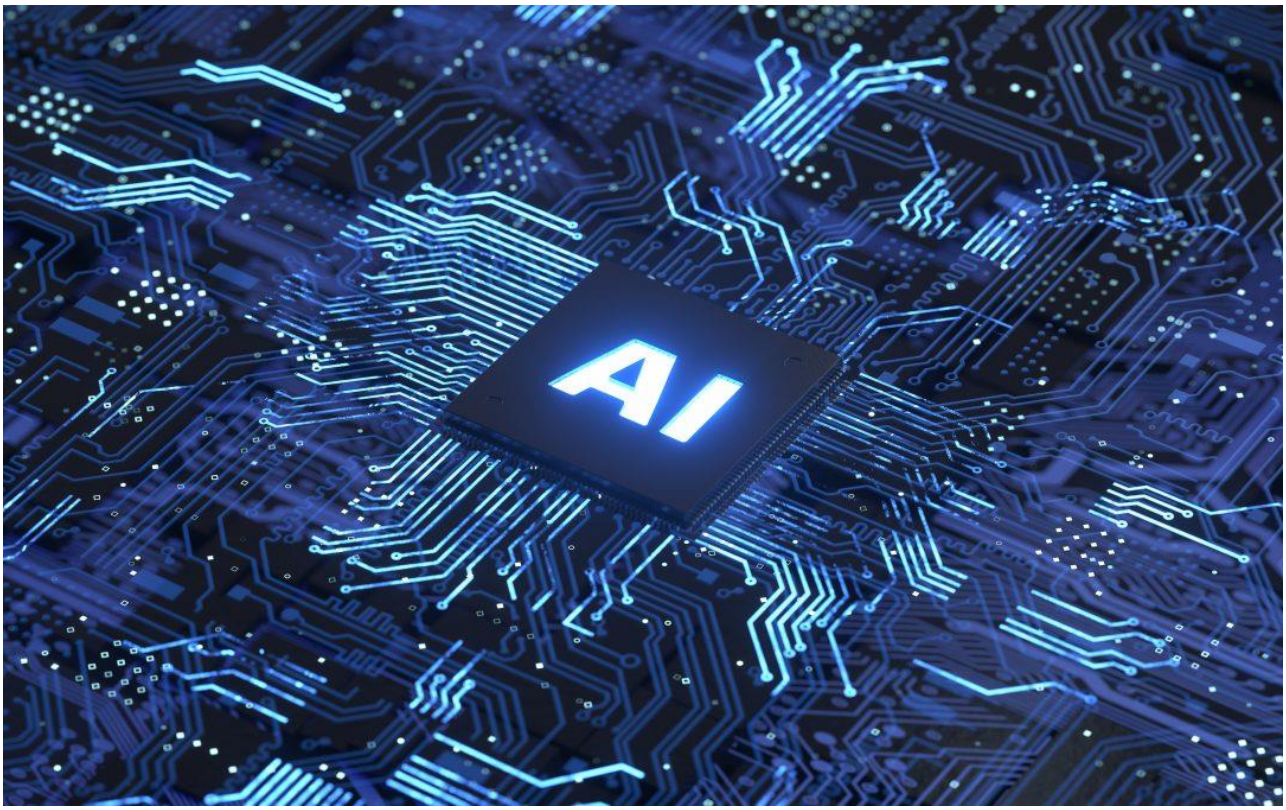
INTRODUCTION TO MACHINE LEARNING – UNSUPERVISED LEARNING

21 DE DEZEMBRO DE 2020



Nome: Miguel Oliveira

Aluno nº: 96200

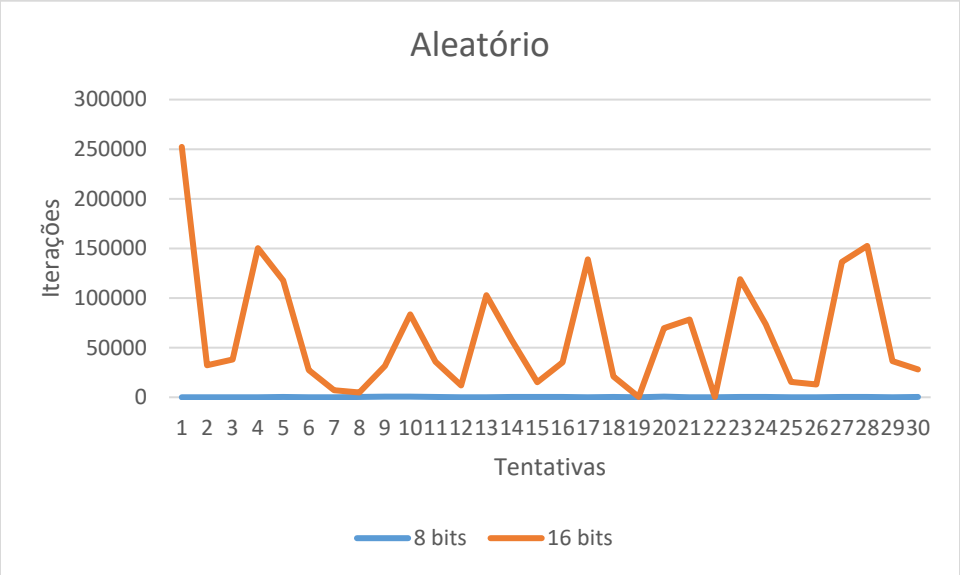
Email: mdlco@iscte-iul.pt

Docente: Professor Doutor Luís Nunes

Docente: Professor Doutor Sancho Oliveira

Exercícios 1 e 2:

Foi criada uma função que dado um número de bits, por exemplo 8 bits, produz um padrão aleatório desse tamanho e medido quantas vezes levava esse formato aleatório a atingir um resultado igual ao padrão previamente gerado.



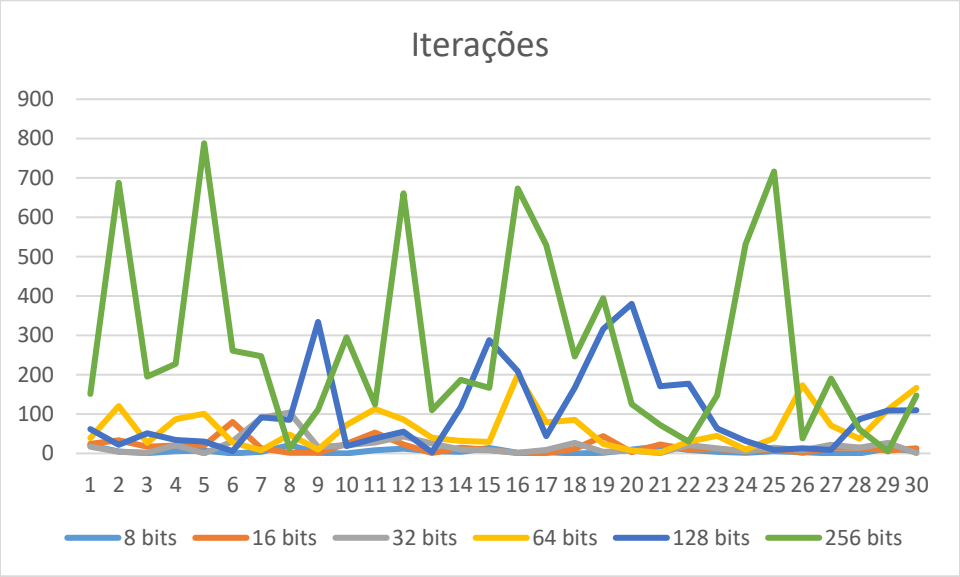
	8 bits		*16 bits*	
	Attempts	**Run Times**	**Attempts**	**Run Times**
Average	199,7	62896,06667	0,000738	0,41350844
Std. Deviation	184,0686738	58957,48198	0,000663819	0,390310346

Para analisar o desempenho deste procedimento testou-se com diferentes tamanhos de bits, 8 bits e 16 bits, contudo números superiores como 32 bits tornavam o processo de computação demasiado lento e que se pode concluir que para tamanhos elevados o tempo de computação bem como o número de tentativas até atingir o resultado é enorme caso se recorre apenas à "sorte" para encontrar o que se pretende.

Exercício 3:

Foi criada uma função de avaliação que compara cada bit de um dado padrão com os bits correspondentes do padrão que se está a procurar, contanto quantos são iguais, no fim desta avaliação atribui uma pontuação ao padrão que estava a ser analisado entre 0 e 1.

