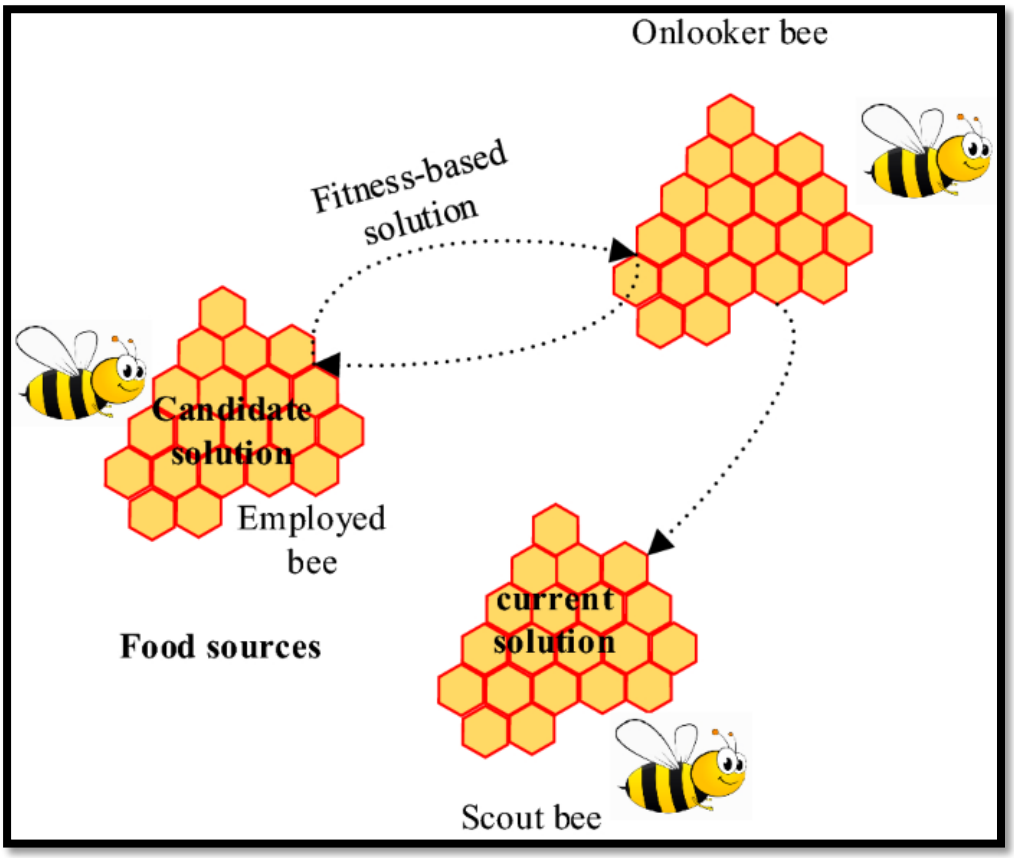
Departamento de Engenharia Informática e de Sistemas

Instituto Superior de Engenharia de Coimbra

Licenciatura em Engenharia Informática

Inteligência Computacional 2022/2023

Projeto Fase – 2



Emanuel Saraiva | 2019130219 | a2019130219@isec.pt | P1

Índice

2. Em que consiste a Computação Evolucionária?

A Computação Evolucionária (CE) é nada mais nada menos que "uma área da ciência da computação que usa ideias da evolução biológica para resolver problemas computacionais" (Mitchell; Taylor, 1999).

O processo evolucionário do algoritmo é descrito da seguinte maneira:

O primeiro componente do algoritmo consiste em população de possíveis soluções com mutações diferentes. Essas mutações, normalmente expressas em bits, são os genes nesse processo, onde cada indivíduo na população, é um cromossomo. Assim, um exemplo de um cromossomo seria 000000, em que cada 0 é um gene ou um bit. Um segundo seria 111111, um terceiro seria 111001, ou seja, cada indivíduo na população terá um código diferente, um gene diferente, sendo o tamanho da população igual à combinação de cada bit com o tamanho da variável que armazena esses bits. Cada cromossomo é então avaliado pela sua capacidade de resolver o problema ou ao menos de chegar mais perto da solução. Assim, o cromossomo que está mais apto a providenciar uma melhor solução ao problema computacional sobrevive.

