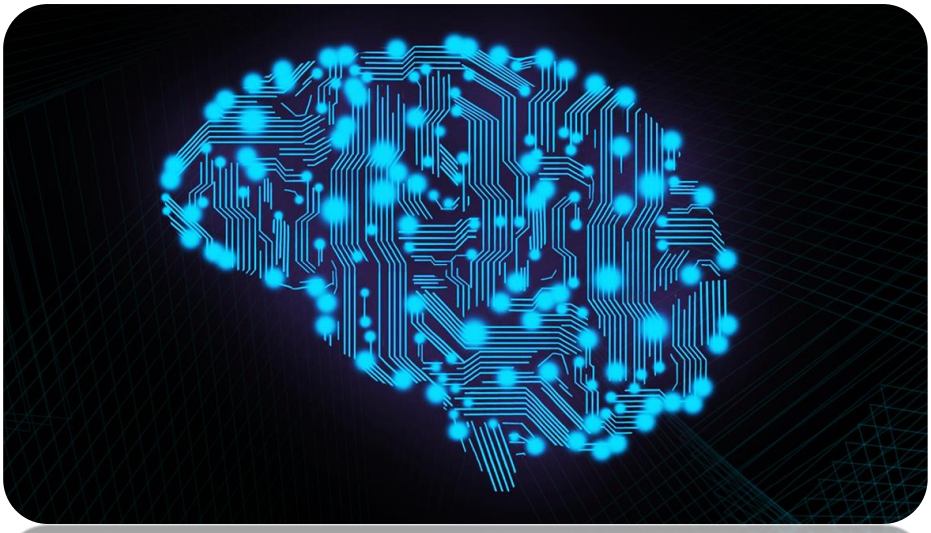
UNIVERSIDADE LUSÓFONA DO PORTO



Inteligência Artificial

META HEURISTICAS

Pedro Assunção | 21003163 | 20 de janeiro de 2020

Âmbito

Este relatório visa documentar todo o projeto realizado no âmbito da disciplina de inteligência artificial lecionado pelo docente David Rua do curso de engenharia informática da Universidade Lusófona do Porto.

Os objetivos gerais deste projeto passam por:

- Identificar a natureza dos problemas a resolver
- Modelar problemas
- Escolher que algoritmos/heurísticas para a resolução
- Implementar em python
- Testar performance

Problema 1

O 1º problema passa por resumidamente implementar 2 algoritmos diferentes para a minimização de funções matemáticas que são dadas.

1. Função de ackley
2. Função de Rastrigin
3. Função de Rosenbrock

A estas função podem ser aplicados os algoritmos

1. Algoritmo genético (GA)
2. Enxame de partículas (PSO)
3. Entropia cruzada (CE)

Nos exercícios seguintes iremos abordar estratégias diferentes para o mesmo objetivo que é minimizar a função aplicando o parâmetro D = 2 para alterar a dimensionalidade do problema.

Os exercícios apenas demonstram uma dimensionalidade por dificuldades de na implementação com mais dimensões.