Assim, usando esta avaliação desenvolveu-se um simples algoritmo para aplica-la. É gerado aleatoriamente a cada iteração do algoritmo um novo padrão, em seguida atribui-se uma pontuação recorrendo à função de avaliação, se esta pontuação for superior à pontuação obtida na iteração anterior este novo padrão substitui a iteração antiga, caso contrário repete o processo até que o novo padrão obtenha uma pontuação de 1 (pontuação ótima) e seja igual ao padrão que se pretende igualar.



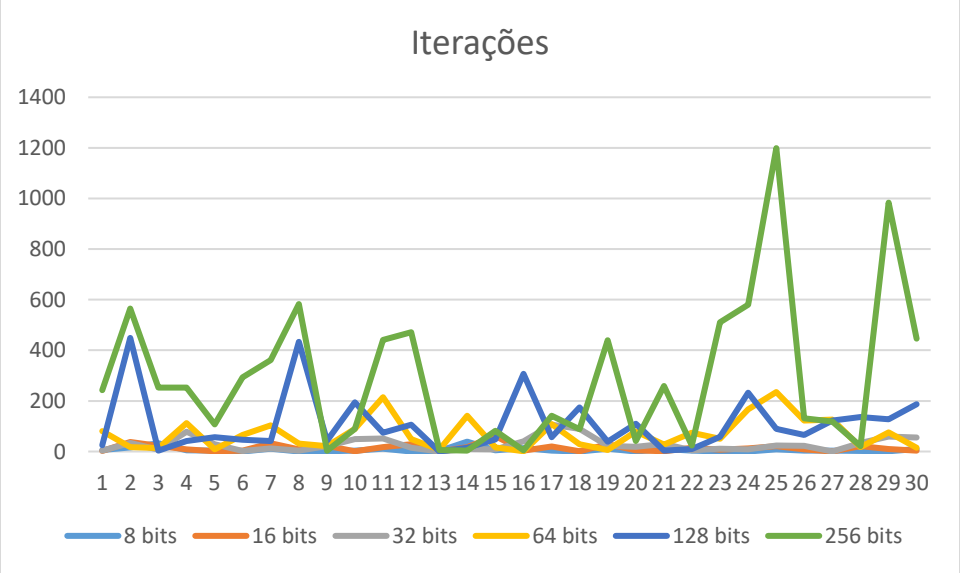
	8 bits		*16 bits*		*32 bits*		*64 bits*		*128 bits*		*256 bits*	
	Attempts	Run Times	Attempts	Run Times	Attempts	Run Times	Attempts	Run Times	Attempts	Run Times	Attempts	Run Times
Average	25	0.0001	139	0,0006	517	0,003	1893	0,013	9502	0,108	29060	0,60
Std. Deviation	19	0,00008	106	0,0004	411	0,0002	1390	0,009	5912	0,07	19843	0,42

Para analisar o desempenho do algoritmo e comparar com a abordagem aleatória do exercício 2, foi medido o número de tentativas necessárias até atingir o objetivo te verificou-se uma melhoria drástica em relação ao formata de teste anterior que apenas recorria à “sorte” na sua busca, conseguindo com este novo algoritmo não ultrapassar as 1000 tentativas mesmo com um tamanho de 256 bits, algo que para o formato anterior um tamanho de 16 bits já ultrapassava estas 1000 tentativas por largas margens.

Exercício 4:

Para este caso foi criada uma simples função de avaliação. Esta função conta o número de “0” de um padrão e de acordo com o número e o tamanho do padrão atribui uma pontuação. Assim o caso ótimo desta avaliação seria um padrão em que todos os seus bits fossem “0” e é uma função de avaliação cujos os padrões que se tentam alcançar são facilmente percetíveis à primeira vista.

Assim, para avaliar o desempenho desta nova função, aplicou-se o mesmo simples algoritmo do exercício anterior e obteve-se os seguintes resultados:



Verificou-se que com esta função de avaliação o algoritmo consegue encontrar o padrão pretendido mais rapidamente em relação à função de avaliação usada no exercício anterior.

	8 bits		*16 bits*		*32 bits*		*64 bits*		*128 bits*		*256 bits*	
	Attempts	Run Times	Attempts	Run Times	Attempts	Run Times	Attempts	Run Times	Attempts	Run Times	Attempts	Run Times
Average	29	0.0001	130	0,0005	676	0,003	2033	0,013	8032	0,07	37546	0,54
Std. Deviation	19,8	0,00008	90,8	0,0004	450	0,002	1245	0,008	4819	0,04	49517	0,7

Exercício 5:

Foi implementada uma função que altera um bit aleatório de um dado padrão e aplicado um algoritmo que atribui segundo a função de avaliação desenvolvida no exercício anterior um dado padrão. Assim apenas se consideravam aceites mutações que melhorassem a pontuação do padrão. Com isto concluiu-se que após 1000 mutações, para os tamanhos 8, 16, 32, 64 e 128 bits era possível alcançar o padrão esperado usando este sistema, contudo para padrões com tamanho de 256 bits isso não aconteceu.