Estes cromossomos sobreviventes são então selecionados e podem reproduzir-se. A próxima geração irá recombinar os genes dos cromossomos sobreviventes, de modo a "carregar" os genes dos cromossomos antigos. Por fim, esta nova geração de cromossomos sofrerá mutações *random* que irão resolver o problema de forma mais eficiente.

Quando não há muita divergência entre os descendentes produzidos e os novos, então temos um conjunto de soluções possíveis ao nosso problema.

No entanto, é importante salientar que, embora a computação evolucionária e toda a aprendizagem das máquinas sejam impressionantes, não existe até ao momento, máquinas que verdadeiramente evoluem. Podemos dizer que a computação evolucionária é o uso de matemática e estatística para a resolução de problemas, onde a sua principal utilização seja talvez para criar redes neuronais e inteligência artificial.

Para a resolução de problemas de otimização, os Algoritmos Genéticos (AG) proporcionam a abordagem mais natural para a sua solução.

Os AG são métodos de otimização que pertencem à CE, os quais usam mecanismos inspirados na teoria da evolução das espécies para a resolução de problemas de otimização.

As Redes Neuronais Convolucionais (CNN) usam estes AG para a identificação de objetos em imagens, destacando-se assim para aplicações em visão computacional.

3. O que é Inteligência Swarm?

A **Inteligência Swarm** (**IS**) "estuda" o comportamento e desenvolvimento de sistemas inteligentes que são bio-inpirados, isto é, baseados na evolução biológica, e tem como base os coletivos do mundo animal, como por exemplo, colónias de formigas, pinguins, cardumes de peixe, entre outros.

Os sistemas de IS normalmente são compostos por uma população de agentes simples que interagem localmente entre si e com o ambiente, seguindo um conjunto de regras.

Esta é igualmente usada para resolver problemas de otimização de grande complexidade.

De todos os algoritmos evolucionários, os que são baseados na inteligência de enxame, são os que saltam mais à vista devido a muitas razões, das quais destaco: a IS usa vários agentes como uma evolução, isto é, a população está em interação, e, portanto, fornece boas maneiras de imitar os sistemas naturais, e os algoritmos que usam IS são simples e fáceis de implementar, são flexíveis e suficientemente eficientes.

Como resultado, estes algoritmos conseguem lidar com um vasto número de problemas em aplicações.

Particle Swarm Optimization (**PSO**) é um método heurístico de otimização global, que é baseado na inteligência de enxame, a qual foi falada anteriormente. Este é inspirado na pesquisa sobre o movimento do comportamento de pássaros e peixes. O algoritmo é muito utilizado e de fácil implementação. No PSO, os indivíduos são referidos como partículas, que "voam" sobre um universo de espaço hiperdimensional. Estas partículas movimentam-se com base na tendência psicossocial de avaliação do sucesso de cada indivíduo, ou seja, as mudanças para cada partícula dentro do enxame são influenciadas pelas experiências, ou conhecimentos dos seus vizinhos.

Na prática, na fase de inicialização, cada partícula recebe uma posição inicial aleatória e uma velocidade inicial. A posição da partícula representa uma solução do problema, tendo assim um valor que lhe foi dado pela função objetivo. Enquanto se movem no espaço de busca (região onde procuramos uma solução), estas memorizam a posição da melhor solução que encontraram. A cada iteração, a partícula desloca-se com uma velocidade que é baseada em três componentes: a velocidade anterior, um componente de velocidade que impulsiona a partícula em direção ao local no espaço de busca onde ela encontrou anteriormente a melhor solução até agora, e por último, um componente de velocidade que impulsiona a partícula em direção ao local no espaço de busca onde as partículas vizinhas encontraram a melhor solução até agora.

De entre os diversos algoritmos baseados em IS, destacam-se alguns que têm chamado mais a atenção ao longo dos últimos anos, tais como:

"Firefly Algorithm":

- O algoritmo de vaga-lumes é simples, flexível e fácil de implementar. Foi desenvolvido por Yang em 2008, o qual foi baseado nos padrões intermitentes e comportamentos de vaga-lumes tropicais;
- O movimento de um vaga-lume é atraído por outro mais atraente, mais brilhante.

