Evolutionary Algorithms – a2rl

André Filipe Frade Guerra

Outubro/2022

Conteúdo

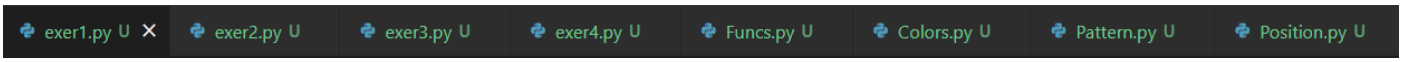
Introdução

Neste relatório irá ser explicado resumidamente as classes criadas, funcionalidade dos métodos criados e por fim a execução e discussão dos resultados dos exercícios.

Este trabalho tem como objetivo implementar vários algoritmos que descobrem um padrão pré-definido/solução, e comparar o tempo e tentativas que cada um levou a processar até chegar à solução. Este padrão pode ter um tamanho variável e em cada posição terá uma cor que foi atribuída aleatoriamente.

Classes e métodos

Para desenvolver os exercícios da segunda prática da disciplina de Introdução à Aprendizagem Automática foi desenvolvido um código em python usando o vsCode para realizar a mesma.



As seguintes classes foram criadas, sendo as 4 últimas auxiliares para as anteriores, que são os exercícios resolvidos.

Começando pela classe “Funcs.py”, contem métodos que são comuns nos exercícios:

```
4  def BuildPatterns(howManyPatterns):
5      aux = []
6      aux.append(1)
7      w = 2
8      for j in range(howManyPatterns):
9          aux.append(w)
10         w += 2
11     return aux
```

Este método recebe um valor numérico que vai construir um array com o tamanho das patterns a serem testadas.

Ex: se receber valor 9, irá retornar um array = [1, 2, 4, 6, 8, …, 18], basta duplicar o valor que foi recebido para percebermos qual o tamanho da última pattern.

```
13 def GetRandomColor(howManyColors):
14     r =  randrange(1,howManyColors + 1)
15     match r:
16         case 1: return Colors.RED
17         case 2: return Colors.BLUE
18         case 3: return Colors.BROWN
19         case 4: return Colors.GREEN
20         case 5: return Colors.WHITE
21         case 6: return Colors.BLACK
22         case 7: return Colors.PURPLE
23         case 8: return Colors.ORANGE
```

Neste método é retornada uma cor aleatória tendo em conta as cores possíveis que o são definidas no início do algoritmo, estas podem ir de 2 a 8 no máximo.

```
25 def CreateRandomPatern(howManyColors, cols):
26     r = []
27     for i in range(cols):
28         r.append(GetRandomColor(howManyColors))
29     return r
```

Este método retorna uma pattern aleatória tendo em conta o tamanho da pattern em questão “cols” e quantas cores foram definidas no início do algoritmo.

```python
def CreateStartPatern(cols):
    r = []
    for i in range(cols):
        r.append(Colors.NO_COLOR)
    return r
```

Este método retorna uma pattern com o tamanho definido, mas sem cores definidas.

```python
def ValuePatern(solution, patern, cols):
    value = cols
    for i in range(cols):
        if patern[i] != solution[i]:
            value -= 1
    return value
```

Este método retorna o valor de uma pattern tendo em conta a solução. Se a pattern for de tamanho 15, esse será o valor mais alto que será retornado pois significa que é igual à solução.

```python
def ValuePaternInverse(solution, patern, cols):
    value = 0
    for i in range(cols):
        if patern[i] == solution[i]:
            value += 1
    return value
```

Este método é o inverso do anterior, porque retorna 0 quando a pattern é igual a solução.

```python
def FlipOneBit(pattern, sizePattern, howManyColors):
    aux = pattern[:]
    aux[randrange(0, sizePattern)] = GetRandomColor(howManyColors)
    return aux
```

Este método recebe uma pattern, na qual será alterado numa das posições, aleatoriamente escolhida, por uma cor, também aleatoriamente escolhida.

```python
def CrossOver(best30Paterns, sizePattern):
    colors1 = best30Paterns[randrange(0, len(best30Paterns) - 1 )].colors[:]
    colors2 = best30Paterns[randrange(0, len(best30Paterns) - 1 )].colors[:]
    colors1[randrange(0, sizePattern)] = colors2[randrange(0, sizePattern)]
    return colors1
```

Este método recebe um array com X patterns (tendo em conta o exercício 3 serão 30 patterns). Destas 30 patterns são escolhidas 2 aleatoriamente. Na primeira pattern é escolhida uma posição aleatória que irá receber uma cor da segunda pattern que também foi escolhida aleatoriamente.

