

Introdução à Aprendizagem Automática

Assignment nº2

João Carlos Cambaia Gomes de Almeida

nº87583

| | |
|----------------------|-----------|
| Threads | 2 |
| Exercício 1 | 2 |
| a) | 2 |
| b) | 3 |
| c) | 5 |
| d) | 6 |
| Exercício 2 | 6 |
| Exercício 3 | 8 |
| Population | 8 |
| Stagnation | 8 |
| Resultados | 8 |
| Exercício 4 | 9 |
| Crossover | 9 |
| Resultados | 9 |
| Exercício 5 | 10 |
| Evaluate and Fitness | 10 |
| Mutation | 11 |
| Crossover | 13 |
| Exercício 6 | 13 |
| Generation | 13 |
| Evaluate and Fitness | 14 |
| Mutation | 15 |
| Crossover | 17 |
| Exercício 7 | 17 |

Threads

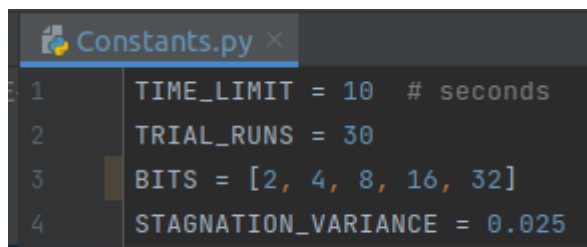
Para todos os exercícios decidi usar threads, tanto porque nunca as usei no Python então queria experimentar e também porque quando acabei o primeiro Assignment pensei que as podia ter usado e neste decidi desde o começo utilizá-las.

Para isso usei a seguinte estrutura:

```
def launch_threads(method_to_run, results: [Result]):  
    with concurrent.futures.ThreadPoolExecutor() as executor:  
        # [executor.submit(randomTest,bit) for bit in bits]  
        for _ in range(TRIAL_RUNS):  
            executor.map(method_to_run, BITS)
```

O método launch_threads recebe como argumento um method_to_run que é o método que as threads irão executar e uma lista de resultados, onde as threads irão colocar os seus resultados, para depois poder fazer a visualização.

As variáveis TRIAL_RUNS e BITS são as constantes usadas em todo o Assignment e estão no seguinte ficheiro:



```
Constants.py ×  
1 TIME_LIMIT = 10 # seconds  
2 TRIAL_RUNS = 30  
3 BITS = [2, 4, 8, 16, 32]  
4 STAGNATION_VARIANCE = 0.025
```

Exercício 1

a)

```
@staticmethod  
def random_bit_pattern(size: int) -> str:  
    result: str = ""  
    for _ in range(size):  
        result += Mastermind.random_bit()  
    return result  
  
@staticmethod  
def random_bit() -> str:  
    return random.choice(['0', '1'])
```

b)

