ANTÔNIO CARLOS DURÃES DA SILVA
JOEL WILL BELMIRO

GitHub: https://github.com/duraes-antonio/IA_trab1_A_Star

INTELIGÊNCIA ARTIFICIAL – TRABALHO 2 – PSO

1 EXPLICAÇÃO TEÓRICA DO ALGORITMO

Semelhante a técnicas como Algorítimo Genético, Otimização por Colônia de Formigas, Otimização por Enxame de Vaga-lumes, o desenvolvimento deste algorítimo foi inspirado na observação do comportamento de um indivíduo em seu meio natural e atuação no coletivo [1].

O PSO simula o comportamento de um ou mais grupos (enxames) de uma espécie qualquer em busca de encontrar o máximo possível de alimento [2].

Sempre que uma partícula encontra uma quantidade de alimento, verifica-se se este número é superior ao máximo já encontrado, se assim for, a quantidade máxima é atualizada, e a localização em que essa partícula encontrou o alimento é armazenada.

As demais partículas ao se movimentarem em busca de alimento levarão em conta a localização que trouxe maior quantidade de alimento até então, dessa forma, tendem a seguir a partícula mais “bem-sucedida”.

Por trabalhar com a atuação de grupos sociais, o algorítimo leva em consideração dois principais conceitos sobre o comportamento de cada partículas, são eles: Inteligência cognitiva e social [3]. A inteligência social dita o quanto que o comportamento do bando influencia as ações do indivíduo, enquanto que a inteligência cognitiva diz respeito ao quanto o indivíduo considera como relevante seus próprios achados.

Além das inteligências citadas acima, o algorítimo trabalha com os conceitos de posição do indivíduo (em N dimensões, neste trabalho 2 dimensões serão abordadas) no espaço, e por este se movimentar, inclui-se também a variável velocidade (um valor para cada dimensão). Pelo fato de haver uma velocidade presente, há autores que optam pela adição de uma variável que represente um fator de inércia (representada em muitas obras como “w”), isto é, um valor que limite ou controle o quanto cada partícula está avançado no espaço.

2 PROBLEMA PROPOSTO:

O problema que este trabalho propõe para compreensão de aplicação do algorítimo de Otimização por Enxame de Partículas é a minimização do resultado de uma fórmula matemática, a sexta fórmula de Schaffer[4].

A ideia é atingir o resultado mais próximo de zero com a execução da função por cada uma das partículas.

3 IMPLEMENTAÇÃO:

3.1 MODELAGEM DOS ATORES

Para implementação deste algorítimo, optamos por modelar estrutura contendo dois valores reais, par de eixos (x e y), nomeada de “Eixos”.

```
/*Estrutura contendo o par de eixos X e Y;
 *Representa a posição, um ponto em um plano 2D;*/
struct Eixos {
      double x;
      double y;
};
```

Também modelamos uma estrutura chamada “Best” responsável por armazenar a posição (Eixos) e o melhor (neste caso, menor) valor (número real) encontrado até então.

```
/*Armazena a posição que trouxe o melhor resultado da aplicação de uma função;
 *Fitness é o melhor valor, propriamente dito;*/
struct Best {
      Eixos pos;
      double fitness;
};
```

Por fim, foi realizada a modelagem da partícula em si, contendo sua posição atual (“pos”), velocidade (“vel”) para cada eixo, e seu momento Best, contendo a posição e melhor valor encontrado.

```
struct Particula {

    /*Posição atual da partícula no plano*/
    Eixos pos;

    /*Par contendo a velocidade no eixo X e a velocidade no eixo Y*/
    Eixos vel;

    /*Estrutura com a posição do melhor valor alcançado pela partícula até então*/
    Best pbest;
};
```

3.2 FUNÇÃO FITNESS

6ª Função de Schaffer, que juntamente às suas outras funções, são utilizadas para teste de otimização [5]:

```
double f(double x, double y) {
    double numerador = pow(sin(sqrt(x*x + y*y)), 2) - 0.5;
    double denominador = pow(1 + 0.001 * (x*x + y*y), 2);
    return 0.5 + (numerador / denominador);
}
```

3.3 CONSTANTES

As seguintes constantes foram definidas de acordo com a especificação. Valores máximo e mínimo para a posição da partícula no plano:

```
#define POS_MIN -100   /*Valor mínimo do intervalo de posição*/
#define POS_MAX 100    /*Valor máximo do intervalo de posição*/
```