Exercício 1

Presente no ficheiro exer1.py, este ficheiro começa por declarar vários valores iniciais que definem o algoritmo, como quantas patterns vão ser testadas, runs por pattern, quantas cores, e o tempo máximo que o algoritmo pode correr (neste caso 3600 segundos o que corresponde a 1 hora).

```
5   howManyPatterns = 9# howManyPatterns = 9 => 1, 2, 4, ..., 18
6   patterns = BuildPatterns(howManyPatterns)
7   howManyColors = 2 # MAX is 8
8   runs = 30 # sets the runs for each pattern
9   maxRunTimeSecs = 3600
```

De seguida, encontra-se o algoritmo implementado, que vai testar todas as patterns tendo em conta as runs que essa pattern deve ser testada. Para cada run é criada uma solução aleatória, que vai tentar ser descoberta criando-se uma pattern aleatória (linha 25). Na linha 28 e 29, encontra-se comentado os métodos que calculam a distância incremental e inversa, entre a solução e a pattern gerada. Esta pattern gerada aleatoriamente é sempre gerada até ser igual à solução definida.

```
11  patern = []
12  runTimes = []
13  countTentativasRuns = []
14  stOverall = time.time()
15  for sizePattern in patterns:
16      runtimesForEachPattern = []
17      countTentativasRunsForEachPattern = []
18      for run in range(runs):
19          patern = CreateStartPatern(sizePattern)
20          solution = CreateRandomPatern(howManyColors, sizePattern)
21
22          countTentativas = 0
23          st = time.time()
24          while solution != patern:
25              patern = CreateRandomPatern(howManyColors, sizePattern)
26              countTentativas += 1
27
28              #print("ValuePatern: " + str(ValuePatern(solution, patern, sizePattern)))
29              #print("ValuePaternInverse: " + str(ValuePaternInverse(solution, patern, sizePattern)))
30
31          runtimesForEachPattern.append(time.time() - st)
32          countTentativasRunsForEachPattern.append(countTentativas)
33
34      if time.time() - stOverall > maxRunTimeSecs:
35          print("1 hour of execution completed. Stopping...")
36          break
37
38      runTimes.append(runtimesForEachPattern)
39      countTentativasRuns.append(countTentativasRunsForEachPattern)
40
41  print("tentativas: " + str(countTentativasRuns) + " \ntime: " + str(runTimes))
```

No fim serão mostrados 2 gráficos com boxplots, que contêm as tentativas e os tempos necessários para descobrir cada pattern a ser testada.

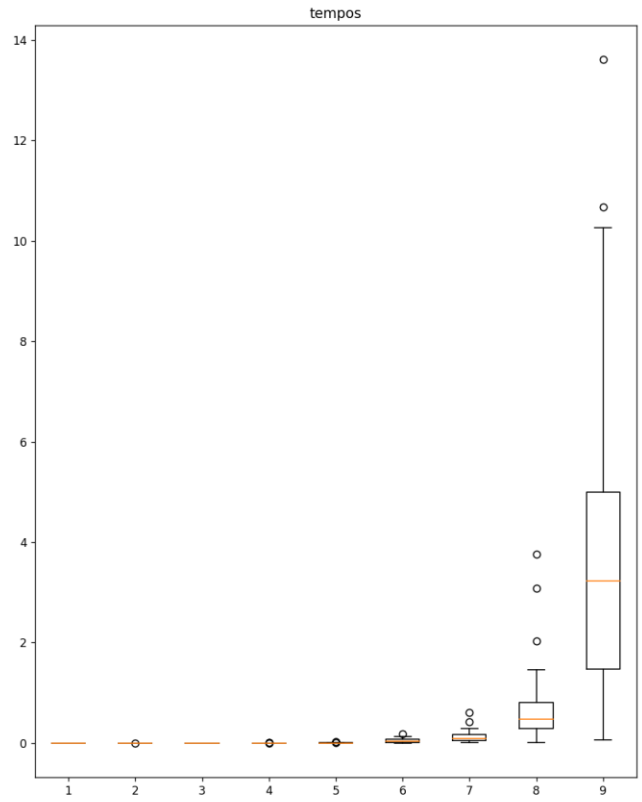
```
43  fig, axs = plt.subplots(1,2)
44  axs[0].boxplot(runTimes)
45  axs[0].set_title('tempos')
46  axs[1].boxplot(countTentativasRuns)
47  axs[1].set_title('tentativas')
48  plt.show()
```

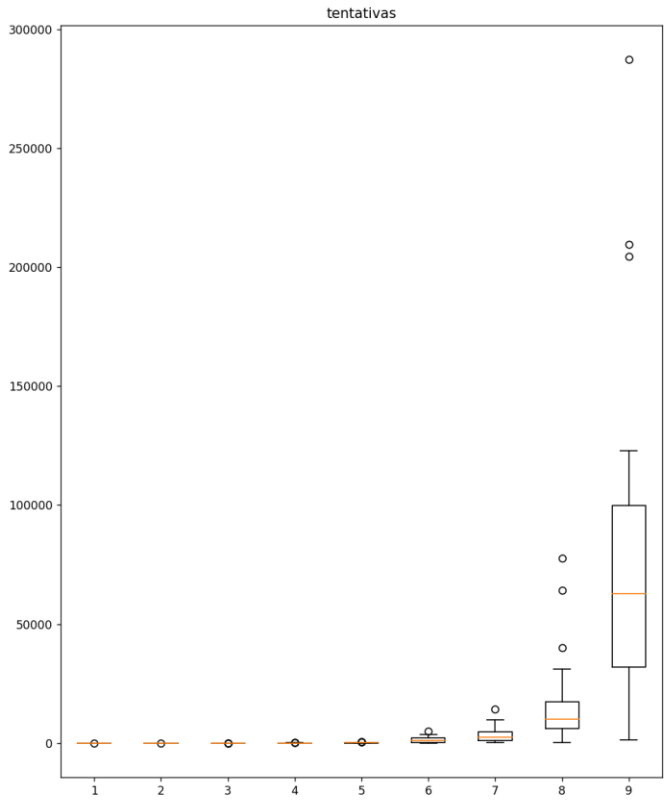
Resultados exercício 1

O exercício 1 foi inicializado para testar até à pattern de tamanho 16.

```
5  howManyPatterns = 8# howManyPatterns = 8 => 1, 2, 4, ..., 16
6  patterns = BuildPatterns(howManyPatterns)
7  howManyColors = 2 # MAX is 8
8  runs = 30 # sets the runs for each pattern
9  maxRunTimeSecs = 3600
```