A função que usei para fazer o random guessing recebe um tamanho de padrão e gera aleatoriamente um goal, depois entra no ciclo while até gerar um padrão igual ao goal ou até exceder o tempo limite, controlado pela variável `_sucess` que passa para false quando passamos o tempo máximo.

```
def _assignment2_exercise1_line_b(patternSize: int):
    _goal_pattern: str = Mastermind.random_bit_pattern(size=patternSize)
    _generated_pattern: str = ""

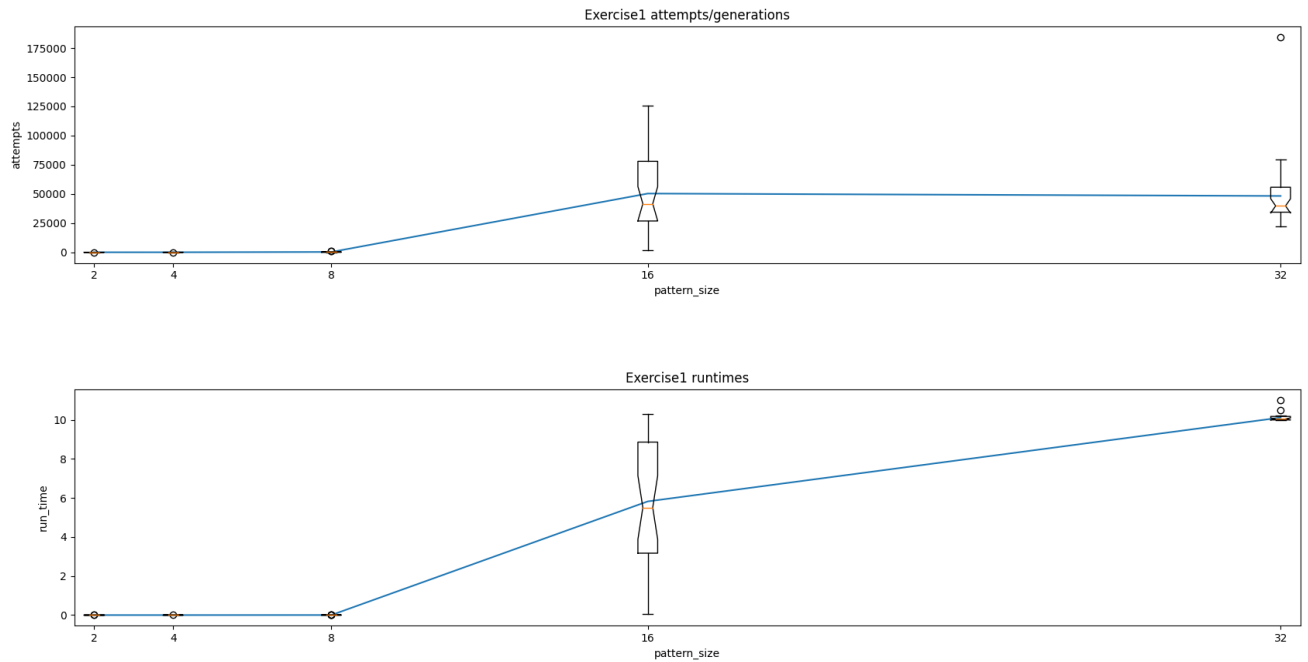
    _start = timeit.default_timer()
    _attempts = 0
    _success: bool = True

    while _goal_pattern != _generated_pattern and _success:
        _current_time = timeit.default_timer()
        if _current_time - _start > TIME_LIMIT:
            _success = False
            result = Result(run_time=timeit.default_timer() - _start, attempts=_attempts,
                           pattern_size=patternSize,
                           successfull=_success)
            store_result(result=result, results=exercise1_results_list, lock=exercise1_lock)
        else:
            _generated_pattern = Mastermind.random_bit_pattern(size=patternSize)
            _attempts += 1

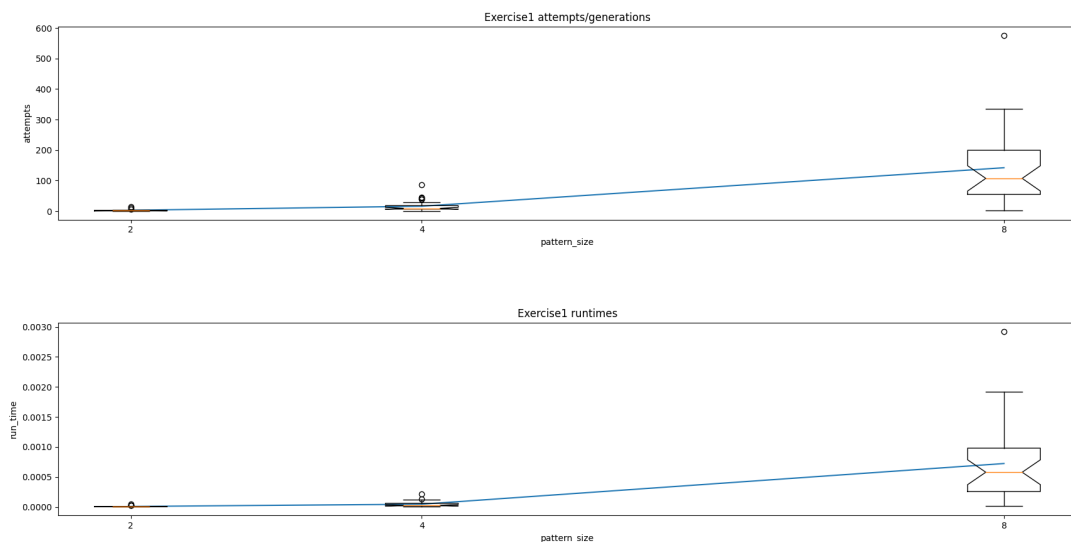
    if _success:
        _stop = timeit.default_timer()
        _time: float = _stop - _start
        _result = Result(run_time=_time, attempts=_attempts, pattern_size=patternSize, successfull=_success)

        #print("Assignment2_EvolutionaryMastermindSimulator::Exercise1:", threading.get_ident(),
        #      "population_mean_fitness:" + str(_result))
        store_result(result=_result, results=exercise1_results_list, lock=exercise1_lock)
```

Como estou a usar threads o tempo limite que estou a controlar é o tempo máximo para cada thread chegar ao goal pattern e não o tempo total de execução das 30 experiências.