Fator multiplicativo de velocidade e número desejado de execuções (utilizado para capturar as métricas de eficácia e, posteriormente, plotagem do gráfico comparativo):

```
#define V_FATOR 0.15   /*15% do intervalo da posição máxima ou mínima*/
#define N_EXEC  10     /*Número de execuções do algorítimo*/
```

Coeficiente cognitivo, coeficiente social e fator de inércia (valor que o grupo encontrou resultados mais satisfatórios):

```
double c1 = 2.05, c2 = 2.05, w = 0.5;
```

3.4 ALGORÍTIMO PSO

3.4.1 PseudoAleatoridade

Defina a semente de randomicidade e o mecanismo (que neste caso é o Mersenne Twister 19937) responsável por definir o número aleatório.

```
/*Variáveis para gerar números pseudo-aleatórios*/
random_device rand_semente;
mt19937 rand_engine(rand_semente());
```

Para cada grupo de número que será gerado de forma pseudoaleatória defina seu intervalo de aleatoriedade (menor valor e maior valor possível):

```
/*Valor randômico para posição da partícula, inclui parte negativa*/
uniform_int_distribution<int> rand_int(POS_MIN, POS_MAX);

/*Valor randômico para velocidade, inclui parte negativa. Já calcula a porcentagem de velocidade*/
uniform_real_distribution<double> rand_double_vel(POS_MIN * V_FATOR, POS_MAX * V_FATOR);

/*Valores reais randômicos entre 0 e 1 para as variáveis r1 e r2, inclusas no cálculo da velocidade*/
uniform_real_distribution<double> rand_double_pos(0, 1);
```

3.4.2 Geração e Inicialização das N Partículas

Função que gera um conjunto com N partícula já inicializadas com os valores dentros dos limites estabelecidos:

```
vector<Particula*> gerar_particulas(int num_particulas) {
    Particula *temp_part;
    vector<Particula *> particulas;

    /*[Instrução da ESPECIFICAÇÃO]
      *Gere a velocidade randômica de X e a randômica de Y, atribua a p/ todas partículas*/
    double geral_vx = rand_double_vel(rand_engine);
    double geral_vy = rand_double_vel(rand_engine);

    for (int i = 0; i < num_particulas; ++i) {
        temp_part = (Particula *) malloc(sizeof(Particula));

        /*Defina um fitness padrão (flag), atribua a posição X e Y randômicas*/
        temp_part->pbest.fitness = -1;
        temp_part->pos.x = rand_int(rand_engine);
        temp_part->pos.y = rand_int(rand_engine);

        /*Atribua a velocidade gerada à partícula*/
        temp_part->vel.x = geral_vx;
        temp_part->vel.y = geral_vy;
        particulas.push_back(temp_part);
    }
    return particulas;
}
```

3.4.3 Cálculo de velocidade

```
/*Calcula e retorna a nova velocidade da partícula*/
double calc_v(double pbest_pos, double current_v, double p_pos, double gbest_pos){

    /*Gere os valores randômicos entre 0 e 1 para r1 e r2*/
    double r1 = rand_double_pos(rand_engine);
    double r2 = rand_double_pos(rand_engine);

    /*Calcule a velocidade cognitiva e social*/
    double veloc_cognit = c1 * r1 * (pbest_pos - p_pos);
    double veloc_social = c2 * r2 * (gbest_pos - p_pos);

    /*Calcule a velocidade final com o fator de inércia*/
    double v = w * current_v + veloc_cognit + veloc_social;

    /*Se ultrapassar o limite máximo/mínimo, defina vel. como máx/min*/
    if (v > POS_MAX * V_FATOR) v = POS_MAX * V_FATOR;
    else if (v < POS_MIN * V_FATOR) v = POS_MIN * V_FATOR;

    return v;
}
```

3.4.4 Atualização da posição da partícula

Função responsável por atribuir uma posição válida (que não ultrapasse o mínimo ou máximo):

```
/*Atualiza a posição de uma partícula, considerando seus máximos e mínimos*/
void atualizar_posicao(Particula* particula) {

    particula->pos.x = particula->pos.x + particula->vel.x;
    particula->pos.y = particula->pos.y + particula->vel.y;

    /*Se a posição X ultrapassar a posição máxima*/
    if (particula->pos.x > POS_MAX) {
        particula->pos.x = POS_MAX;
        particula->vel.x = 0;
    }

    else if (particula->pos.x < POS_MIN) {
        particula->pos.x = POS_MIN;
        particula->vel.x = 0;
    }

    /*Se a posição Y ultrapassar a posição máxima*/
    if (particula->pos.y > POS_MAX) {
        particula->pos.y = POS_MAX;
        particula->vel.y = 0;
    }

    else if (particula->pos.y < POS_MIN) {
        particula->pos.y = POS_MIN;
        particula->vel.y = 0;
    }
}
```