Nos gráficos apresentados, podemos observar os resultados das tentativas e os tempos que cada tamanho das patterns demoraram a chegar à solução. Tendo a última solução um tempo médio de aproximadamente 3 segundos e 70 000 tentativas para chegar a solução.





Exercício 2

Presente no ficheiro exer2.py, este ficheiro começa por declarar vários valores iniciais que definem o algoritmo, como quantas patterns vão ser testadas, runs por pattern, quantas cores, e o tempo máximo que o algoritmo pode correr (neste caso 3600 segundos o que corresponde a 1 hora). E adicionalmente, comparado com o exercício 1, foi definido também quantas vezes o algoritmo pode alterar uma posição aleatória de uma pattern até que esta esteja mais próxima da solução "runsForFlipOneBit", se não mesmo encontrar a solução.

```
5   howManyPatterns = 9# howManyPatterns = 9 => 1, 2, 4, ..., 18
6   patterns = BuildPatterns(howManyPatterns)
7   howManyColors = 2 #MAX is 8
8   runs = 30 # sets the max size of the patern to reach
9   runsForFlipOneBit = 1000
10  maxRunTimeSecs = 3600
```

Após a declaração e definição das variáveis, encontra-se o algoritmo que irá tentar descobrir a solução, alterando apenas uma posição de uma pattern regada aleatoriamente, que para quando o limite de tentativas de alterar uma posição foi alcançado "runsForFlipOneBit", ou a solução foi encontrada. Quando a alteração de uma posição fica mais próxima da solução, esta é armazenada e utilizada na próxima iteração de alterar uma outra posição.

```
12  patern = []
13  runTimes = []
14  countTentativasRuns = []
15  stOverall = time.time()
16  for sizePattern in patterns:
17      runtimesForEachPattern = []
18      countTentativasRunsForEachPattern = []
19
20      for run in range(runs):
21
22          patern = CreateStartPatern(sizePattern)
23          solution = CreateRandomPatern(howManyColors, sizePattern)
24
25          countTentativas = 0
26          st = time.time()
27          for rffon in range(runsForFlipOneBit):
28              auxPattern = FlipOneBit(patern, sizePattern, howManyColors)
29
30              if ValuePaternInverse(solution, auxPattern, sizePattern) > ValuePaternInverse(solution, patern, sizePattern):
31                  patern = auxPattern
32
33              if solution == patern:
34                  break
35
36              countTentativas += 1
37
38          if solution != patern:
39              print("Solution not found at pattern size: " + str(sizePattern))
40
41          runtimesForEachPattern.append(time.time() - st)
42          countTentativasRunsForEachPattern.append(countTentativas)
43
44      if time.time() - stOverall > maxRunTimeSecs:
45          print("1 hour of execution completed. Stopping...")
46          break
47
48      runTimes.append(runtimesForEachPattern)
49      countTentativasRuns.append(countTentativasRunsForEachPattern)
50
51  #print("tentativas: " + str(countTentativasRuns) + " \ntime: " + str(runTimes))
```

No fim serão mostrados 2 gráficos com boxplots, que contêm as tentativas e os tempos necessários para descobrir cada pattern a ser testada.

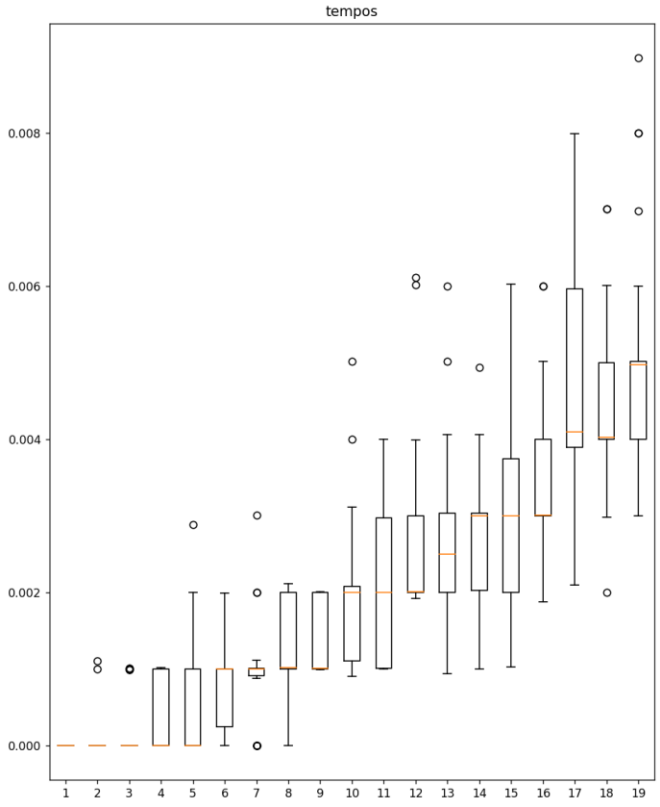
```
43    fig, axs = plt.subplots(1,2)
44    axs[0].boxplot(runTimes)
45    axs[0].set_title('tempos')
46    axs[1].boxplot(countTentativasRuns)
47    axs[1].set_title('tentativas')
48    plt.show()
```