Como podemos ver nos gráficos resultantes o nosso run_time para padrões com size de 2,4 e 8 é menos de 1 segundo porque para padrões com 16bits o tempo que o algoritmo demora a encontrar o goal salta para o intervalo entre 3 e 9 segundos, este salto em tempo de execução é correspondido pelo número de tentativas que foram efetuadas, nos tamanhos 2,4 e 8 as tentativas estão a volta do 0 (claro que não foram 0 tentativas para estes tamanhos mas como a escala do gráfico é dominada pelas 50000 tentativas dos 16bits os boxplots dos tamanhos mais pequenos são “squashed”)

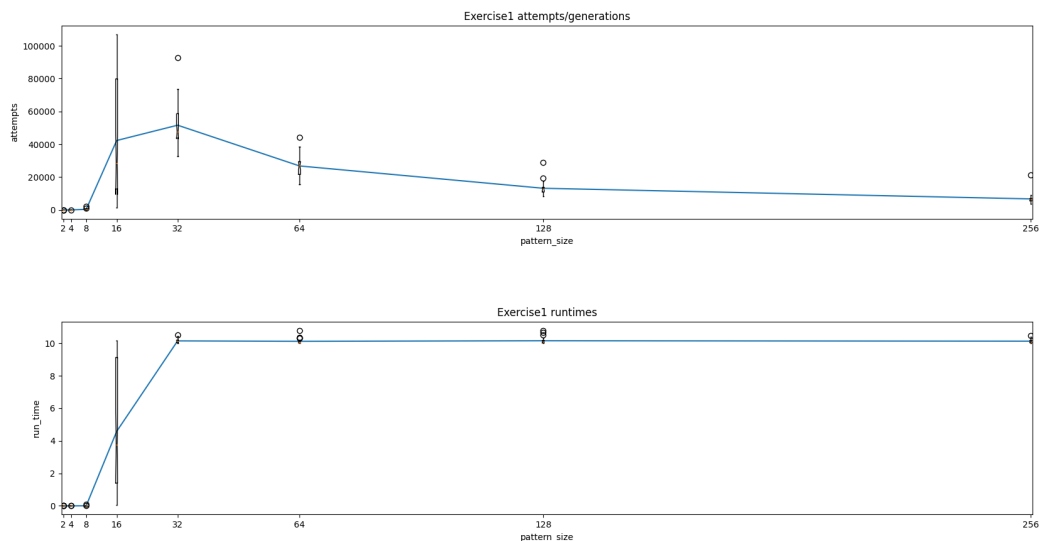


Na realidade os valores de tentativas para os sizes 2 e 4 estão a volta das 10 enquanto que tamanhos de 8 bits já estão nas poucas centenas

No primeiro gráfico podemos observar que o número de tentativas entre os 16 e os 32bits não aumentou nada comparando com o aumento que houve entre os 8 e os 16bits, isto é

porque, e podemos observar no gráfico dos run_times, limitei o tempo de procura de cada thread para 10 segundos, ou seja, cada thread não gasta mais de 10 segundos a tentar encontrar o goal pattern.

Este “efeito” é ainda mais predominante se experimentarmos com pattern sizes cada vez maiores, quanto maior o size, mais tempo a thread precisará para executar os seus guesses então como têm sempre o mesmo tempo limite acabam por efetuar cada vez menos tentativas, como podemos observar no seguinte gráfico:



Podemos ver que os run_times permanecem constantes nos 10 segundos e as tentativas efetuadas vão descendo à medida que o patter_size aumenta, porque para gerar cada novo guess demoramos cada vez mais o que começa a ocupar uma porção cada vez maior do tempo que a thread tem para chegar ao objetivo final

c)

Para a função de avaliação decidi contar quantos caracteres são diferentes do goal, para o fazer começo com um contador igual ao número de caracteres do goal e vou subtraindo por cada caracter igual.

Desta maneira, se não houver nenhum caracter igual o valor que é retornado é igual ao número de caracteres do goal e se forem todos diferentes o valor retornado é zero.

```
@staticmethod
def evaluate(goal: str, curr: str):
    close_value = len(goal)
    for char_index in range(len(goal)):
        if goal[char_index] == curr[char_index]:
            close_value -= 1
    return close_value
```

d)

Para a função de fitness basicamente inverti a função de avaliação, desta vez começo com o contador a zero e incremento cada vez que encontro um caracter igual ao goal.

Se encontrarmos o goal, o valor retornado será o mais alto possível, ou seja o valor do size do goal