Analisando o resultado final do padrão obtido para o tamanho de 256 bits concluiu-se melhor pontuação alcançada em 1000 mutações foi 97.66%, ou seja, dos 256 bits, 5 bits encontravam-se ainda errados, podendo ser ainda necessárias pelo menos mais 2 mutações para alcançar o resultado/padrão esperado.

Exercício 6:

Para implementar a lógica e o comportamento de um algoritmo evolucionário foi gerada aleatoriamente uma população inicial, ou seja, um conjunto de 100 padrões todos eles aleatórios e cada um deles avaliado de modo a terem todos uma classificação inicial. Dos 100 elementos da população apenas os 30% melhores foram mantidos e desses 30% gerada a geração seguinte.

Para repor os 100 elementos da população a partir dos 30% melhores da geração anterior foram automaticamente adicionados os 30% melhores à população seguinte e os restantes 70% gerados percorrendo a lista dos 30% melhores duas vezes e em cada elemento era alterado 1 bit

aleatoriamente (método usado em exercícios anteriores para introduzir novas características na população) conseguindo gerar assim os “filhos” dos melhores 30%. Optou-se por deixar sempre na geração seguinte inalterados os 30% melhores da geração anterior, pois estes podem ter características (e assim avaliações) melhores do que a sua mutação iria ter.

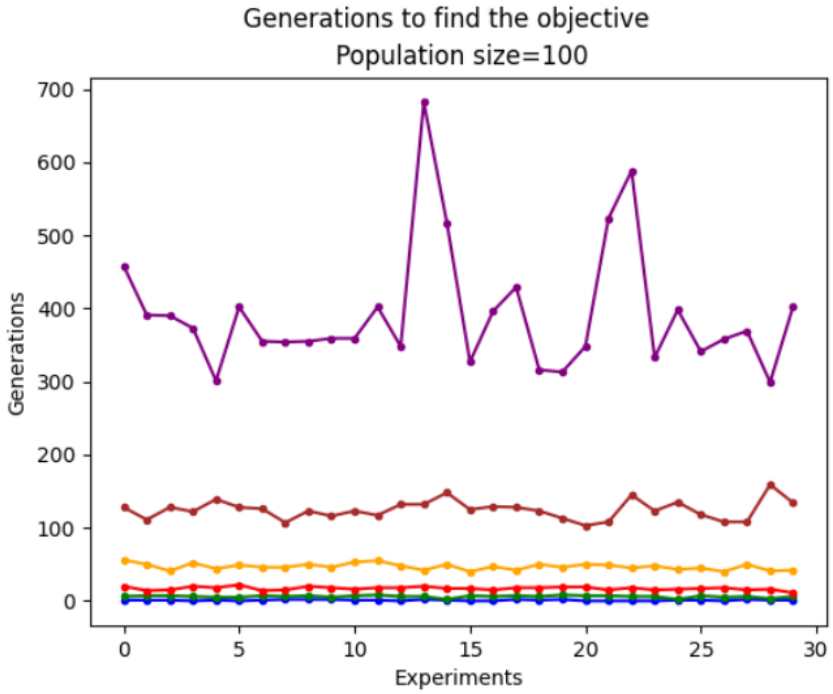
Após obtida a nova geração eram posteriormente avaliados todos os elementos e processo repetia-se até que fosse encontrado pelo menos um elemento da população em vigor que correspondesse ao padrão desejado.