3.4.5 Fluxo principal

```
int main(int argc, char *argv[]){

    /*PASSO 1: Determinar o número de partículas, iterações*/
    n_particulas = atoi(argv[1]);
    int iteracoes[] = {20, 50, 100};
    FILE * arquivo;

    /*Loop para quantidade de execuções*/
    for (int i = 0; i < N_EXEC; ++i){

        /*Para cada conjunto de iterações (20, 50, 100)*/
        for (int n_iteracoes : iteracoes){

            /*Nomeie os arquivos de saída no seguinte formato:
              *{NUM-PARTICULA}p_{NUM-ITERAÇÕES}i_{NUM-EXECUÇÃO-ATUAL}exec.csv*/
            char buffer[128];
            sprintf(buffer, "%dp_%di_%dexec.csv", n_particulas, n_iteracoes, i+1);

            arquivo = fopen(buffer, "w");

            /*PASSO 2 e 3: Inicializar cada partícula com a mesma veloc. e posição diferente*/
            vector<Particula *> ps = gerar_particulas(n_particulas);
            Best gbest;
            gbest.fitness = DBL_MAX;

            for (int k = 0; k < n_iteracoes; ++k) {

                /*PASSO 4: Para cada partíc., calcule sua aptidão e verifique seu p-best*/
                for (Particula *part: ps) {
                    double fitness = f(part->pos.x, part->pos.y);

                    if (fitness < part->pbest.fitness || part->pbest.fitness == -1.0) {
                        part->pbest.pos = part->pos;
                        part->pbest.fitness = fitness;

                        /*PASSO 5: Descobrir qual a melhor partícula globalmente*/
                        if (fitness < gbest.fitness) {
                            gbest.pos = part->pos;
                            gbest.fitness = fitness;
                        }
                    }

                    /*PASSO 6a e 6b: Atualizar a velocidade e a posição de cada eixo*/
                    part->vel.x = calc_v(part->pbest.pos.x, part->vel.x, part->pos.x, gbest.pos.x);
                    part->vel.y = calc_v(part->pbest.pos.y, part->vel.y, part->pos.y, gbest.pos.y);
                    atualizar_posicao(part);
                }

                /*Escreva o número da iteração e o resultado do gbest atual*/
                fprintf(arquivo, "%d;%.30lf\n", k+1, gbest.fitness);
            }

            fclose(arquivo);

            /*Libera a memória alocada dinamicamente*/
            for (int j = 0; j < n_particulas; ++j) free(ps[j]);
        }
    }
}
```

4 EXEMPLO DE CHAMADA

As instruções abaixo foram testadas em distribuições Linux (Ubuntu 18.04 LTS). Para executar o código é necessário ter instalado:

- Python 3
- g++ (Ou outro compilador para C++)

IMPORTANTE: Os comandos devem ser executados na pasta "src"!

Se deseja plotar o gráfico com a média de cada iteração em todas execuções, execute o seguinte comando:

- python3 grafico.py [Para distribuições **Linux**]
- python grafico.py [Para sistemas **Windows**]

Com o comando acima, o script em Python se encarregará de encapsular o processo de compilação e chamada do algorítimo escrito em C++, além ler os arquivos de saída gerados, realizar cálculo da média de cada iteração em todas execuções e plotagem do gráfico. Além disso, o script instala a biblioteca necessária para plotagem do gráfico, a matplotlib.

Se deseja apenas gerar os arquivos de saída, sem a plotagem, siga os passos:

1. Compile o código com o seguinte comando (se estiver em um terminal com suporte ao compilador g++):
   - g++ -Wall -o nome_executavel PSO.cpp

2. Execute o código com o seguinte comando:
   - ./nome_executavel numero_de_particulas [Em distribuições **Linux**]
   - nome_executavel.exe numero_de_particulas [Em sistemas **Windows**]

4.1 ENTRADAS

O algoritimo recebe como entrada um parâmetro:

- Número de partículas: Quantidade de partículas que buscarão a minimização do resultado da função fitness.