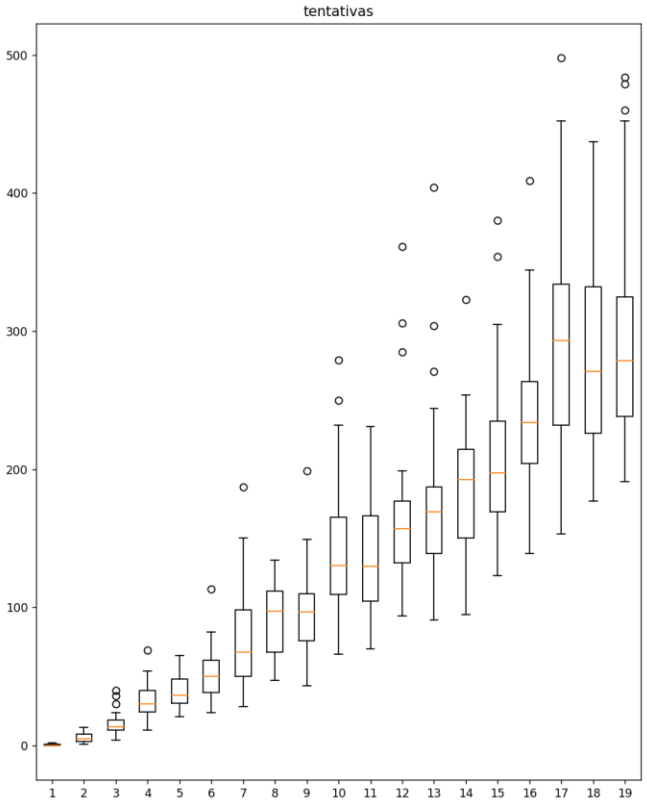
Resultados exercício 2

Tendo em conta a velocidade com que o algoritmo chega a solução o tamanho das patterns a ser testado foi aumentado para 18 (1, 2, 4, 6, …, 36), sendo o 36 a pattern maior a ser testada.

```
5     howManyPatterns = 18# howManyPatterns = 9 => 1, 2, 4, ..., 18, ..., 36
6     patterns = BuildPatterns(howManyPatterns)
7     howManyColors = 2 #MAX is 8
8     runs = 30 # sets the max size of the patern to reach
9     runsForFlipOneBit = 1000
10    maxRunTimeSecs = 3600
```

Com os gráficos obtidos, podemos observar o tempo e as tentativas a aumentarem quando a pattern aumenta de tamanho, este algoritmo chega à solução com a pattern mais alta (pattern com 36 posições), num tempo médio de 0.005 segundos e 280 tentativas. Comparando como exercício 1, este é significativamente mais rápido, pois quando se altera uma das posições da pattern, esta só é guardada caso esteja mais próxima da solução.





Exercício 3

Presente no ficheiro exer3.py, este ficheiro começa por declarar vários valores iniciais que definem o algoritmo, como quantas patterns vão ser testadas, runs por pattern, quantas cores, e o tempo máximo que o algoritmo pode correr (neste caso 3600 segundos o que corresponde a 1 hora). E adicionalmente, comparado com o exercício 1, foi definido também o tamanho da população a ser gerado por cada pattern a ser testada "howManyPatternsPopulation".

```
 7   howManyPatterns = 9# howManyPatterns = 9 => 1, 2, 4, ..., 18
 8   patterns = BuildPatterns(howManyPatterns)
 9   howManyPatternsPopulation = 100 # 100
10   howManyColors = 2 #MAX is 8
11   runs = 30 # sets the max size of the patern to reach
12   maxRunTimeSecs = 3600
```