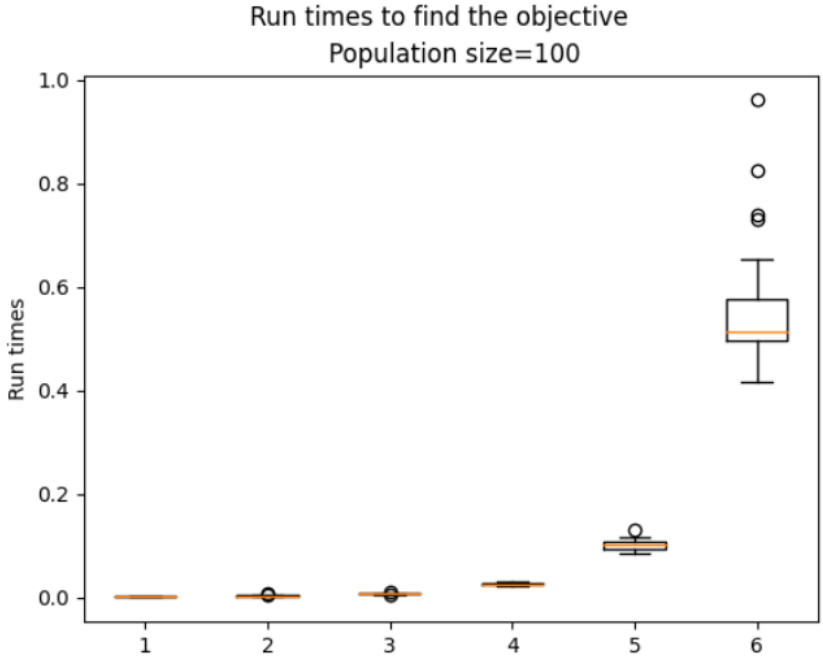
Com este novo método de procura é possível ter uma mais vasta e acertada avaliação e seleção dos melhores “espécimes” gerados, pois ao contrário de procurar apenas um padrão mais favorável que o anterior consegue selecionar-se uma “elite” de cada geração e aprimorar a seguinte com os melhores da anterior.

Isto foi possível de se verificar pelos resultados obtidos, por exemplo, entre o exercício 4 e o exercício 6, sendo o exercício 4, onde se introduz a função de avaliação e no exercício 6, onde se introduz um conceito de “evolução”.

Pelos dados e observando, por exemplo, para o caso de padrões com 256 bits, verifica-se uma melhoria no número de tentativas necessárias para alcançar o padrão desejado, ainda que, para o exercício 6 estejamos a falar de gerações. Constata-se que o exercício 4 atingiu uma média de 37.546 tentativas e um máximo de 279.924 das mesmas, enquanto que, para o exercício 6, a média foi de 392,9 gerações e um máximo de 686. Analisando os tempos de execução necessários para se obterem estes resultados, denota-se que para o exercício 4 foram 0,54 segundos em média comparados aos 0,52 segundos do exercício 6.

Assim os métodos de pesquisa aplicados no exercício 4 são mais lentos e menos eficientes do que os usados no exercício 6, contudo, é importante notar que são exercícios consideravelmente diferentes, em que um, ainda consiste numa pesquisa por “força bruta” e o outro uma pesquisa mais “inteligente” aplicando conceitos base da evolução, tornando estes dois métodos difíceis de comparar com precisão, mesmo que estejam ambos a tentar convergir nos mesmos objetivos.


Generations to find the objective
Population size=100
Generations
Experiments
0
100
200
300
400
500
600
700
0
5
10
15
20
25
30


Run times to find the objective
Population size=100
Run times
0.0
0.2
0.4
0.6
0.8
1.0
1
2
3
4
5
6

Exercício 7:

