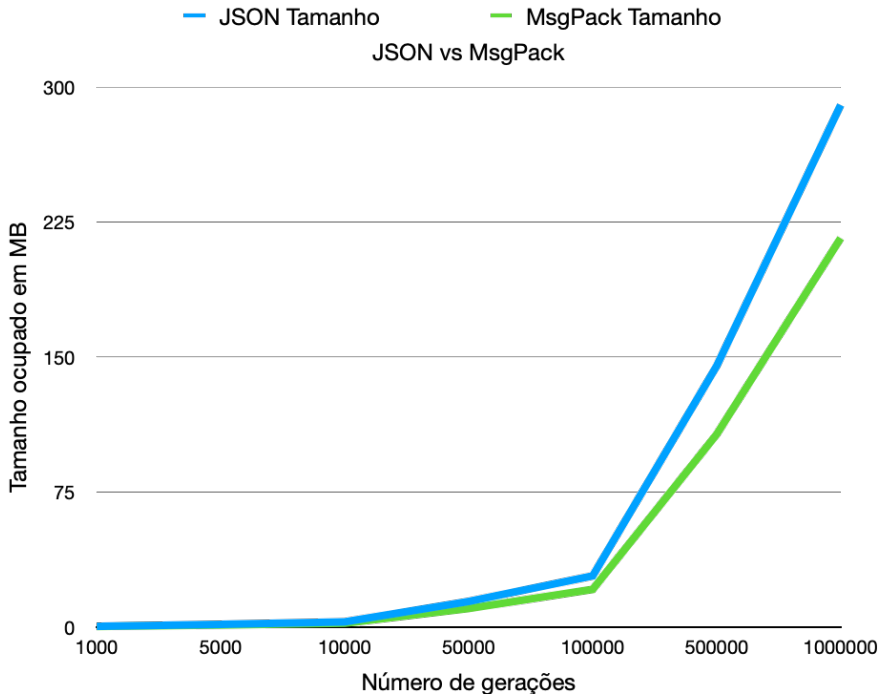

Figura 16 - Diferença do tamanho ocupado em disco pelos ficheiros serializados no formato JSON e MsgPack

4.1 Análise dos resultados experimentais

Através da análise dos gráficos acima, conseguimos observar que é o processo de serialização que demora mais tempo a finalizar no geral. Neste processo, é possível observar que existe um aumento exponencial no tempo de serialização do JSON, enquanto que o MessagePack tem um crescimento pouco acentuado, o que se traduz numa serialização entre 93% e 96% mais rápida do que no formato JSON.

Ao analisar o tempo de desserialização da informação, esta é significativamente menor comparada com o tempo de serialização (cerca de 8x menor) e, também ocorre de forma mais rápida no formato binário MessagePack.

Finalmente relativamente ao espaço ocupado em disco, este é maior no formato *JSON* e, quando maior for o volume de informação melhor será a capacidade do formato *MessagePack* de armazenamento dado que é mais eficiente e ocupa cerca de 22% de espaço a menos que o *JSON*.

5 Conclusão

Através das tabelas e dos gráficos da secção anterior, conseguimos observar que o formato *MessagePack* é superior ao formato *JSON* em todos os aspetos sendo estes o tempo de serialização e desserialização e o espaço que ocupa em disco a informação serializada. Esta conclusão seria previsível uma vez que o formato *MessagePack* é do tipo binário o que permite codificar a informação usando um menor número de *bytes*, ao contrário do formato *JSON* que é um formato de texto e como tal necessita de um maior número de *bytes* para codificar a mesma informação de modo a deixar a informação legível.