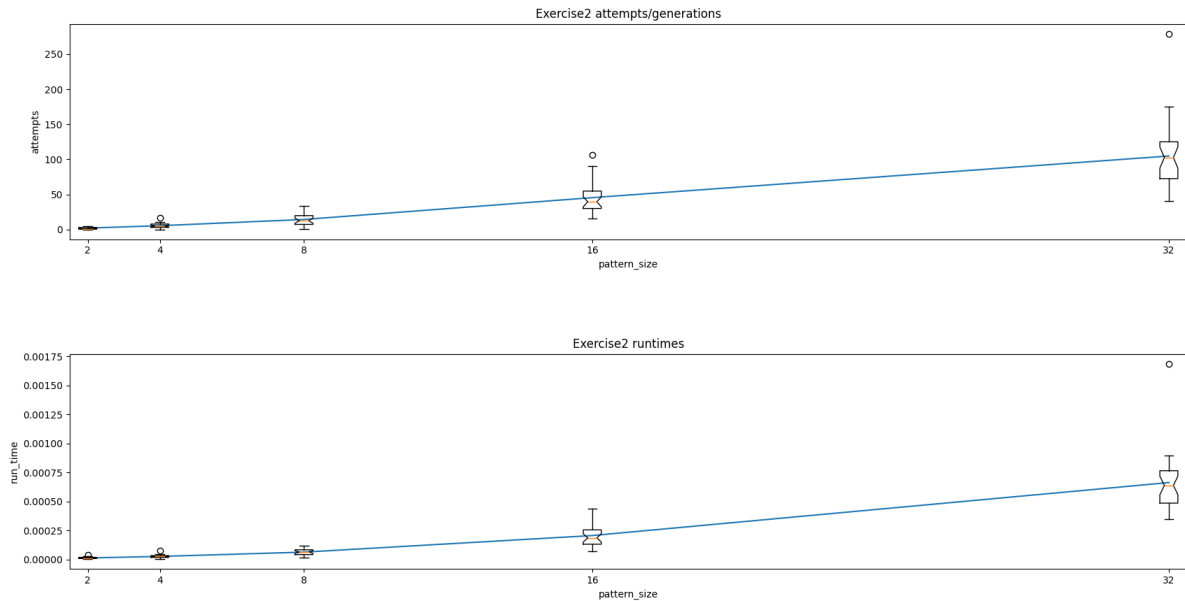
```
@staticmethod
def fitness(goal: str, curr: str):
    fitness_value = 0
    for char_index in range(len(goal)):
        if goal[char_index] == curr[char_index]:
            fitness_value += 1
    return fitness_value
```

Exercício 2

A função de mutação que criei é a seguinte, escolho um index para fazer o flip e depois baseado numa chance de 50% mudo de "0" para "1" ou de "1" para "0"

```
@staticmethod
def mutate(input: str) -> str:
    input_as_list = list(input)
    index = randint(0, len(input) - 1)
    if input_as_list[index] == "0":
        input_as_list[index] = "1"
    elif input_as_list[index] == "1":
        input_as_list[index] = "0"
    return "".join(input_as_list)
```

De seguida usei a função para gerar os resultados visíveis em baixo:



Cada vez que uma mutação tem uma fitness maior que a fitness do padrão anterior começo a usá-la como base para novas mutações, o ciclo repete-se até chegar ao goal pattern ou até exceder o tempo limite, como no exercício 1.

Podemos ver que aqui os resultados já melhoraram imensamente, passamos de a volta dos 50000 attempts para 100, é uma melhoria com um factor de 500x (50000/100)

Relativamente á pergunta colocada no enunciado: *“Does it always converge to the solution? If not, can you understand if there is one mutation of the last sequence that could result in a better pattern?”* segundo os meus testes o algoritmo converge sempre para a melhor solução, o que me leva a acreditar que implementei bem a função de fitness e evaluation, os únicos momentos onde o algoritmo não consegue chegar ao goal são quando ele atinge o tempo limite, para testar isto coloquei um tempo limite extremamente rigoroso(0.0001 segundos) e testei um padrão de 16bits. A ordem de padrões gerados foi a seguinte:

padrões gerados:

```
'1001000010100110',
'1001000110100110',
'1001000110100100',
'0001000110100100',
'0000000110100100',
'0000010110100100',
'000001110100100',
'000001110100101',
'000001110101101',
'001001110101101'
```

goal: '0010111100101101'

último '0010011110101101'

Como podemos ver, mesmo com este tempo limitado apenas 2 bits estavam errados no final, é apenas uma questão de tempo, mais algumas iterações e muito provavelmente o algoritmo teria chegado ao resultado.

Exercício 3

Population

Para o Exercício 3 e adiante decidi criar uma class que encapsulasse todo o comportamento de uma população de padrões.

```
_patterns_list: [str] = []
_fitness_list: [int] = []
_goal: str

def __init__(self, initial_goal: str, initial_patterns_list: [str] = None, initial_fitness_list: [int] = None, ):
    self._patterns_list = initial_patterns_list
    self._fitness_list = initial_fitness_list
    self._goal = initial_goal

def add_individual(self, pattern: str, fitness: int = None) -> None:
    if fitness is None:
        fitness = Mastermind.fitness(curr=pattern, goal=self._goal)
    self._patterns_list.append(pattern)
    self._fitness_list.append(fitness)
```

A estrutura básica de uma population é uma lista de padrões e as suas fitnesses e um padrão goal que a população tenta atingir.

Defini vários métodos para interagir com a classe, como vários getters (size, pattern e fitness values) e outros métodos, como `_get_mean_fitness`, `sort` e `extract_best_patterns`.

O `sort` organiza as listas da população pelos valores de fitness, como são duas listas separadas, tive o cuidado de nunca quebrar as ligações entre o padrão e a sua fitness correspondente.

O `extract_best_patterns` recebe como argumento uma percentagem de “indivíduos” a extrair e retira os melhores nessa percentagem, se chamarmos a função com o argumento 0.3 ela extrair os 30% melhores “indivíduos” da população.