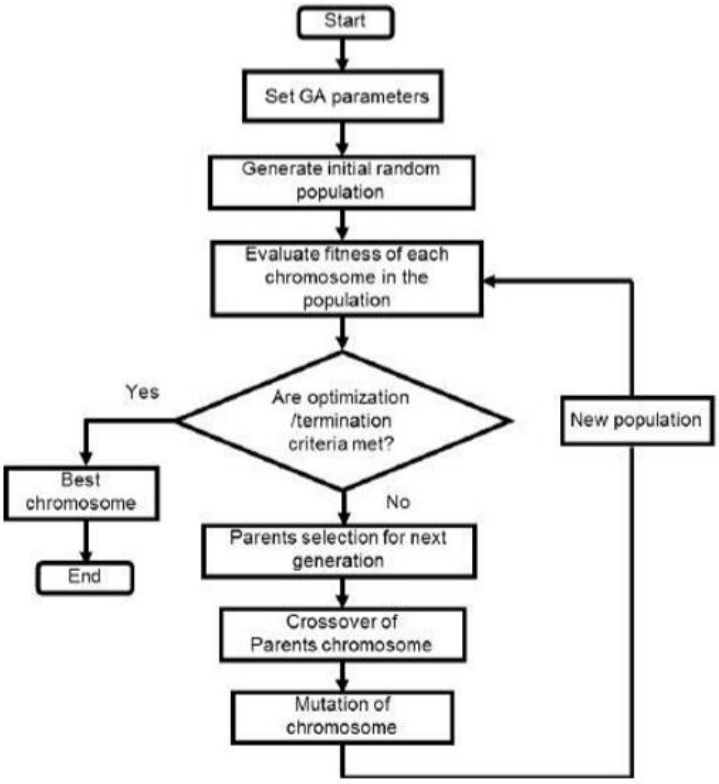


Figura 1 Flowchart AG

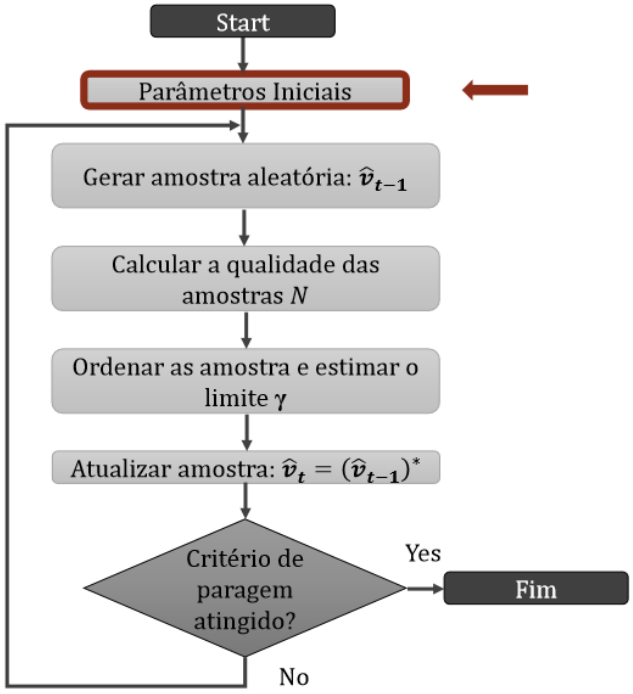


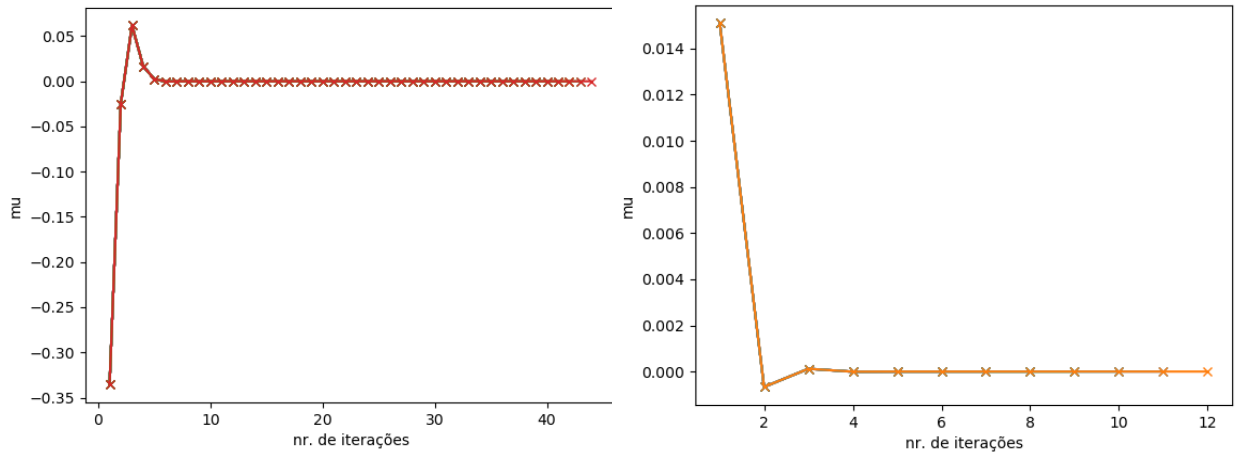
Figura 2 Flowchart EC

Nas figuras anteriores podemos ver o flowchart de cada algoritmo onde podemos concluir que existem diferentes abordagens para resolver o mesmo problema tendo cada um as suas vantagens como tempo de execução e resultado obtido mais perto do ótimo etc.