4.2 SAÍDA

Ao executar o PSO, em C++, serão gerados os arquivos contendo o melhor valor global (gbest) encontrado pelas partículas após cada iteração. O nome do arquivo de saída obedece o seguinte formato: {núm. de partículas}p_{núm. de iterações}i_{núm. da execução atual}exec.csv.

Exemplos de nomes de arquivo p/ 20 partículas, 50 iterações e 10 execuções:

- "20p_50i_1exec.csv"
- "20p_50i_2exec.csv"
- "20p_50i_3exec.csv"

Cada linha contém o número da iteração atual (começando de 1) e o valor do fitness do gbest nesta iteração.

Ao executar o script em Python, também será gerada uma planilha (.csv) para cada conjunto de iterações. Nesta planilha é possível acompanhar o valor de cada gbest de cada iteração em todas execuções, todos gbests da iteração 1, nas 10 execuções (exemplo abaixo).

n_partic	n_iteracao	i_execução	i_iteracao	i_fitness
20	50	1	1	0,051820600118896
20	50	2	1	0,483880886852294
20	50	3	1	0,475886015619045
20	50	4	1	0,276426060322395
20	50	5	1	0,455111119159534
20	50	6	1	0,464452347483977
20	50	7	1	0,485056367630231
20	50	8	1	0,487540363274722
20	50	9	1	0,44412393691165
20	50	10	1	0,482418079904142
20	50	1	2	0,043701118214037
20	50	2	2	0,38964262630222
20	50	3	2	0,475886015619045
20	50	4	2	0,247856328647459
20	50	5	2	0,452926677023316

O nome da planilha segue o formato: TABELA_{núm. de partículas}p_{núm. de iterações}i.csv (Exemplo: “TABELA_20p_50i.csv” para 20 partículas e 50 iterações).

5 RESULTADOS E OBSERVAÇÕES

Por se tratar de um algorítimo com valores randômicos e reais envolvidos, a cada execução é possível um resultado inédito. Contudo, por fim de registro e exemplificação, abaixo estão exemplos de gráficos exibindo o resultado de execuções com 20 partículas, 10 execuções e 20, 50 e 100 iterações respectivamente.