De seguida, encontra-se o algoritmo que tenta encontrar a solução, organizando a população das patterns geradas aleatoriamente pelo valor mais próximo da solução. A cada tentativa de descobrir a solução, o algoritmo escolhe as 30 melhores patterns, que de seguida são usadas para gerar 70 novas patterns alterando uma das posições aleatoriamente. Este processo volta a repetir-se selecionando novamente as melhores 30 patterns até que a solução seja encontrada. A solução é declarada quando a primeira posição da população de patterns é igual a solução, pois essa população de patterns está organizada da mais próxima até a mais longe da solução.

```
patern = []
runTimes = []
countTentativasRuns = []
stOverall = time.time()
for sizePattern in patterns:
    runtimesForEachPattern = []
    countTentativasRunsForEachPattern = []

    for run in range(runs):
        patern.clear()
        solution = CreateRandomPatern(howManyColors, sizePattern)

        for j in range(howManyPatternsPopulation):
            p = Pattern()
            p.colors = CreateRandomPatern(howManyColors, sizePattern)
            p.valueInverse = ValuePaternInverse(solution, p.colors, sizePattern)
            p.value = ValuePatern(solution, p.colors, sizePattern)
            patern.append(p)

        countTentativas = 0
        st = time.time()
        while solution != patern[0].colors:

            patern.sort(key=lambda x: x.value, reverse=True)
            countTentativas += 1
            best30Paterns = []
            new70Paterns = []
            for p in range(30):
                best30Paterns.append(patern[p])

            for p in range(70):
                p = Pattern()
                p.colors = FlipOneBit(best30Paterns[randrange(0,29)].colors, sizePattern, howManyColors)
                p.valueInverse = ValuePaternInverse(solution, p.colors, sizePattern)
                p.value = ValuePatern(solution, p.colors, sizePattern)
                new70Paterns.append(p)

            patern.clear()
            patern.extend(best30Paterns)
            patern.extend(new70Paterns)

        runtimesForEachPattern.append(time.time() - st)
        countTentativasRunsForEachPattern.append(countTentativas)

    if time.time() - stOverall > maxRunTimeSecs:
        print("1 hour of execution completed. Stopping...")
        break

    runTimes.append(runtimesForEachPattern)
    countTentativasRuns.append(countTentativasRunsForEachPattern)

#print("tentativas: " + str(countTentativasRuns) + " \ntime: " + str(runTimes))
```

No fim serão mostrados 2 gráficos com boxplots, que contêm as tentativas e os tempos necessários para descobrir cada pattern a ser testada.

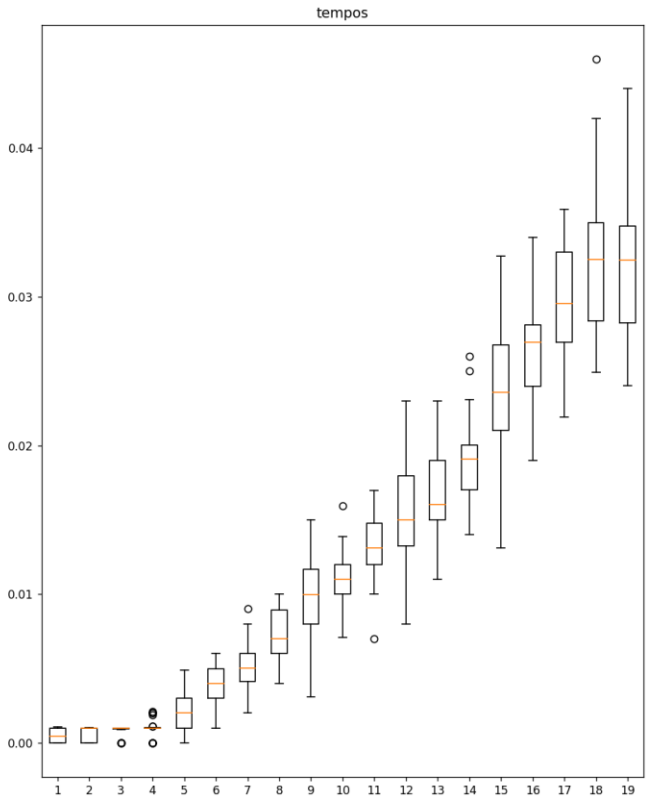
```
fig, axs = plt.subplots(1,2)
axs[0].boxplot(runTimes)
axs[0].set_title('tempos')
axs[1].boxplot(countTentativasRuns)
axs[1].set_title('tentativas')
plt.show()
```

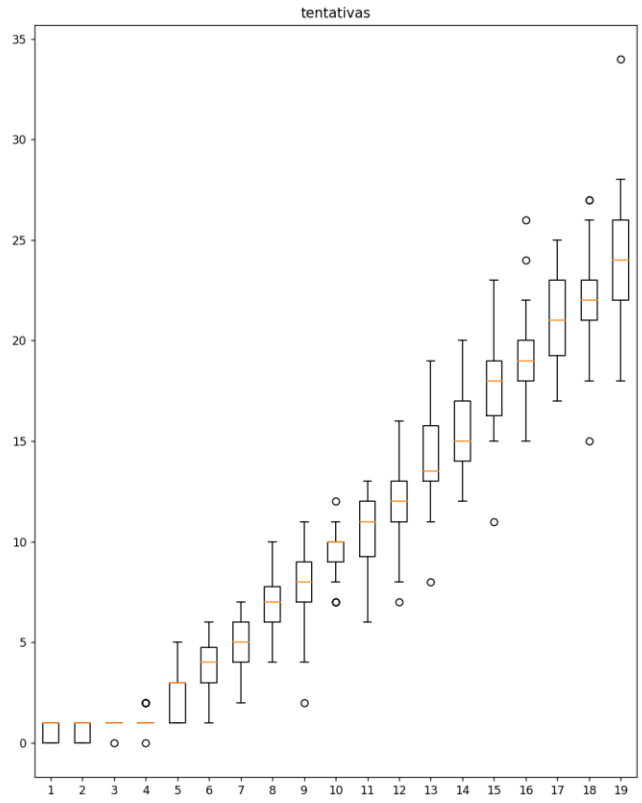
Resultados exercício 3

No exercício 3, as variáveis de inicialização, permaneceram iguais às do exercício 2, mais a variável que define o tamanho da população a ser gerada, neste caso 100.

```
howManyPatterns = 18# howManyPatterns = 9 => 1, 2, 4, ..., 18, ..., 36
patterns = BuildPatterns(howManyPatterns)
howManyPatternsPopulation = 100 # 100
howManyColors = 2 #MAX is 8
runs = 30 # sets the max size of the patern to reach
maxRunTimeSecs = 3600
```