ACKLEY (CROSS ENTROPY)

Sabendo o funcionamento base do algoritmo em questão foi implementado o código que está em repositório (aqui).

Após alteração de alguns parâmetros chegamos a conclusão que o nosso algoritmo converge bastante rápido bastando apenas cerca de 12 iterações para chegar a um resultado ótimo.



Como podemos ver no gráfico gerado o nrº de iterações podem variar dependendo do número dos parâmetros definidos no início do programa, fazendo a função convergir mais rápido. O número de iterações foi diferente pois temos 2 critérios de paragens definidos no nosso ciclo while onde primeiramente o critério de paragem é o número de iterações (t < maxits) e outro critério é quando a função de gauss já está tao "fechada" que a evolução de nosso um já está estagnada (sigma2 > épsilon). Epsilon é um número positivo muito próximo de zero de forma a apertar a nossa gaussiana o máximo possível. Neste algoritmo a gaussiana será desenhada (X = np.random.normal(mu, sigma2, N)) centrada em mu, com a escala de sigma2, de tamanho N.

Ao longo do ciclo podemos ver ser atribuídos a um o valor medio dos 10 melhores pontos e a cada iteração a gaussiana a ser convergida de modo a minizar o nosso mu.

RASTRIGIN (GENETIC ALGORITHM)

.

Neste algoritmo foi aplicado um exemplo de algoritmo genético. Este exercício também acaba por apenas representar a função em apenas 1 dimenção.

Foi utilizado e adaptado um exemplo de código.

Foi alterado a funções para estas realizarem o pretendido como por exemplo. Gerar nossas soluções dentro do domínio pretendido, calcular a fitness para minimizar a função.

Apos realizar alguns testes alterando os parâmetros chegamos a conclusão que o parâmetro que mais teve impacto no nosso algoritmo seria alterar o valor de mutation_rate para valores mais baixos pois assim o algoritmo iria convergir mais rápido. Em todas as execuções o algoritmo chegou sempre a uma solução ótima ou quase opima.

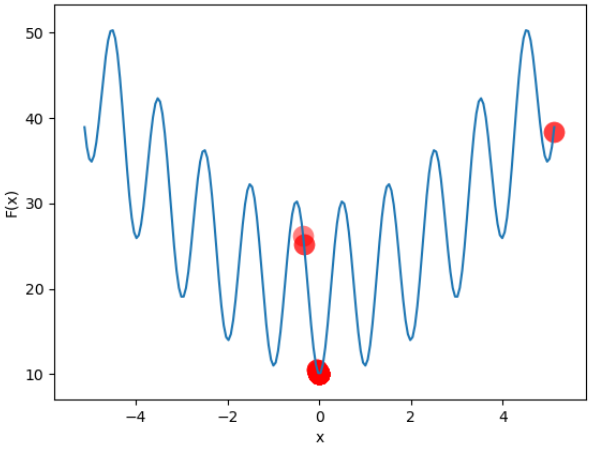


Figura 3 resultado Rastrigin(GA)

Problema 2

O 2º problema consiste num conjunto de programas com diferentes tempos de execução que serão corridos em vários servidores.

O objetivo do problema é encontrar a combinação de servidores que permita minimizar o tempo de execução dos programas utilizando uma meta heurística.

CONTEXTO

Foi optado por aplicar um algoritmo genético devido a natureza do problema pois os AG's são apropriados para problemas de otimização complexos que evolvem várias variáveis. Os AG's são no geral uma otima escolha pois conseguem abranger um grande número de aplicações

Código e funções

Parâmetros

```
SERV = 10                # numero de servidores
TASKS = 10               # numero de tarefas
POPULATION = 100        # Size of Population
CROSS_RATE = 0.8        # CrossOver Chance
MUTATION_RATE = 0.03    # Mutation Rate
N_GENERATIONS = 100
```

Aqui são declarados todos os parâmetros que serão usados no algoritmo utilizando a metáfora evolutiva

É importante analisar de que maneira estes parâmetros irão influenciar o comportamento do algoritmo para podermos adaptá-los ao nosso problema.