Gráfico 1: Resultado de 10 execuções, 20 partículas e 20 iterações

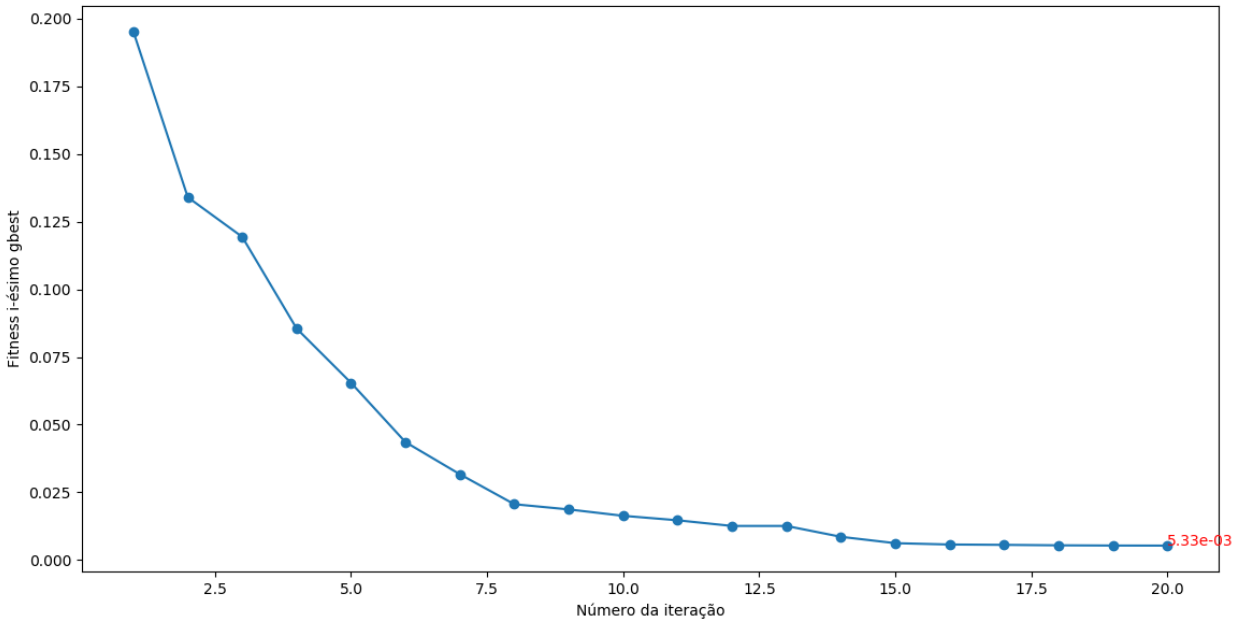


Gráfico 2: Resultado de 10 execuções, 20 partículas e 50 iterações

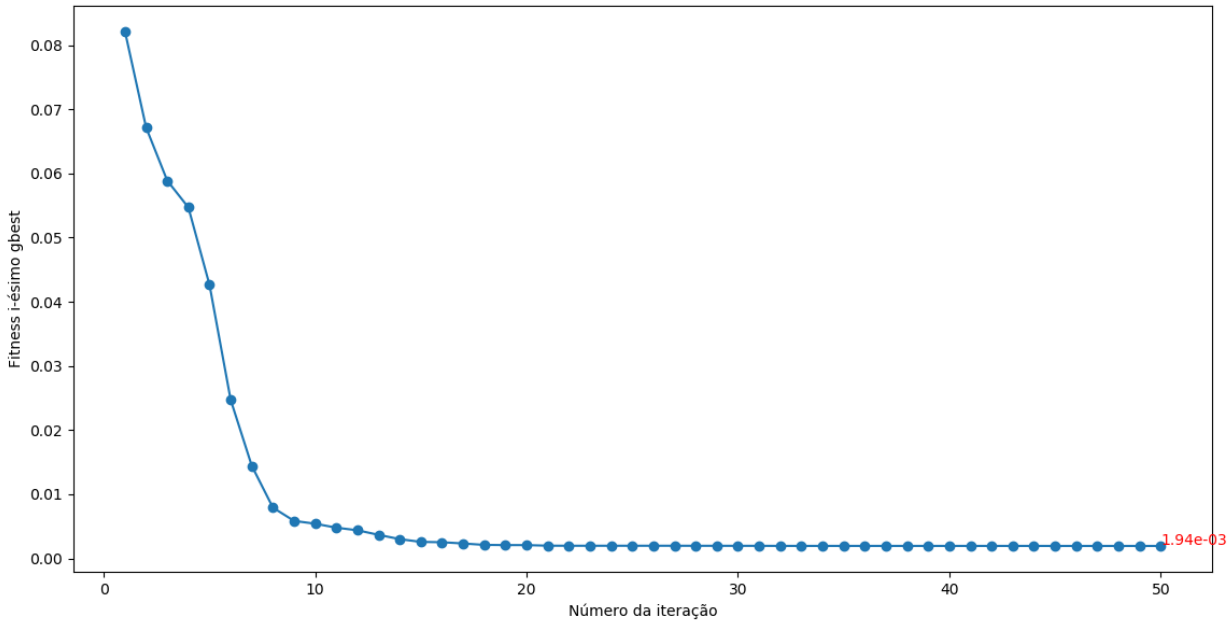
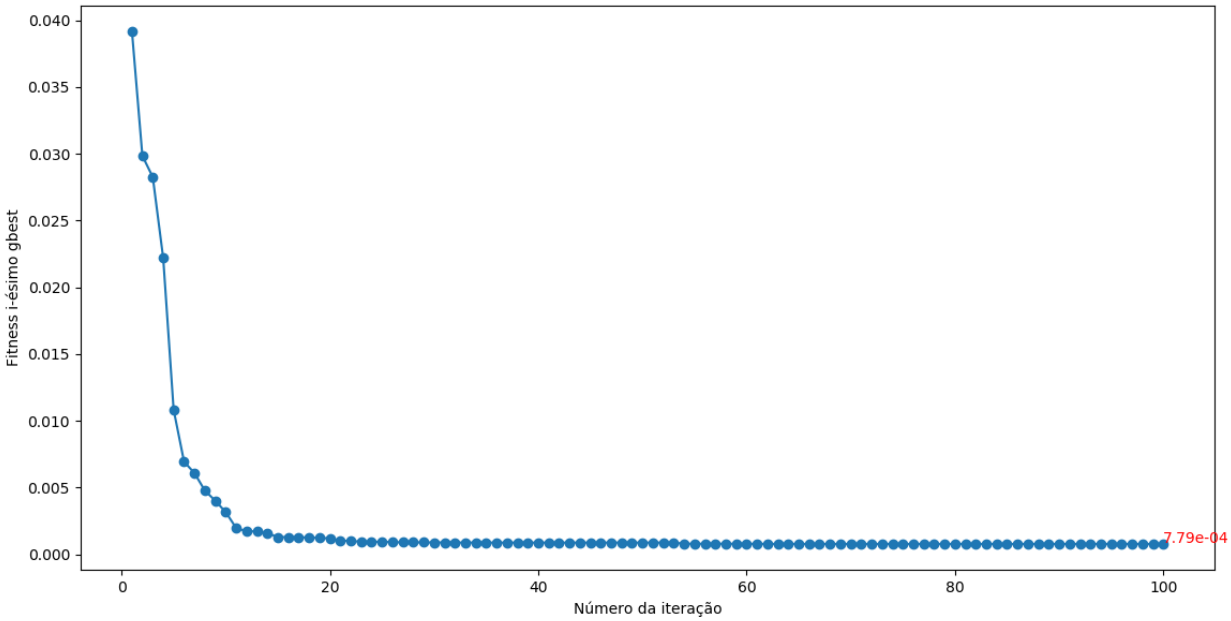