Com os gráficos seguintes, podemos concluir que apesar do tempo ter aumentado, as tentativas reduziram significativamente, o que no fim originou um algoritmo ainda mais rápido, comparado com os exercícios anteriores. Esta diminuição das tentativas, pode explicar-se, devido a este algoritmo guardar recursivamente as 30 melhores patterns em 100, e criar uma população fazendo a alteração de uma posição aleatória nessas 30 patterns, que também são escolhidas aleatoriamente.





Exercício 4

Presente no ficheiro exer4.py, este ficheiro começa por declarar vários valores iniciais que definem o algoritmo, como quantas patterns vão ser testadas, runs por pattern, quantas cores, e o tempo máximo que o algoritmo pode correr (neste caso 3600 segundos o que corresponde a 1 hora). Adicionalmente, comparado com o exercício 3, foram adicionadas 2 variáveis que definem a percentagem das melhores patterns presentes na população que irão gerar a próxima população, como também a percentagem das novas petterns a serem criadas, neste caso está definido escolher as melhores 30% da população, e criar 70% patterns (estas duas variáveis devem somar no total 1).

```
6   howManyPatterns = 9# howManyPatterns = 9 => 1, 2, 4, ..., 18
7   patterns = BuildPatterns(howManyPatterns)
8   howManyPatternsPopulation = 100 # 100
9   bestPercentage = 0.3 # best value patterns percentage to the next generation
10  newPercentage = 0.7 # percenatage how many new patterns created based on howManyPatternsPopulation
11  howManyColors = 2 #MAX is 8
12  runs = 30 # sets the max size of the patern to
13  maxRunTimeSecs = 3600
```

De seguida, encontra-se o algoritmo que tenta encontrar a solução, e semelhante ao exercício 3, organiza a população das patterns geradas aleatoriamente pelo valor mais próximo da solução. A cada tentativa de descobrir a solução, o algoritmo escolhe as 30 melhores patterns, que de seguida são usadas para gerar 70 novas patterns.

Estas 70 novas patterns são geradas a partir da escolha aleatória de duas patterns das 30 melhores, substituído uma posição aleatória da primeira pela posição aleatória de segunda (ver descrição do método “CrossOver”).

Este processo volta a repetir-se selecionando novamente as melhores 30 patterns até que a solução seja encontrada. A solução é declarada quando a primeira posição da população de patterns é igual a solução, pois essa população de patterns está organizada da mais próxima até a mais longe da solução.

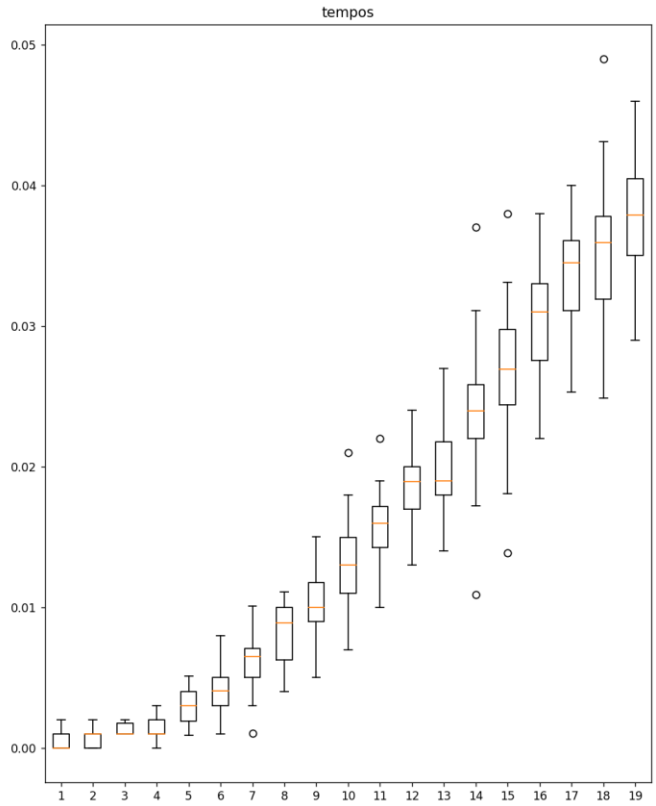
```
patern = []
runTimes = []
countTentativasRuns = []
stOverall = time.time()
for sizePattern in patterns:
    runtimesForEachPattern = []
    countTentativasRunsForEachPattern = []

    for run in range(runs):
        patern.clear()
        solution = CreateRandomPatern(howManyColors, sizePattern)
        for j in range(howManyPatternsPopulation):
            p = Pattern()
            p.colors = CreateRandomPatern(howManyColors, sizePattern)
            p.valueInverse = ValuePaternInverse(solution, p.colors, sizePattern)
            p.value = ValuePatern(solution, p.colors, sizePattern)
            patern.append(p)

        countTentativas = 0
        st = time.time()
        while solution != patern[0].colors:
            patern.sort(key=lambda x: x.value, reverse=True)
            countTentativas += 1
            best30Paterns = []
            new70Paterns = []

            for p in range(round(howManyPatternsPopulation * bestPercentage)):
                best30Paterns.append(patern[p])

            for p in range(round(howManyPatternsPopulation * newPercentage)):
                p = Pattern()
                p.colors = CrossOver(best30Paterns, sizePattern)
                p.valueInverse = ValuePaternInverse(solution, p.colors, sizePattern)
                p.value = ValuePatern(solution, p.colors, sizePattern)
                new70Paterns.append(p)

            patern.clear()
            patern.extend(best30Paterns)
            patern.extend(new70Paterns)

        runtimesForEachPattern.append(time.time() - st)
        countTentativasRunsForEachPattern.append(countTentativas)

    if time.time() - stOverall > maxRunTimeSecs:
        print("1 hour of execution completed. Stopping...")
        break

    runTimes.append(runtimesForEachPattern)
    countTentativasRuns.append(countTentativasRunsForEachPattern)

#print("tentativas: " + str(countTentativasRuns) + " \ntime: " + str(runTimes))
```