Population - Afeta o desempenho global e a eficiência do algoritmo. Este valor sendo baixo faz o desempenho cair pois fornece pouca cobertura do espaço de busca do problema. Se o valor for grande o algoritmo demorará um maior tempo a processar todas as soluções.

Cross rate – Com valores baixos o algoritmo torna-se bastante lento a "andar para a frente". O valor sendo muito alto as soluções com melhor fitness serão substituídas rapidamente por outras podendo "saltar" outras soluções que poderiam ser melhores para o problema.

Mutation rate - Este parâmetro geralmente é baixo para explorar melhor o nosso espaço de soluções sendo que valores muito altos tornaria a nossa procura mais "aleatória".

Nº Gerações – Este será o principal critério de paragem do nosso algoritmo. Portanto terão de ser feitos testes para ver qual será o melhor valor para chegar a uma solução quase ótima em menor espaço de tempo.

FITNESS

```
def fitness(elemento, matcusto):
    elemento = np.squeeze(np.asarray(elemento))
    result = elemento * matcusto
    custo_total = np.sum(result)
    return custo_total
```

Esta função é responsável por transformar a nossa "matriz de matrizes" em um valor de custos que será feito para determinar as melhores soluções obtidas. Para isso é somado todos os valores da matriz e "convertido" num único valor (custo total).

CROSSOVER

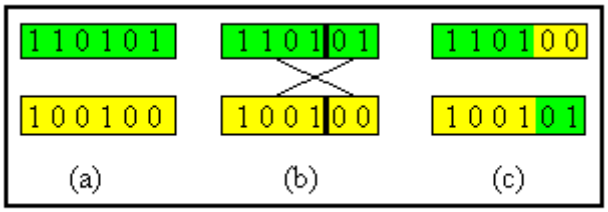
```
def crossover(solution1, solution2):
    corta1 = solution1[:, 0:3]
    print(" seleciona 3 colunas primeira solução", "\n\n", corta1, "\n\n")

    corta2 = solution2[:, 3:10]


    print(" seleciona da 4 a 10 coluna da 2º solucao", "\n\n", corta2, "\n\
n")

    # Junta os cortes das outras matrizes para gerar nova solução
    filho = np.hstack([corta2, corta1])

    print(" Matriz Filho com 3 colunas da solução 1 e 7 colunas da solução
2", "\n\n",
          filho, "\n\n")

    return filho
```

Esta é uma função arbitrária que define o critério do crossover entre pais para gerar um novo filho. Neste caso é selecionado as colunas das matrizes "pais" e combinando as 2 meias matrizes é gerado um novo filho com características diferentes.

A estratégia é arbitrária e para testarmos a sua eficiência teremos de testar na prática.

A estratégia de crossover

Neste exemplo o crossover é feito desta forma

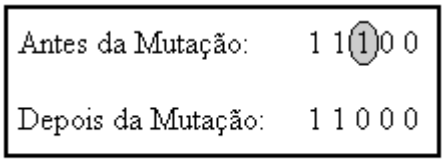


Um exemplo de crossover de um ponto.

MUTATION

```
def mutacao(novo_filho):
    crianca = np.matrix(novo_filho)
    aux = crianca[:, [1, 3]]
    crianca[:, [1, 3]] = crianca[:, [3, 1]]
    crianca[:, [3, 1]] = aux

    return crianca
```

Esta função é utilizada para procurar novas características nas soluções que podem surgir aleatoriamente, quebrando a monotonia da descendência



Exemplo de mutação.

ACEITA REJEITA

```
def aceitaRejeita(population, tempototal, matcusto):
    end = 0

    while True:

        solucao = []

        solution = population[
            random.randint(0, len(population) - 1)]
        solucao.append(fitness(solution, matcusto))

        maxFit = choice(tempototal)

        if (maxFit < solucao):
            return solution
        end = end + 1

        if end > 1000:
            return None
```

Esta função é responsável por selecionar quais soluções/matrizes com melhor fitness.