Gráfico 3: Resultado de 10 execuções, 20 partículas e 100 iterações



5.1 OBSERVAÇÕES GERAIS

Durante e principalmente após a realização das pesquisas para materialização deste trabalho, foi possível notar que:

- O intervalo da posição das partículas recomendado pela especificação entregue pelo professor, -100 e 100, são valores de domínio da 6ª função de Schaffer;
- Imagina-se que uma execução com dezenas de milhares de partículas, cada partícula em uma thread possa prejudicar o desempenho do algorítimo. Contudo, ordenar que cada **grupo de partículas** seja executado em um núcleo é algo que além de factível, pode maximizar o desempenho da execução;
- Parte considerável das implementações e artigos lidos tratam as posições e, consequentemente, velocidades, como um conjunto, uma lista, por exemplo. Esta abordagem seria adotada, todavia, apenas acrescentaria complexidade na manipulação das estruturas de dados, tendo em vista que a fórmula de Schaffer dada apenas aceita dois parâmetros (ou seja, no máximo duas dimensões);
- A ponderação do fator de inércia, não trouxe melhorias no cálculo da função fitness, ao final da execução os resultados eram inferiores aos obtidos antes da modificação;
- A heurística de inicializar as partículas em posições randômicas demonstra-se bastante eficaz quando combinada com um número crescente de partículas, uma vez que maximiza-se a chance de, pelo menos, uma das partículas iniciar em um ponto próximo ao resultado objetivo da função fitness.

6 FERRAMENTAS UTILIZADAS

- g++: Compilador que integra o pacote de ferramentas GCC (GNU Compiler Collection), pacote nativo em distribuições Linux.
- c++: Linguagem de programação de alto desempenho, superconjunto da linguagem C. Utilizado para construir o algorítimo PSO.
- Python 3.5: Linguagem de programação de alto nível, utilizada no trabalho para abrir os arquivos de saída da execução do PSO;
- matplotlib: Ferramenta disponível para Python para plotagem de gráficos.

7 REFERÊNCIAS E OUTROS MATERIAIS BASE

[1]. Página 4. Disponível em: http://www.revistaeep.com/imagens/volume11_02/cap01.pdf

[2]. Página 13. Disponível em: http://antigo.nuclear.ufrj.br/DScTeses/Canedo/Tese_Final_Canedo.pdf

[3]. Página 3. Disponível em: https://fei.edu.br/sbai/SBAI2007/docs/30400_1.pdf

[4]. Fórmula 15, página 12. Disponível em: https://www.researchgate.net/publication/229157888_Empirical_Review_of_Standard_Benchmark_Functions_Using_Evolutionary_Global_Optimization

[5]. Página 6. Disponível em: https://mpra.ub.uni-muenchen.de/2718/1/MPRA_paper_2718.pdf

[6]. Disponível em: https://www.cs.unm.edu/~neal.holts/dga/benchmarkFunction/schafferf6.html