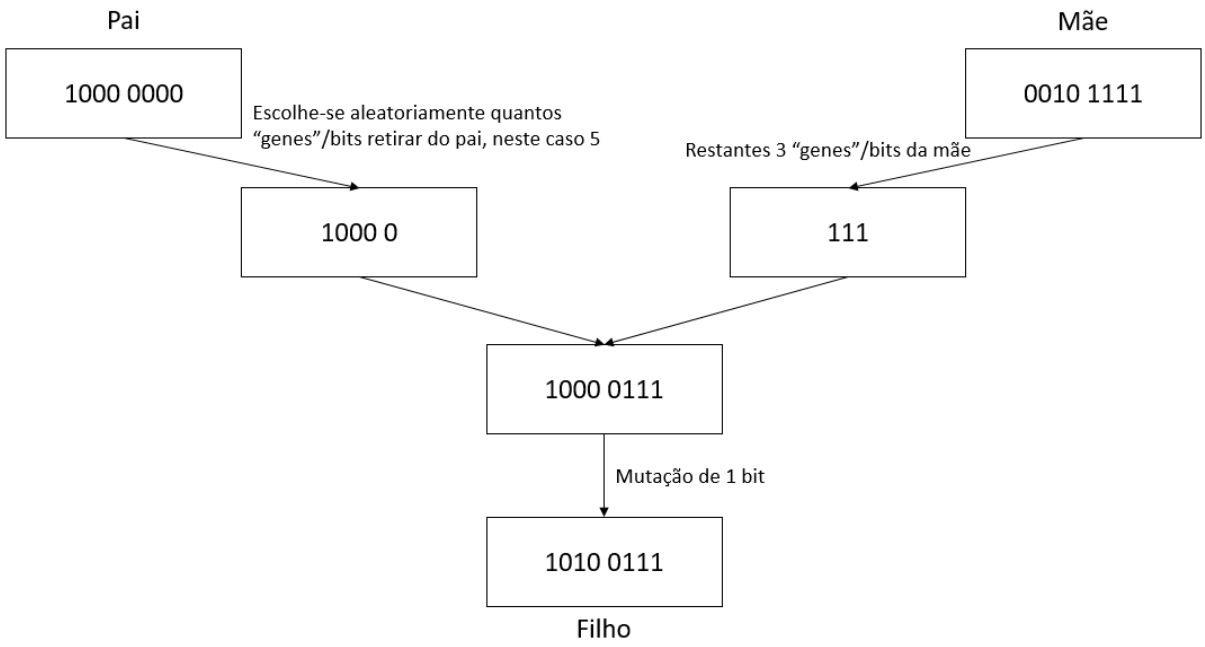
Para implementar a lógica e o comportamento de um algoritmo evolucionário foi gerada aleatoriamente uma população inicial, ou seja, um conjunto de 100 padrões todos eles aleatórios e cada um deles avaliado de modo a terem todos uma classificação inicial. Dos 100 elementos da população apenas os 30% melhores foram mantidos e desses 30% gerada a geração seguinte.

Para repor os 100 elementos da população a partir dos 30% melhores da geração anterior foram automaticamente adicionados os 30% melhores à população seguinte e os restantes 70% gerados percorrendo a lista dos 30% melhores duas vezes e em cada elemento recorria-se a um método de *crossover* com o "pai" imediatamente seguinte assim gerando o "filho".

Foram implementados dois métodos de *crossover* diferentes, obtendo-se diferentes resultados com cada um, ainda que ambos não divergissem por muito. Contudo para efeitos de análise de resultados apenas o método de *crossover* com melhores resultados foi considerado.

O método de crossover escolhido funciona da seguinte maneira: Após escolhidos dois padrões diferentes como sendo os "pais", escolhe aleatoriamente quantos "genes" (neste caso bits) irá usar do "pai" e retira o restante número de "genes" da "mãe", por fim aplica uma mutação de 1 "gene" / bit. Por exemplo, dada um padrão de 8 bits é aleatoriamente determinado que irão ser usados 5 "genes" /bits do "pai", ou seja, para gerar o filho, irão ser necessário ir buscar os restantes 3 "genes"/bits da "mãe", assim, o filho será constituído pelos 5 primeiros "genes"/bits do "pai" e os últimos 3 "genes"/bits da "mãe", para finalizar é aplicada a mutação aleatória de 1 "gene" / bit .

Em termos práticos a transformação teria o seguinte aspeto:



Após obtida a nova geração eram posteriormente avaliados todos os elementos e processo repetia-se até que fosse encontrado pelo menos um elemento da população em vigor que correspondesse ao padrão desejado.

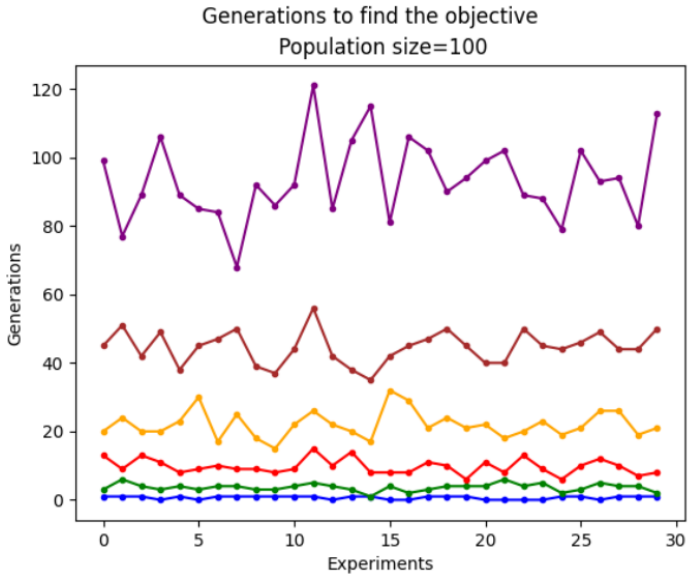
Usando agora um novo método de gerar novas gerações recorrendo ao conceito de *crossover*, permitiu que fosse possível ter uma melhor continuidade dos "genes" /bits dos melhores elementos de cada geração, conseguindo através das mutações continuar a introduzir variedade nas gerações seguintes algo que não se verificava no exercício anterior, pois não estava a tirar total proveito do top 30% de cada geração.