Nesta função é selecionado uma solução de forma aleatória comparando-a com a variável maxFit. Caso a solução selecionada tiver melhor fitness essa passará a ser a melhor solução encontrada.

Foi colocado a variável end para evitar que o ciclo entre em loop infinito por não encontrar nenhuma solução melhor.

Debug do código

Com os seguintes parâmetros:

POPULATION = 2 (Número mínimo de pais para gerarem novas soluções)

N_GENERATIONS = 1 (nº de iterações só uma para podermos ver o que esta a ser feito)

Conseguimos ver mais facilmente o que acontece no código, onde podemos ver as matrizes a serem geradas e o processo de seleção para serem geradas novas matrizes a cada iteração.

```
Solução 1

[[0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0]
[1 0 1 0 0 0 0 0 0 0]
[0 0 0 0 1 0 1 1 0 1]
[0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]]

Solução 2

[[1 0 0 0 0 0 0 1 0 0]
[0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 1 0 1 0 0 0]]


seleciona 3 colunas primeira solução

[[0 0 0]
[0 0 0]
[0 0 0]
[0 1 0]
[0 0 0]
[0 0 0]
[1 0 1]
[0 0 0]
[0 0 0]
[0 0 0]]


seleciona da 4 a 10 coluna da 2º solucao

[[0 0 0 0 1 0 0]
[0 0 0 0 0 0 0]
[0 0 0 0 0 0 1]
[0 0 0 0 0 0 0]
[1 0 0 0 0 0 0]
[0 0 0 0 0 1 0]
[0 0 1 0 0 0 0]
[0 0 0 0 0 0 0]
[0 0 0 0 0 0 0]
[0 1 0 1 0 0 0]]
```

Colocando os parâmetros:

```
# Parameters
SERV = 10                 # numero de servidores
TASKS = 10                # numero de tarefas
POPULATION = 100         # Size of Population
CROSS_RATE = 0.003         # CrossOver Chance
MUTATION_RATE = 0.08     # Mutation Rate
N_GENERATIONS = 100      # Number of Generations
```

Obtemos a seguinte solução

```
Número de Gerações: 100

Número de Crossovers: 10000

Número de Mutações: 773

MELHOR SOLUCAO:

[[0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 1 0 1 0 1]
[0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]
[1 0 0 1 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 1 0 0 0]]
```

Alterando o número de iterações para 200 o tempo de execução aumentou chegando a esta solução

CONCLUSÃO

Podemos concluir que este tipo de algoritmos (GA, CE, PSO, DE que foram estudados em aula) tem se demonstrado bastante eficazes na resolução de problemas de optimização combinatório que seriam bastante trabalhosos.

Concluiu-se que para a resolução do Problema 2 seria melhor optar por um algoritmo genético devido a natureza do problema onde este poderia ser dividido em várias variáveis tornando-se assim o melhor algoritmo a ser aplicado ao problema.

O objetivo inicial seria importar a nossa matriz de custos de um ficheiro externo que apresentava valores negativos (-1 para mostrar que a tarefa não poderia ser resolvida por aquele servidor). Não conseguindo adaptar o código para esta importação devido erros de índex ou arrays "out of shape" foi optado por gerar uma matriz aleatória entre 0 e 1.

O algoritmo retorna sempre uma solução possível atribuindo cada tarefa apenas um servidor.

Este projeto ajudou-me numa melhor interpretação, conhecimento de estratégias para a resolução de problemas aplicando meta-heuristicas.

Tambem desenvolvi as minhas capacidades de desenvolver linguagem de programação em python e um melhor conhecimento de ferramentas de controlo de versões git.

Anexos

#Funcoes python

Import numpy as np

Import random

Python	**Função**
np.array	Conversão explicita do array
np.squeeze	Remove as entradas "single-dimensional" do formato do array.
np.argsort	Organiza o array
np.asarray	Converte a entrada num array
zip	Agrupar listas num "tuple"
np.sum	Somatório dos elementos
random.randint	Gera número aleatório inteiro entre
append	Adiciona elemento à lista
choice	Escolhe aleatoriamente um elemento da lista
Np.hstack	"Stacka" o array em sequência
Np.matrix	Estruturar matriz