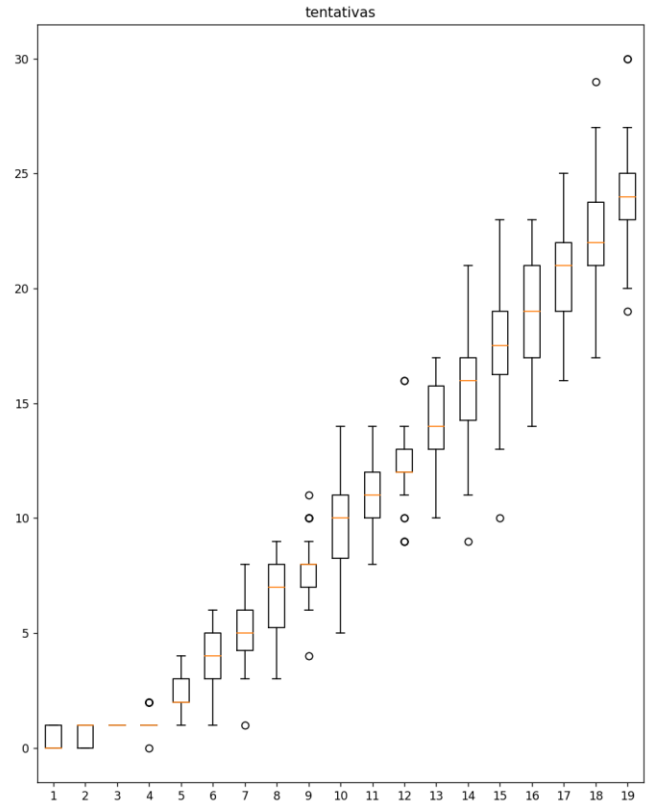
Resultados exercício 4

No exercício 4, utilizando as variáveis de inicialização, permaneceram iguais ao exercício 3, com a adição de duas novas, a percentagem da população que deve ser guardada para criar a próxima geração (30% = 0.3), e a percentagem a ser criada (70% = 0.7).

```
6   howManyPatterns = 18# howManyPatterns = 9 => 1, 2, 4, ..., 18, ..., 36
7   patterns = BuildPatterns(howManyPatterns)
8   howManyPatternsPopulation = 100 # 100
9   bestPercentage = 0.3 # best value patterns percentage to the next generation
10  newPercentage = 0.7 # percenatage how many new patterns created based on howManyPatternsPopulation
11  howManyColors = 2 #MAX is 8
12  runs = 30 # sets the max size of the patern to
13  maxRunTimeSecs = 3600
```

Nos gráficos seguintes, podemos observar valores muito semelhantes ao exercício 3, mesmo o algoritmo sendo um pouco diferente do anterior. A semelhança entre o exercício anterior é a percentagem a ser usada para guardar e criar a população, 30% e 70% respetivamente, apesar de neste, as posições das patterns serem alteradas tendo em conta outra pattern também incluída nas 30% melhores.