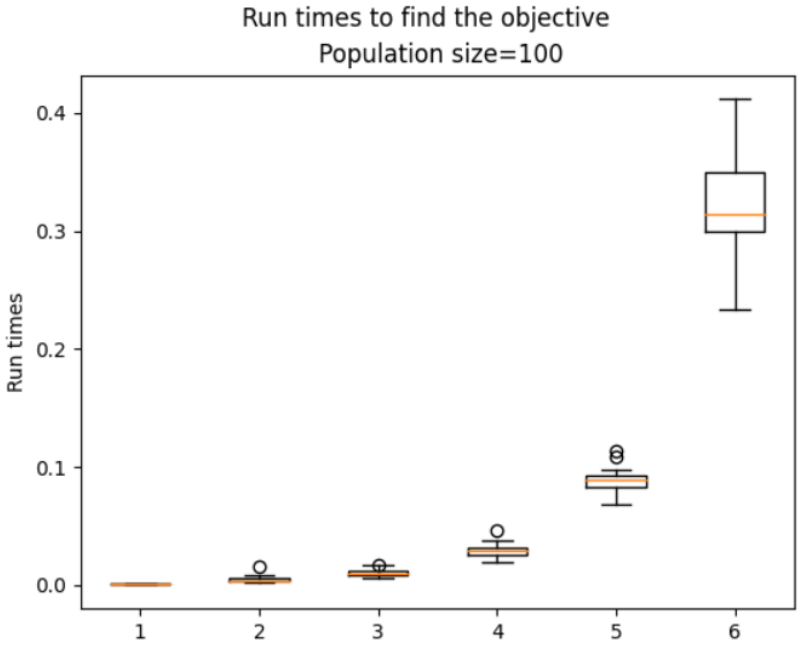
Isto foi possível de se verificar pelos resultados obtidos, comparando, entre o exercício 6 e o exercício 7, sendo o exercício 6, onde se introduz um conceito de "evolução" e o exercício 7 onde se introduz o conceito de *crossover*.

Pelos dados e observando, por exemplo, para o caso de padrões com 256 bits, verifica-se uma melhoria no número de tentativas necessárias para alcançar o padrão desejado, ainda que, a escolha do método de crossover que foi aplicado possa alterar estes resultados como tinha sido mencionado anteriormente. Constata-se que o exercício 7 atingiu uma média de 93,5 gerações e um máximo de 121 das mesmas, enquanto que, para o exercício 6, a média foi de 392,9 gerações e um máximo de 686. Analisando os tempos de execução necessários para se obterem estes resultados, denota-se que para o exercício 7 foram 0,30 segundos em média comparados aos 0,52 segundos do exercício 6.

Assim conclui-se que a introdução do crossover no algoritmo melhorou a capacidade de gerar uma nova população tirando mais proveito do melhor de cada geração. Para além disso verifica-se a implementação de um algoritmo que tenta replicar o comportamento e lógica da

evolução demonstra ser algo capaz de aprender a cada iteração, conseguindo cada vez melhores resultados.





Exercício 8:

Para lidar com casos em que o tamanho dos padrões seria desconhecidos, a função de avaliação do algoritmo permaneceria igual, pois como consiste apenas em contar o número de “0” não necessita de saber o tamanho do padrão. A função de mutação também permaneceria inalterada pois como consiste apenas em trocar 1 bit aleatório de um dado padrão.

Exercício 9:

Para lidar com casos em que o tamanho dos padrões seria desconhecidos, a função de avaliação do algoritmo permaneceria igual, pois como consiste apenas em contar o número de “0” não necessita de saber o tamanho do padrão. A função de mutação também permaneceria inalterada pois como consiste apenas em trocar 1 bit aleatório de um dado padrão.