Stagnation

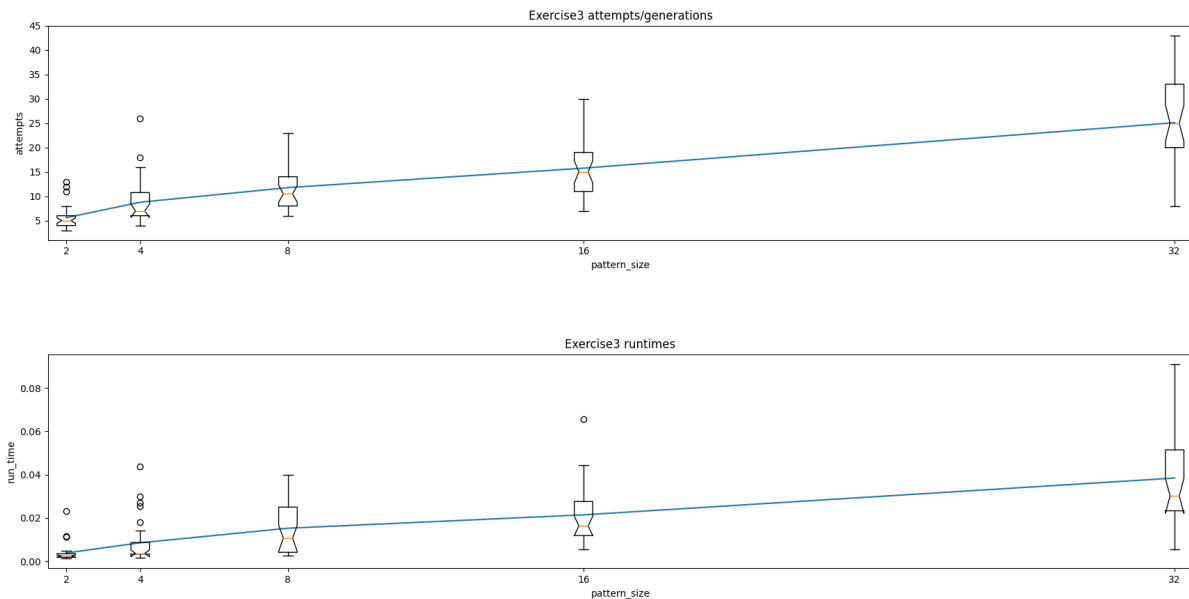
Defino se a população está estagnada ou não através da seguinte condição:

$$\begin{aligned} & _stagnated = \\ & _fitness_history[-1] + STAGNATION_VARIANCE \\ & > _current_population.get_mean_fitness() > \\ & _fitness_history[-1] - STAGNATION_VARIANCE \end{aligned}$$

Ou seja a fitness média da população atual tem de estar entre a fitness anterior mais a variância e a fitness anterior menos a variância que nos meus testes foi sempre igual a 0.025.

Resultados

Observando os gráficos podemos ver que também nenhum atinge o tempo limite de 10 segundos (ou seja todos tiveram sucesso) e que o número de tentativas diminuiu bastante. Por exemplo nos 32bits, baixou de 100 (ex2) para 30(ex3) o que se traduz numa melhoria de 3.33x (100/30).



Exercício 4

Crossover

O exercício 4 em relação ao terceiro não mudou muita coisa, apenas a introdução do método de crossover em vez do mutate.

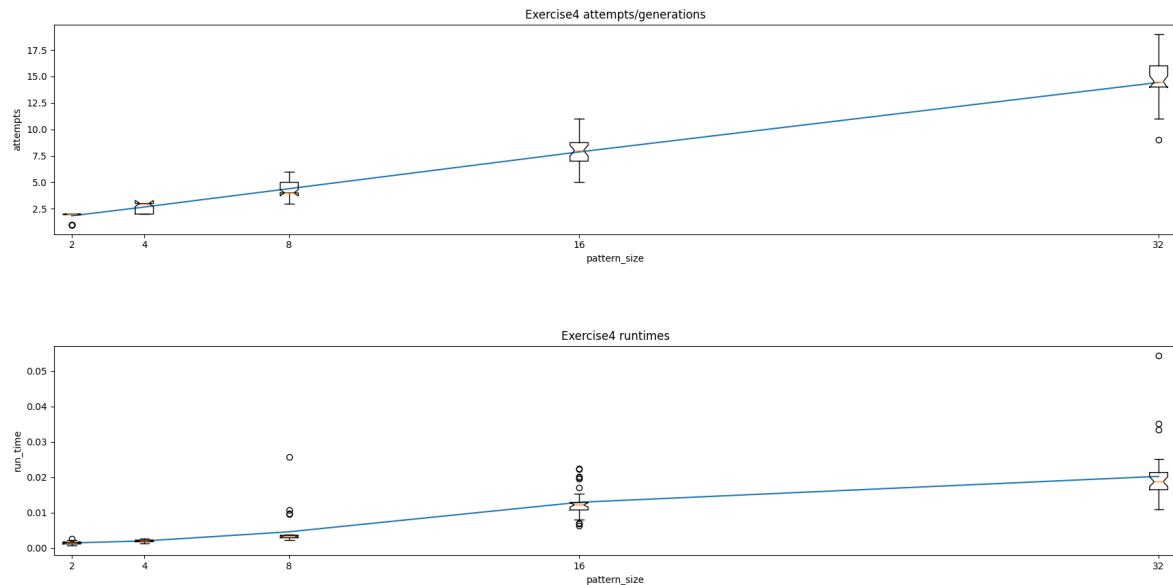
Implementei o crossover da seguinte maneira:

```
@staticmethod
def crossover(input_a: str, input_b: str) -> str:
    first_input = randint(0, 1)
    slice_index = randint(0, min(len(input_a), len(input_b)))
    if first_input == 0:
        return input_a[0:slice_index] + input_b[slice_index:len(input_b)]
    else:
        return input_b[0:slice_index] + input_a[slice_index:len(input_a)]
```