"Cuckoo Search":

- O algoritmo cuco foi desenvolvido em 2009 por Yang e Deb. Baseia-se no parasitismo de crias de algumas espécies de cucos. Estudos recentes mostram que este algoritmo é potencialmente mais eficiente que o PSO e AG. Este algoritmo usa uma combinação balanceada de um passeio aleatório local e de um passeio global exploratório controlado por um parâmetro de comutação.

"Bat Algorithm":

- O algoritmo do morcego foi desenvolvido também por Yang, já em 2010. Foi inspirado no comportamento da ecolocalização de micromorcegos. É o primeiro algoritmo deste tipo a usar sintonia de frequência.

"Flower Algorithm":

- O algoritmo de polinização de flores foi desenvolvido por Yang em 2012, inspirado pelo processo de polinização das flores das plantas que têm flores. Foi estendido para problemas de otimização de multiobjectivo, o qual se mostrou ser muito útil e eficiente.

4. Artificial Bee Colony Algorithm

O algoritmo "Artificial Bee Colony" (ABC) é um algoritmo de otimização que simula o comportamento de uma colónia de abelhas. Este foi proposto pela primeira vez por Dervis Karaboga em 2005 para a otimização de problemas reais.

Neste modelo, a nossa colónia é composta por três tipos de abelhas: as *employed bees* (abelhas empregadas), que têm como função coletar alimentos para a colmeia numa fonte de alimento específica. As *onlooker bees* (abelhas observadoras), as quais fazem o patrulhamento das empregadas (*employed bees*) para verificar quando é que uma determinada fonte de alimento não vale mais a pena, e as *scout bees* (abelhas escoteiras), que serão as que vão procurar novas fontes de localização de alimentos. Esta fonte de alimento é definida como uma posição no espaço de busca, ou seja, é uma possível solução para o nosso problema de otimização, e inicialmente o número de fontes é igual ao número de abelhas empregadas na colmeia, sendo a qualidade da fonte definida pelo valor da função objetivo naquela posição – valor de aptidão.

Os parâmetros que controlam este algoritmo são o tamanho da colónia das abelhas e o número de ciclos.

Etapas do comportamento inteligente das abelhas:

- Primeiramente, as abelhas começam por explorar de forma aleatória o ambiente em busca de boas fontes de alimento – valor fitness, é o valor que é mostrado na consola quando se fazem as experiências com este algoritmo;
- Quando é encontrada uma fonte de alimento, a abelha torna-se uma empregada e começa assim a extrair o alimento da fonte que foi descoberta;
- Esta abelha, por sua vez retorna à colmeia com o néctar extraído e descarrega-o. Após o descarregar, a abelha pode voltar diretamente ao local onde foi descoberto a fonte de alimento ou, pode partilhar a informação do local, fazendo uma dança, no local indicado para dançar;
- Caso uma fonte de alimento se esgote, a *employed bee* passa a ser uma *scout bee* e começa a procurar aleatoriamente uma nova fonte de alimento;
- Enquanto isso, as *onlooker bees* esperam na colmeia e observam as *employed bees* enquanto estas estão a coletar o néctar da fonte, e escolhem uma fonte de entre das que são mais lucrativas (estas também fazem extração de néctar);
- A seleção de uma fonte de alimento é proporcional à qualidade da fonte – valor de aptidão.

Nota: quando é feita a implementação deste algoritmo, existem apenas dois tipos de abelhas, as *employed bees* e as *onlooker bees*. A *scout bee* é na verdade um comportamento exploratório que pode ser efetuado tanto pelas *employed bees* como pelas *onlooker bees*.

Tem se observado pelas simulações, que muitas das vezes o algoritmo ABC supera o PSO em termos de qualidade da solução, mas leva mais tempo para fazer os resultados. Por outro lado, o PSO é mais rápido, mas a *accuracy* dos resultados não é tão boa.