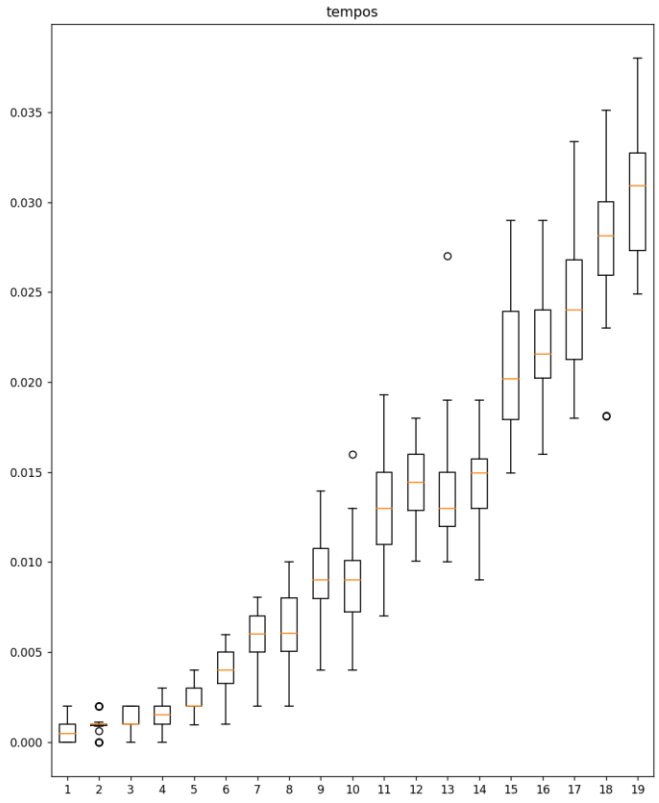


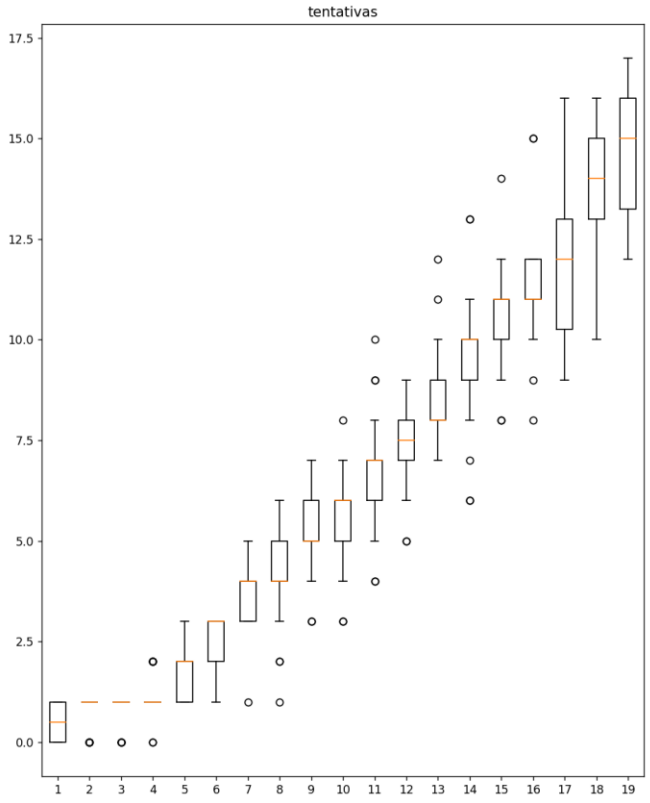


Para compararmos a percentagem de patterns guardadas e criadas, as variáveis de inicialização foram alteradas, de maneira que as patterns guardadas correspondem a 10%, e as criadas a 90%.

```
6   howManyPatterns = 18# howManyPatterns = 9 => 1, 2, 4, ..., 18, ..., 36
7   patterns = BuildPatterns(howManyPatterns)
8   howManyPatternsPopulation = 100 # 100
9   bestPercentage = 0.1 # best value patterns percentage to the next generation
10  newPercentage = 0.9 # percenatage how many new patterns created based on howManyPatternsPopulation
11  howManyColors = 2 #MAX is 8
12  runs = 30 # sets the max size of the patern to
13  maxRunTimeSecs = 3600
```

Depois da execução do algoritmo, podemos observar que guardando apenas as 10% mais próximas da solução, este converge para a solução em menos tempo e tentativas.


tempos
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19


tentativas
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Conclusão

Inicialmente o código foi feito para poder trabalhar com várias cores, assim sendo possível em cada posição da pattern poderem existir até 8 cores, os exercícios anteriores foram executados sempre apenas com 2 cores possíveis.

Para podermos observar mais facilmente a relação entre a quantidade de cores e a velocidade, foi criado um rating que vai ser calculado da seguinte forma:

rating = (média dos tempos * 500) + média das tentativas

```
print("######## rating => " + str(GetRatting(runTimes)*500 + GetRatting(countTentativasRuns)))
```

```
def GetRatting(array):
    somaRunsF = 0
    somaRunsCountF = 0
    for rts in array:
        somaRun = 0
        somaRunCount = 0
        for rt in rts:
            somaRun += rt
            somaRunCount += 1

        somaRunsF += somaRun / somaRunCount
        somaRunsCountF += 1
    return somaRunsF / somaRunsCountF
```

	2 cores	3 cores	4 cores
Exercício 1	10421	10674	---
Exercício 2	134	201	266
Exercício 3	16	27	38
Exercício 4 30%	16	28	---
Exercício 4 10%	11	19	---

Os dados não apresentados na tabela (---), são devido ao tempo elevado que o exercício 1 leva a chegar à solução com 4 cores, e devido a uma falha que pode estar a acontecer no exercício 4.

Esta falha pode ser explicada porque no exercício 4, a criação da população das patterns pode não conter uma das cores da solução, e também, como as novas patterns são geradas a partir das 10% ou 30% melhores, essas podem também não conter a cor, criando gerações sem a cor necessária para a solução. Ao contrário do exercício 3 em que a posição da pattern é alterada para uma cor aleatória das possíveis.

Assim, podemos concluir, que para um elevado número de cores o exercício 3 é mais indicado. Não apresenta falhas independentemente da quantidade de cores, como também, apresenta um rating inferior aos restantes exercícios 1 e 2.