Primeiro calculei um index aleatório para fazer a divisão, aqui já em preparação para os exercícios opcionais tenho em conta o tamanho dos dois inputs e escolho apenas dentro do limite do mais pequeno. Depois ainda introduzir outra autoridade, quem é o “pai” que vai á frente, ou seja, as combinações podem ser A:B ou B:A onde A e B representam metades de cada um dos pais.

Resultados

Mais uma vez temos uma melhoria, nos 32bits antes tínhamos 30 tentativas, agora temos a volta de 15. Melhoria de 2x (30/15).



Exercício 5

No exercício 5 tenho de pensar que alterações seriam necessárias para implementar os exercícios anteriores mas com padrões com tamanhos desconhecidos.

Primeiro tenho de começar a gerar padrões com size aleatório o que é bastante simples, basta chamar a minha função anterior mas utilizando `random.randint` no argumento do tamanho do padrão.

Evaluate and Fitness

Segundo tive de mudar as funções de evaluate e fitness da seguinte forma:

```
def evaluate_undefined_size(goal: str, curr: str) -> int:
    _dif_counter: int = max(len(goal), len(curr))
    for charindex in range(min(len(curr), len(goal))):
        if curr[charindex] == goal[charindex]:
            _dif_counter -= 1
    return _dif_counter
```

Agora a função de evaluate começa com um contador do tamanho do maior input e vai contando para baixo cada vez que encontra um carácter igual.

```
def fitness_undefined_size(goal: str, curr: str) -> float:
    _max_size: int = max(len(goal), len(curr))
    _equal_counter: int = 0
    for charindex in range(min(len(curr), len(goal))):
        if curr[charindex] == goal[charindex]:
            _equal_counter += 1
    return _equal_counter / _max_size
```

Em relação às mudanças da fitness, passei a representá-la como um valor de entre 0 e 1, para isso comecei com um contador a 0 como antes e aumento-o sempre que encontro um caracter igual mas agora retorno a divisão deste contador pelo tamanho do padrão maior, atendendo assim a ambas as situações, quando o goal é maior e quando o goal é menor que a estimativa.

Fiz alguns testes para avaliar os valores que estas funções retornam:

```
-----evaluate_undefined_size_test-----
goal=110,    curr=110    0
goal=1101,   curr=110    1
goal=11011,  curr=110    2
goal=110111, curr=110    3
goal=110111, curr=11     4
goal=1,      curr=110    2
goal=1,      curr=11011  4
goal=1,      curr=1      0
-----fitness_undefined_size-----
goal=110,    curr=110    1.0
goal=1101,   curr=110    0.75
goal=11011,  curr=110    0.6
goal=110111, curr=110    0.5
goal=1,      curr=110    0.3333333333333333
goal=1,      curr=11011  0.2
goal=1,      curr=1      1.0
goal=1,      curr=01     0.0
```

Mutation

Para a função de mutation tive de introduzir duas novas situações, aquela onde adicionamos um novo valor e aquela em que removemos um valor do padrão para além de manter a situação inicial, onde a length permanece a mesma.

Na situação de remoção, escolho um index aleatório e removo-o da lista de caracteres do padrão

Na situação de adição, escolho um index aleatório como na remoção mas agora gero um bit aleatório e insiro-o na lista, no index aleatório calculado anteriormente.