Como vamos poder observar mais à frente na comparação dos resultados que obtive deste algoritmo com o PSO, para funções *benchmark*, o ABC produz resultados com melhor qualidade, tanto em termos de média como de desvio padrão. A exploração do espaço de busca é melhor no ABC do que no PSO, no entanto, fazendo efetivamente as experiências, constatei que o PSO é mais rápido.

O que trás um problema, isto é, se estivermos perante aplicações mais sensíveis, onde se espera um resultado mais rápido, o ABC não pode ser usado.

O PSO é menos sensível ao tamanho da população e à dimensão do problema.

A sua *accuracy* é inferior ao ABC devido ao facto de este usar modelagem probabilística para mudar parte da solução melhor, enquanto que o ABC "corta o mal pela raiz", isto é, remove toda a população que não está a conseguir produzir e inicia aleatoriamente uma nova forma de achar melhores soluções. Assim, o ABC mostra ter uma maior capacidade no que diz respeito à pesquisa global e convergência adequada do que o PSO.

Em suma, podemos concluir que o algoritmo ABC é preferido se quisermos ter muito bons resultados (*high accuracy*). Porém, o PSO é mais adequado se se esperar obter ótimos resultados em menor tempo.

5. ABC *VS* PSO – Análise de desempenho

Para a resolução das experiências, foram usados dois *softwares* dentro do *anaconda-navigator*, o *jupiter-notebook* para fazer os testes relativos ao PSO, e o *spider* para fazer os testes do ABC.

Cada experiência foi realizada 30 vezes, onde foi feita a média e o desvio padrão para cada. As funções *benchmark* utilizadas foram as seguintes:

- Ackley;
- Sphere;
- Rastrigin;
- Rosenbrock.

Nota: é enviado na pasta do trabalho submetido um ficheiro de texto que contém todos os valores das 30 experiências com a média e o desvio padrão respetivos, bem como um ficheiro Excel que foi feito apenas para o cálculo da média e do desvio padrão.

Para o PSO, foi nos fornecido durante uma aula prática de IC, um ficheiro de *python* com este algoritmo implementado. No entanto foi preciso fazer alguns ajustes no código, pois queria que fossem realizadas as 30 experiências todas de uma vez e que apenas os seus resultados fossem mostrados e não as *stats* todas.

Aqui está a alteração feita:

```
for item in range(30):
    cost, stats = optimizer.optimize(fx.sphere, iters=100, verbose = False)
    print(cost)
```

Como o *pyswarms* não usa o ABC, tive de ir à procura de como implementar o algoritmo das abelhas em *python*. Foi então que encontrei o *Hive*, o qual é um algoritmo que se baseia em IS sobre o comportamento inteligente das abelhas. Este usa o algoritmo ABC para resolver diversos problemas e testar as diferentes funções de *benchmark*.

Porém foi igualmente preciso fazer algumas alterações. Em termos de código foi só fazer um ciclo (vou mostrar na próxima página), em termos das funções, com o *Hive* vinham duas funções já definidas e prontas a serem testadas, a *rastrigin* e a *rosenbrock*. No entanto, as que são pedidas são a *sphere* e a *ackley*, as quais tive de implementar em *python*, da seguinte maneira:

```
def ackley(vector):

    vector = np.array(vector)

    return -20.0 * np.exp(-0.2 * np.sqrt(0.5 * (vector[0]**2 + vector[1]**2))) -
        np.exp(0.5 * (np.cos(2 * np.pi * vector[0]) + np.cos(2 * np.pi * vector[1]))) + np.e + 20

def sphere(vector):

    vector = np.array(vector)

    return np.sum(np.square(vector[0]))
```

Estes foram os parâmetros que foram alterados ao longo das experiências:

ABC:

- Lower;
- Upper;
- Função;
- Número de abelhas.

```
for item in range(30):

    # creates model
    ndim = int(2)
    model = Hive.BeeHive(lower     = [-2.048] *ndim ,
                         upper     = [2.048] *ndim   ,
                         fun       = rosenbrock          ,
                         numb_bees =  100          ,
                         max_itrs  =  100           ,
                         verbose   = False          ,)
```

PSO:

- Opções;
- Dimensões;
- Função:
- Número de partículas.

```
options = {'c1': 0.3, 'c2': 0.5, 'w': 0.9}
```

```
optimizer = ps.single.GlobalBestPSO(n_particles=100, dimensions=3,options=options)
```

```
for item in range(30):
    cost, stats = optimizer.optimize(fx.sphere, iters=100, verbose = False)
    print(cost)
```

Experiências:

ABC

	Artificial Bee Colony Algorithm	
	Ackley	
Parâmetros	Média	Desvio Padrão
iterações = 100 bees = 10 intervalo = [-32.768] [32.768]	9.783021390000E-05	1.706248626260E-04
iterações = 100 bees = 50 intervalo = [-32.768] [32.768]	5.420829924200E-08	5.853254689839E-08
iterações = 100 bees = 100 intervalo = [-32.768] [32.768]	1.413990560195E-08	1.086615352150E-08

	Artificial Bee Colony Algorithm	
	Sphere	
Parâmetros	Média	Desvio Padrão
iterações = 100 bees = 10 intervalo = [-5.12] [5.12]	7.748757596301E-10	1.804277295385E-09
iterações = 100 bees = 50 intervalo = [-5.12] [5.12]	1.044817300744E-18	1.483288992318E-18
iterações = 100 bees = 100 intervalo = [-5.12] [5.12]	1.936567216779E-19	2.119072965601E-19

Para 100 iterações, número de abelhas 10, 50 e 100 e intervalo consoante o tipo de função *benchmark*, constatamos que à medida que o número de abelhas aumenta, a accuracy tanto da média como do desvio padrão aumenta também. E fazendo uma comparação entre as duas, os valores que são mais próximos de zero pertencem, à função *sphere*, esta é a que mostra melhores resultados.

	Artificial Bee Colony Algorithm			
	Rastrigin		Rosenbrock	
Parâmetros	Média	Desvio Padrão	Média	Desvio Padrão
iterações = 100 bees = 100 intervalo = [-5.12] [5.12]	1.246173534734E-12	2.605952737789E-12	-	-
iterações = 100 bees = 100 intervalo = [-2.048] [2.048]	-	-	9.084836175179E-04	8.298920318730E-04

Por curiosidade, decidi experimentar as funções *rastrigin* e *rosenbrock*. Usei 100 abelhas para a ter a melhor *accuracy* possível e usei para cada uma, o seu intervalo respetivo.

Assim, podemos observar e concluir que das quatro funções, a que nos dá valores menos perto de zero é a *rosenbrock*. De seguida vem a *ackley*, e depois a *rastrigin* (esta conseguiu chegar mais perto do que a *ackley* a zero). Por fim, a que chegou mais perto de zero foi a *sphere*.

PSO

	PSO	
	Ackley	
Parâmetros	Média	Desvio Padrão
iterações = 100 dimensões = 2 opções = {'c1': 0.5, 'c2': 0.3, 'w': 0.9} número de partículas = 10	4.513054456282E-05	2.429846261960E-04
iterações = 100 dimensões = 2 opções = {'c1': 0.5, 'c2': 0.3, 'w': 0.9} número de partículas = 50	1.687666969585E-05	9.235987823442E-05
iterações = 100 dimensões = 2 opções = {'c1': 0.5, 'c2': 0.3, 'w': 0.9} número de partículas = 100	1.902627864855E-05	1.038603720390E-04

	PSO		
	Ackley		
Parâmetros	Média	Desvio Padrão	Nota: Dava sempre o mesmo valor
iterações = 100 dimensões = 2 opções = {'c1': 0.3, 'c2': 0.5, 'w': 0.1} número de partículas = 10	3.096005192990E+00	0.000000000000E+00	
iterações = 100 dimensões = 2 opções = {'c1': 0.3, 'c2': 0.5, 'w': 0.9} número de partículas = 50	9.940218910239E-06	5.336217571073E-05	
iterações = 100 dimensões = 3 opções = {'c1': 0.3, 'c2': 0.5, 'w': 0.9} número de partículas = 100	1.542696250292E-05	8.366004866754E-05	

Usando o PSO, para 100 iterações, número de partículas 10, 50 e 100, duas dimensões e opções, c1: 0.5, c2: 0.3, w:0.9, observamos que os valores aumentam quando passamos a ter 50 partículas, mas quando temos 100, o valor fica mais longe de zero. Podemos constatar também que os valores são menos próximos de zero do que no ABC.

Mudando as opções para c1: 0.5, c2: 0.3, w:0.1, porque achei que seria interessante ver a inércia quase a 0, o valor era sempre o mesmo, devido ao facto de as partículas não se mexerem, por isso voltei a colocar a inércia a 0.9 e troquei apenas os valores de c1 e c2. Com 50 partículas, o valor aproximou-se de zero, com o c1 e c2 trocados, mas com 100, volta a afastar-se de zero, mesmo com a dimensão a 3.

	PSO	
	Sphere	
Parâmetros	Média	Desvio Padrão
iterações = 100 dimensões = 2 opções = {'c1': 0.5, 'c2': 0.3, 'w': 0.9} número de partículas = 10	2.424246432029E-08	1.327682541991E-07
iterações = 100 dimensões = 2 opções = {'c1': 0.5, 'c2': 0.3, 'w': 0.9} número de partículas = 50	7.476277626111E-11	4.093134287448E-10
iterações = 100 dimensões = 2 opções = {'c1': 0.5, 'c2': 0.3, 'w': 0.9} número de partículas = 100	7.579944967933E-11	4.151706703905E-10

	PSO		
	Sphere		
Parâmetros	Média	Desvio Padrão	Nota: Dava sempre o mesmo valor
iterações = 100 dimensões = 2 opções = {'c1': 0.3, 'c2': 0.5, 'w': 0.1} número de partículas = 10	2.957803900539E-02	0.000000000000E+00	
iterações = 100 dimensões = 2 opções = {'c1': 0.3, 'c2': 0.5, 'w': 0.9} número de partículas = 50	1.759340999252E-10	9.633388315613E-10	
iterações = 100 dimensões = 3 opções = {'c1': 0.3, 'c2': 0.5, 'w': 0.9} número de partículas = 100	4.700914487565E-09	2.574787176332E-08	

Aqui, mantive tudo igual, mas usei a função *sphere*.

Com 50 partículas, os valores aproximam-se mais de zero e como vemos à esquerda, com 100 partículas eles mantém-se mais ou menos iguais, não decrescem, como é o caso da ackley.

Mudando as opções como em cima o valor era sempre o mesmo com a inércia a 0.1, por isso voltei a colocar a inércia a 0.9 e troquei apenas os valores de c1 e c2. Com 50 partículas, o valor aumentou, com o c1 e c2 trocados, mas com 100, volta a afastar-se de zero, mesmo com a dimensão a 3.

No entanto podemos conferir que os valores são mais próximos de zero usando a função *sphere* do que a *ackley*, tanto para o **PSO** como para o **ABC**.

Curiosidade

Por último foi feito uma última experiência, graças ao *Hive*, que tem por base este algoritmo ABC para fazer uma réplica de uma pintura com polígonos.

Esta pintura foi pintada por Henri Matisse, em 1952.

Pintura "azul nu"

100 iter + 20 Abelhas

50 iter + 100 Abelhas

Inicialmente usei 20 abelhas e depois 100. No entanto, para usar as 100 abelhas, tive de baixar o número de iterações para 50, porque o meu computador não aguentava. Mesmo assim, nota-se que o algoritmo vai conseguir replicar bem a imagem, pois os polígonos estão a ficar mais bem definidos e a ficar na posição correta à medida que o número de abelhas aumenta. O único problema é que demora muito tempo.

6. Conclusão

Com a realização deste trabalho, consegui adquirir bastantes conhecimentos no que diz respeito à computação evolucionária, a algoritmos genéticos, a inteligência swarm, aos algoritmos PSO e ABC, à linguagem de programação *python* e aos softwares Excel, *spider* e *jupiter-notebook*.

Deu-nos a conhecer o potencial dos algoritmos de otimização PSO e ABC para a análise de problemas.

Com este trabalho foi-nos possível observar que a função *benchmark* que apresentou um melhor desempenho quando utilizada foi a função *sphere*.

Conclui-se assim, que este trabalho irá ajudar para a fase III, pois envolveu problemas de otimização, funções *benchmark* e testes com diferentes híper-parâmetros.