A situação de permanecer na mesma length continua como antes, não necessita de alterações.

```

def mutate_undefined_size(mutation_input: str) -> str:
    input_as_list = list(mutation_input)
    # print(input_as_list)
    same_add_remove: int = randint(0, 3)
    if same_add_remove == 1:
        # same_length
        index = randint(0, len(mutation_input) - 1)
        print("same_length in ", index)
        if input_as_list[index] == "0":
            input_as_list[index] = "1"
        elif input_as_list[index] == "1":
            input_as_list[index] = "0"
    elif same_add_remove == 2:
        # remove_bit
        index = randint(0, len(mutation_input) - 1)
        print("remove_bit in", index)
        input_as_list.pop(index)
    else:
        # add_bit
        index = randint(0, len(mutation_input))
        _new_bit: str = Mastermind.random_bit()
        print("added_bit", _new_bit, " in", index)
        input_as_list.insert(index, _new_bit)
    # print(input_as_list)
    return "".join(input_as_list)

```

Para a função de mutate também fiz um teste

-----mutate_undefined_size_test-----

```

added_bit 1 in 0      : 000 1000
added_bit 1 in 0      : 1 11
remove_bit in 2       : 010 01
remove_bit in 1       : 00 0
same_length in 1      : 000 010
remove_bit in 3       : 1011 101
added_bit 1 in 1      : 011 0111
same_length in 1      : 11 10
same_length in 2      : 0001 0011
added_bit 1 in 3      : 010 0101

```

Crossover

```
def crossover_undefined_size(input_a: str, input_b: str) -> str:
    first_input = randint(0, 1)
    slice_index = randint(0, min(len(input_a), len(input_b)))
    if first_input == 0:
        return input_a[0:slice_index] + input_b[slice_index:len(input_b)]
    else:
        return input_b[0:slice_index] + input_a[slice_index:len(input_a)]
```

O crossover, já tinha implementado com isto em mente mas basicamente o unico cuidado que ele precisa ter é na escolha do index para fazer o slice, não pode ser maior que nenhum dos comprimentos dos inputs.

Podemos ver vários outputs desta função com o teste que efetuei:

-----crossover_undefined_size_test-----

| pai1 | pai2 | resultado |
|------|------|-----------|
| 111 | 0111 | 011 |
| 10 | 0 | 1 |
| 000 | 011 | 011 |
| 1001 | 01 | 10 |
| 101 | 10 | 10 |
| 0 | 10 | 1 |
| 00 | 010 | 010 |
| 0110 | 10 | 10 |
| 01 | 11 | 01 |
| 101 | 0 | 001 |

Exercício 6

Generation

```
def random_decimal_pattern(size: int) -> str:
    result: str = ""
    for _ in range(size):
        result += random_decimal()
    return result

def random_decimal() -> str:
    numbers: [chr] = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
    return random.choice(numbers)
```

Primeiro tive de mudar como gero os padrões e os “bits” que agora são números decimais, para isso estou a usar o random.choice com uma lista de todos os números decimais em formato char

Evaluate and Fitness

Para estas funções, inicialmente pensei em simplesmente fazer cast para numeros inteiros e subtraí-los para obter um valor tangível da sua diferença mas rapidamente reparei que isso iria colocar um peso muito grande nos numeros mais significativos, sendo que esse peso não existe em mais nenhum sitio do algoritmo, mudar um bit é igual tanto em casas mais significativas ou menos, portanto decidi uma “approach” diferente onde analisava cada algarismo do padrão e comparava-o o com o algarismo na mesma posição do padrão objetivo.

Tive de alterar ambas as funções porque apenas verificar quando é que caracter é igual já não chega, não dá informação suficiente, decidi iterar por todos os caracteres como antes mas agora somo ao counter o valor absoluto da subtração entre o goal e o current guess. Na função de evaluate começo com o counter no valor absoluto da diferença entre o tamanho do goal e do current para ter em conta tamanhos diferentes e por cada caracter que esteja a mais ou a menos adicionar um ponto ao contador.

```
def evaluate_decimal(goal: str, curr: str) -> int:
    _dif_counter: int = abs(len(goal)-len(curr))
    for charindex in range(min(len(curr), len(goal))):
        _goal_int: int = int(goal[charindex])
        _current_int: int = int(curr[charindex])
        _dif_counter += abs(_goal_int - _current_int)
    return _dif_counter
```

Podemos observar alguns outputs da função evaluate no seguinte teste:

```
-----evaluate_decimal_test-----
goal= 91      _current= 177      evaluation= 15
goal= 6317    _current= 6       evaluation= 3
goal= 9074    _current= 91      evaluation= 3
goal= 000     _current= 6       evaluation= 8
goal= 60      _current= 77      evaluation= 8
goal= 01231   _current= 01231   evaluation= 0
goal= 01231   _current= 01233   evaluation= 2
goal= 01231   _current= 012339  evaluation= 3
goal= 012     _current= 012339  evaluation= 3
goal= 012     _current= 012339  evaluation= 3
goal= 012     _current= 019339  evaluation= 10
```

Na função de fitness começo com o contador com o valor máximo da diferença entre o goal e o current vezes o numero de caracteres minimo, ou seja, se o goal for 123 e o curr for 1234 o counter vai começar com o valor $9 \times 3 = 27$, isto porque estou a subtrair a este valor o absoluto da diferença entre cada caracter do goal e do current.

Decidi manter a filosofia de dividir por um valor máximo para normalizar os resultados entre 0 e 1, neste caso dividido por 9* o comprimento máximo entre os dois, se dividisse pelo comprimento mínimo (como estou a usar esse valor para o counter) não teria em conta guesses com comprimento maior que o goal.

```
def fitness_decimal(goal: str, curr: str) -> float:
    _max_diff: int = 9 * max(len(goal), len(curr))
    _dif_counter: int = 9 * min(len(goal), len(curr))

    for charindex in range(min(len(curr), len(goal))):
        _goal_int: int = int(goal[charindex])
        _current_int: int = int(curr[charindex])
        _dif_counter -= abs(_goal_int - _current_int)
    return _dif_counter / _max_diff
```

Podemos observar alguns outputs da função fitness no seguinte teste:

```
-----fitness_decimal-----
goal= 53      _current= 74      fitness= 0.8333333333333334
goal= 6       _current= 2       fitness= 0.5555555555555556
goal= 158     _current= 8344     fitness= 0.3888888888888889
goal= 8690    _current= 3662     fitness= 0.7222222222222222
goal= 851     _current= 8128     fitness= 0.6111111111111112
goal= 5707    _current= 4       fitness= 0.2222222222222222
goal= 2283    _current= 3       fitness= 0.2222222222222222
goal= 68      _current= 957     fitness= 0.4444444444444444
goal= 890     _current= 8288     fitness= 0.3333333333333333
goal= 60      _current= 5983     fitness= 0.2222222222222222
goal= 01231   _current= 01231     fitness= 1.0
goal= 01231   _current= 012315   fitness= 0.8333333333333334
goal= 01231   _current= 01233     fitness= 0.9555555555555556
goal= 01231   _current= 012330    fitness= 0.7962962962962963
goal= 01231   _current= 0123      fitness= 0.8
```

Mutation

As alterações à função de mutação são relativamente simples

```

def mutate_decimal_size(mutation_input: str) -> str:
    input_as_list = list(mutation_input)
    # print(input_as_list)
    same_add_remove: int = random.choice([1, 2, 3])
    if same_add_remove == 1:
        # same_length
        index = randint(0, len(mutation_input) - 1)
        print("same_length in ", index)
        add_or_subtract = random.choice([True, False])
        int_index_value: int = int(input_as_list[index])
        if add_or_subtract:
            input_as_list[index] = str(int_index_value + 1)
        else:
            input_as_list[index] = str(int_index_value - 1)

    elif same_add_remove == 2:
        # remove_bit
        index = randint(0, len(mutation_input) - 1)
        print("remove_bit in", index)
        input_as_list.pop(index)
    else:
        # add_bit
        index = randint(0, len(mutation_input))
        _new_bit: str = random_decimal()
        print("added_bit", _new_bit, " in", index)
        input_as_list.insert(index, _new_bit)
    # print(input_as_list)
    return "".join(input_as_list)

```

Tive como base a função que criei no exercício 5, assim este exercício 6 pode atender às duas situações, tamanho variável e padrões com decimais.

Aqui as únicas alterações foram quando gero mutação com o mesmo size e quando adiciono um valor novo.

Quando é o mesmo size, decidi implementar fazendo cast para inteiro de um valor aleatório do padrão e aleatoriamente aumentá-lo ou diminuí-lo por 1 e voltar a inseri-lo no padrão, na mesma posição.

Quando é adicionar novo valor, a única alteração é que o novo valor a adicionar tem de usar a nova função de geração de “bit” específica deste exercício, que apenas retorna um número decimal convertido para char em vez de 0 ou 1 convertidos para char, como nos exercícios obrigatórios

Em baixo pode ver alguns dos valores que esta função retorna:

-----mutate_decimal_test-----

remove_bit in 1: 867 87

added_bit 7 in 2: 508 5078

added_bit 8 in 1: 903 9803

added_bit 0 in 2: 921 9201


```
remove_bit in 0: 110 10
same_length in 2: 434 435
same_length in 0: 541 441
added_bit 7 in 2: 824 8274
same_length in 0: 577 477
same_length in 1: 656 666
```

Crossover

A função de crossover não recebeu qualquer alteração

```
def crossover_decimal(input_a: str, input_b: str) -> str:
    first_input = random.choice([True, False])
    slice_index = randint(0, min(len(input_a), len(input_b)))
    if first_input:
        return input_a[0:slice_index+1] + input_b[slice_index-1:len(input_b)]
    else:
        return input_b[0:slice_index+1] + input_a[slice_index-1:len(input_a)]
```

Alguns padrões gerados por crossover:

-----crossover_decimal_test-----

```
839 0306 86
78 8387 78387
6 604 64
4 1 14
69 208 6908
97 98 9798
51 96 5196
6 870 876
023 925 92023
168 878 18
```

Exercício 7

No problema do robô no labirinto, a função de update da QMatrix é parecida com uma espécie de função de avaliação, a diferença é que esta função de avaliação receberia um caminho que o robô seguiria e retornaria um valor baseado, por exemplo no comprimento do caminho, para que caminhos mais curtos fossem melhor avaliados, e caminhos que passassem em células de qualidade maior também fossem melhores.